Autonomous Navigation System Based On TurtleBot

Pletta A [*†], Karthikeyan S [‡§]

Department of Electrical and Computer Engineering

National University of Singapore

Summer 2019

[*]apletta@wisc.edu

[†]Pletta A completed this project as a research intern at National University of Singapore

[‡]ks6@illinois.edu

[§]Karthikeyan S completed this project as a research intern at National University of Singapore

Abstract

Autonomous navigation can be a very beneficial thing for society. For example, social robots can clean rooms, serve as a waiters in a restaurant, provide companionship, carry out tasks, provide safety, and more. Autonomous vehicles are also becoming increasingly popular and have the potential to improve transportation for people and trade all over the world. Thus, it is important to design algorithms and navigation systems so that these robots can function independently of an operator.

The project reviewed and evaluated key existing simulation, SLAM, and navigation techniques with TurtleBot and differential drive robots in general. Noticing a lack of similar open source support for four wheeled vehicles, a custom ROS package providing a framework for autonomous navigation and simulation of an Ackermann steering vehicle was created. A more efficient, robust, and generally compatible local motion planner algorithm was designed as an online adaptive MPC controller.

Preface

This report hopes to serve as a useful guide to people who are looking to explore navigation systems. This report gives a review on working with TurtleBot so it will not be as necessary to scour the Internet to search for assistance. Another goal of this report was to experiment with navigation systems and control methods. The report also explores creating an open source navigation system for autonomous cars and in particular those that have ackermann-steering. Hopefully upon reading this report people would be able to get a headstart with the various uses of ROS and control techniques to be better equipped for developing autonomous navigation systems for all sorts of vehicles.

Acknowledgments

The research conducted in this project was supported by the Controls and Simulation Lab at the National University of Singapore. We would like to thank Professor Shuzhi Sam Ge for his oversight and guidance in project design. We thank our colleagues Ji Ruihang, Liang Xiaoling, Wang Duansong, Zhang Yuexin, Fu Jianting, Liu Qiong, Xu Hao, Yan Mu, Wang Jiahong, and Ryan Kemmer who provided expertise that greatly assisted research. Acknowledgement is also given to numerous business professionals for their insight into common technology use in various fields.

We would also like to thank our friends and family for their encouragement and support throughout the duration of this project while we were at NUS far from our homes.

Further acknowledgment is extended to all those who may not have been mentioned but offered advice and inspiration for the work done on this project.

Contents

1 Introduction

The main goal of this project is to explore and evaluate indoor navigation systems for TurtleBot. This project is very relevant to numerous modern day applications. One popular example would be the iRobot Roomba. The iRobot Roomba is a TurtleBot that autonomously navigates around a room and performs the task of cleaning the room. Robots with navigating capabilities could also be used in disaster management to guide blind people, or assist the elderly. Self-driving cars have the capability to provide safer, more efficient, lower cost, and more accessible transport to people all over the world. The potential uses for robots with autonomous navigation capabilities are unlimited. Thus, research and development in this area is of considerable interest.

2 Theoretical Perspectives

2.1 TurtleBot

As the objective of this project is to create a system in which it would be possible for a TurtleBot to autonomously navigate a room, it is vital to get the necessary software first. As this was a robotics based project, it was only natural to use Ubuntu as the main operating system. This allows for the usage of ROS in this project. It is important to note that the versions of Ubuntu and ROS used is critical for using existing packages developed for TurtleBot. For instance, if one was working with a robot based on the TurtleBot2, one would have to utilise Ubuntu 14.04 and ROS-Kinetic Kame only whereas while working on a TurtleBot3 based robot, it would be necessary to use Ubuntu 16.04 and ROS-Kinetic Kame or newer. If the software requirements are not met, then it would be necessary to construct all the necessary packages from scratch.

Simulations can help accurately visualize the behaviour of the TurtleBot in numerous real world scenarios. Starting with simulations reduce costs and can be very time effective because no hardware is required to run tests. Therefore, a simulation model of the TurtleBot and the room that it would be navigating in must be made. The simulation software that should be used for this is Gazebo because it is user friendly and has many premade plug-ins

for ROS that make it easy to use. However, in Gazebo, one can only see the movements of the TurtleBot in its environment and not see what the TurtleBot is seeing through its sensors. In order to access TurtleBot sensors and better visualize sensor readings, a software called RViz should be used. RViz is also easy to use because it is designed with ROS.

Fundamental knowledge of using the terminal in Ubuntu is critical for this project. There are numerous packages that need to be installed and run via the terminal and there is a wide variety of errors that can occur. One must also have good debugging skills in order to deal with these issues. Once all required software have been installed, it is imperative that the simulated TurtleBot and ROS-Gazebo controller are able to form a connection between each other so they can send messages to each other. This makes it possible to control the TurtleBot while simultaneously receiving information from the TurtleBot. One common task called teleoperation controls robotic motion through keyboard commands given by a user on the master computer. It is possible to perform teleoperation on the TurtleBot through a variety of methods such as using different types of remote controllers or even sensors which have gesture recognition. Existing control and navigation techniques can inspire more new ways of controlling TurtleBots.

2.2 SLAM

Before the TurtleBot can navigate through a room, it must determine its position and evaluate the surrounding environment. Simultaneous localization and mapping (SLAM) is a common description of this problem. In some approaches, a robot is manually controlled to explore as much of the area as possible. During this process, the robot collects state estimates for various obstacles it encounters along its path. The robot also often also takes into account odometry data to determine the overall distance travelled. As all the data is being collected, it starts building a map of what it has seen. It is generally observed that the more area covered within a search space, the more accurate SLAM maps can become.

Many methods for SLAM exist in ROS with varying performance characteristics. These include HectorSLAM, gmapping, KartoSLAM, CoreSLAM, LagoSLAM, and Cartographer. gmapping is one of the most widely-used SLAM methods and uses a Rao-Blackwellized

particle filter to make it one of the most accurate and robust methods available [22]. Cartographer, developed by Google, employs a graph-based approach where nodes represent poses and features while edges represent constraints [8]. Cartographer can show confidence in space exploration, where less solid patches of white represent explored but lesser-known space. This could be advantageous in knowing where further exploration should be done to improve map accuracy.

2.3 Navigation

Robotic navigation has many approaches for getting a robot from one point to another efficiently while avoiding obstacles. Existing solutions vary in terms of speed, memory usage, accuracy, and applicability to certain environments. Path and control planning commonly broken down into global and local frames. Some implementations of these algorithms are used by existing ROS packages, namely the navigation and move_base packages for TurtleBot. The move_base package provides a navigation solution for differential drive robots. It includes all of the nodes, topics, and algorithms needed to perform robotic navigation with a differential drive robot, such as TurtleBot.

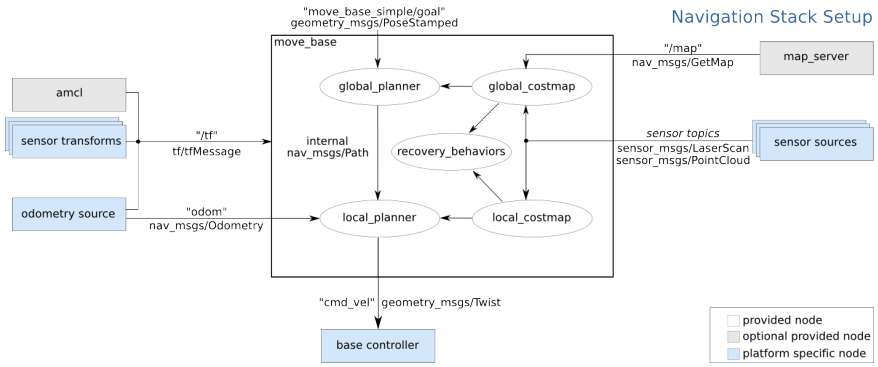

Figure 1: move_base ROS package designed for differential drive robots.

A global map is given to a global planner that sends a path to a local planner which then generates the control commands to move a robot along. Recovery behaviors exist in case the robot becomes stuck. For more information on the move_base package, see [12].

Common global path planning approaches include Djikstra, A*, and Genetic Algorithms (GA) among others [23]. Various other optimization techniques are often used as well.

Local planning algorithms generate commands for a robot's motors in order to achieve a desired motion and position. Many techniques exist for generating such commands, such as PID controllers, Dynamic Window Approach (DWA) [3], Timed Elastic Bands (TEB) [14], Model Predictive Controllers (MPC) [4], [6], [17] and more.

DWA samples feasible velocity space of translational and angular velocity pairs that make circular trajectories by extending constant control actions over a prediction horizon, selecting an optimal pair according to a cost function and then applying that pair for one time step [3]. Cost functions can be non-smooth so it can be used with grid-based cost functions. The current DWA algorithm uses a hard constraint optimization problem. However, it cannot currently be used for non-differential-drive (two wheeled) robots.

TEB discretizes a given trajectory and uses a continuous optimization approach to increase control resolution. It uses a soft constraints for optimization. Cost functions must be smooth. Only local solutions can be found but multiple trajectories can be evaluated [20]. These trajectories begin as the desired trajectory and stretch, similar to an elastic band, to accommodate local obstacles. TEB is computationally more expensive. However, it can be generalized for robots other than differential drive and has developing support for avoiding moving obstacles.

MPC uses a system model and optimization techniques to predict future outcomes for different control inputs and select a control sequence to meet a desired outcome.

The move_base package and ROS in general make experimenting with differential drive robots, such as TurtleBot, relatively easy and provide an excellent base for robotic development. While ROS is somewhat limited in application for real-time use [30], it is invaluable as a research tool because it is open source, has existing wiki-documentation, and a widely used Q&A network. Packages such as move_base can support the research and development that goes into non-ROS solutions. However, move_base can only be used for differential drive robots. Efforts made to convert the package to general use have been somewhat complex [29], [32] . In light of these observations, this project introduces a similarly easy-to-use package designed for four wheeled vehicles.

2.4 Ackermann steering

Ackermann steering is one of the most common steering methods used on four wheeled vehicles. There are many variations on Ackermann steering [1] and steering systems in general [26] to account for wheel slip angles, chassis dynamics, and more. With Ackermann steering, the left and right front wheels take on different steer angles in order to minimize excessive wheel slip and guide the vehicle more smoothly along a trajectory around an instantaneous center of rotation. In practice, the chassis of a vehicle often connects the two wheels to assist in achieving the proper steer angles [28].

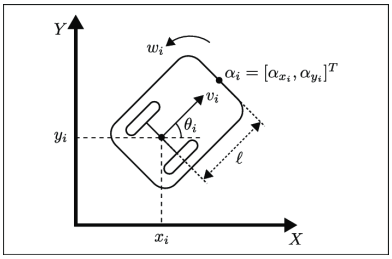

Figure 2: Differential drive robot.

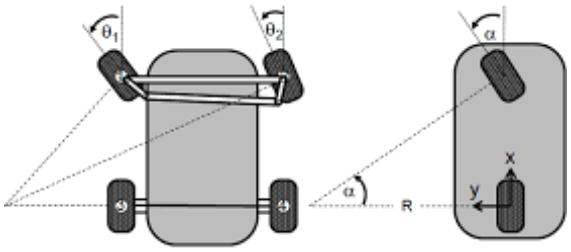

Figure 3: Ackermann steering model.

Ackermann steering vehicles operate with different kinematic constraints than differential drive robots, and thus need different controllers and local motion planning techniques. Efforts to support four wheeled vehicles in ROS include SBPL_Lattice_Planner which is intended to function as a global planner [9], teb_local_planner [21], and the Ackermann group [33] which attempts to consolidate Ackermann-related resources. The SBPL_Lattice_Planner and teb_local_planner have good performance but are specific algorithms intended to be implemented in larger navigation packages. The Ackermann page makes a good effort at organizing information but suffers from limited existing packages for Ackermann steering robots.

2.5 MPC

Noting the need for a common framework package designed for four wheeled vehicles, and taking into account the advantages and limitations of DWA and TEB local planners as they relate to such vehicles, this paper proposes an improved local planner algorithm using an

online adaptive MPC controller.

Model Predictive Controllers (MPC) have many characteristics desired for local planning. They use a system model and optimization solvers to predict and select future outcomes for different control inputs. This allows for control that is robust to non linearities, can use constraints and handle multi-input, multi-output systems. They can be solved online for live data or offline where control space can be evaluated for specified conditions (sometimes called explicit MPC). The primary disadvantage for MPC is that large processing power is needed, especially for online operation.

Discrete controllers divide time into time steps of a certain resolution and determine control actions for each time step. A prediction horizon sets how far into the future command sequences will be sampled and a control horizon sets how many time steps actions will be applied for. A cost function is used to analyze control sequence possibilities and then optimization techniques are used to find an optimal sequence. Control actions are applied over a control horizon (common practice is to use one time step) before the cycle of generating sequences and evaluating them repeats.

Controller parameters are typically tuned by analyzing desired response time, situational environment, computing capabilities, and more. Optimizations are often made to prediction horizon length and control sequences to improve processing efficiency. Adjustments can also be made for online vs offline, internal optimization techniques, and adaptive controllers. Adaptive MPC controllers allow for changing parameters and even algorithmic processes according to environment or system states. For a more complete MPC review and tuning guide see [25].

3 Methodology

3.1 TurtleBot

3.1.1 Gazebo simulation environment

In order to test performance under different conditions, functionality for loading the TurtleBot into custom maps was created. Different ways of making simulation environments were explored including Gazebo, Unity, and SolidWorks. Ultimately Gazebo was chosen for its compatibility with ROS and ease of use. Gazebo 9 was chosen as this was the compatible version for the Linux Ubuntu and ROS Melodic Morenia software setup being used [34]. Gazebo allows users to create their own models, use existing models from online libraries, and even has built-in assistance for tracing floor plans into 3D walls. There are also existing plug-ins to translate ROS commands into model motion in Gazebo. The result is a relatively user-friendly way of testing robotic control in simulation environments ranging from simple to very complex. Worlds can be changed by simply using an argument in the launch file that loads the robot and a world.

3.1.2 SLAM

Gazebo worlds can also be used for SLAM using simulated robot sensors. Of the SLAM methods reviewed, gmapping and Google Cartographer were the most accurate and so were selected for comparison.

It is also possible to perform SLAM with multiple robots. When doing SLAM with multiple TurtleBots, one must first create as many ROS namespaces as the number of TurtleBots they intend to use. It is then possible to run operations on each TurtleBot independently of each other. A map-merger tool can be run from the terminal. This package takes the individual maps built by each of the TurtleBots and combines them into a single map. This process happens in real time as the TurtleBots are going about their SLAM process.

3.1.3 State estimation

In the navigation stack, the robot's state is estimated using Adaptive Monte Carlo Localization (AMCL). The node uses many algorithms described in Probabilistic Robotics [24] and has parameter documentation on the package wiki [5]. The robot's location cannot be accurately determined until the user provides an initial pose estimate or odometry/sensor data is generated. By spinning in place, odometry data can help inform state estimation without the robot needing to move away from its position.

When the navigation stack is first launched the costmap should be properly aligned over environment obstacles. If the map is initialized as misaligned to true surroundings, a user can click the "2D Pose Estimate" button and then click/drag the green arrow into position to match the robot location in the map to where robot location in true surroundings should be. The robot can also be driven around the map until AMCL has determined a state estimate and automatically aligned the costmap to detected obstacles.

3.1.4 Navigation

Navigation with TurtleBot was performed using the navigation package which includes packages such as move_base to provide functionality. The primary considerations that can be made are those regarding a global planner, local planner, and the application for the robot.

The global planner can be changed between the carrot_planner [10], navfn [13], global_planner [9], and sbpl_lattice_planner [16] packages depending on the use case. All packages take in a global costmap of where obstacles are with an inflated safety zone around them before sending an optimal path to a local planner.

Navfn was chosen as a global planner as its performance is relatively good and, because it was decided to explore applications of path planning more than the algorithms themselves, extra functionality, such as that provided by global_planner, was not needed.

For the local planner, the navigation package uses dwa_local_planner by default but can also be adjusted to use teb_local_planner. The DWA approach was selected for use with TurtleBot for this project because of the relatively less-complex navigation space expected

(floorplans) and extensive documentation making it easy to adjust parameters to improve simulation performance.

Sending goals in the move_base package can be done by publishing a message to the "move_base_simple/goal" topic. This can be performed in several ways including using the rviz gui, publishing directly from the terminal command line, and by using a separate node. Nodes then can either publish preset commands, take in user input, or publish an offset location from other robot position for leader-follower operations.

One application that robotic navigation can be used for is taking items from place to place, such as a social robot moving within a cafe to serve patrons. This example was explored by setting locations and home on a map and allowing a user to input where they wanted to send the robot.

3.2 ackermann_nav package

Because the goal of this new ROS package is to provide general support for developing navigation systems for autonomous four wheeled vehicles, a pure Ackermann model was selected. It was decided to programmatically control the wheels separately instead of using mechanical linkage to simplify the simulation model.

3.2.1 Model

To allow for more general acceleration and heading commands, the vehicle's state is approximated by a point mass whose motion is constrained by Ackermann geometry. This was also done to allow for compatibility with the online adaptive MPC planner algorithm proposed by this paper. For a given turn, inner and outer wheel angles can be calculated using trigonometry with the only parameters needed being the vehicle length and width between wheel centers.

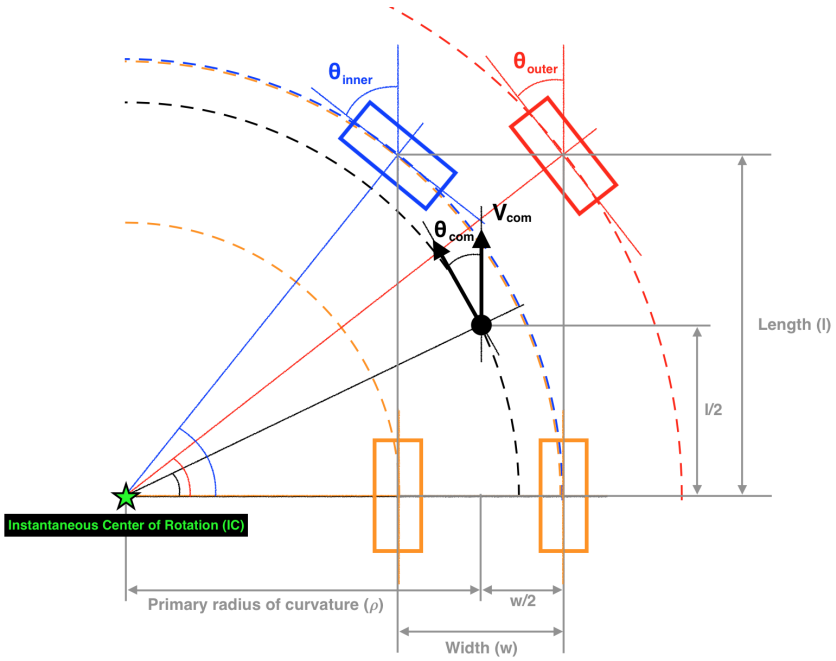

Figure 4: Ackermann steering geometry.

Steer angles for center of mass and inner and outer wheels can be written as follows.

$$tan(\theta_{com}) = \frac{l/2}{\rho} \tag{1}$$

$$tan(\theta_{inner}) = \frac{l}{\rho - w/2} \tag{2}$$

$$tan(\theta_{outer}) = \frac{l}{\rho + w/2} \tag{3}$$

where

θ_{com} is heading angle for center of mass

θ_{inner} is heading angle for inner wheel

θ_{outer} is heading angle for outer wheel

l is length of vehicle between wheel centers

w is width of vehicle between wheel centers

ρ is primary radius of curvature between center of mass and instantaneous center of rotation.

Rearranging for the primary radius of curvature results in the following. Note that absolute value for center of mass heading is used so that radius of curvature is always positive.

$$\rho = \frac{l}{2 \cdot tan(|\theta_{com}|)} \tag{4}$$

This radius of curvature can then be used to find the steer angles for the inner and outer wheels.

$$\theta_{inner} = tan^{-1}(\frac{l}{\rho - w/2}) \tag{5}$$

$$\theta_{outer} = tan^{-1}(\frac{l}{\rho + w/2}) \tag{6}$$

These calculations can be performed for any desired heading command. Angles are defined in radians for compatibility with Gazebo-ROS controller plug-ins where positive and negative angles result in left and right turns, respectively.

Algorithm 1: Ackermann steering control

input : θ_{com} heading angle for center of mass
output : θ_{inner} heading angle for inner wheel, θ_{outer} heading angle for outer wheel
parameter: l vehicle length, w vehicle width
$\rho \leftarrow \frac{l}{2 \cdot tan(|\theta_{com}|)}$
$\theta_{inner} \leftarrow tan^{-1}(\frac{l}{\rho - w/2})$
$\theta_{outer} \leftarrow tan^{-1}(\frac{l}{\rho + w/2})$
if $\theta_{com} > 0$ **then**
  $\theta_{left} \leftarrow \theta inner$
  $\theta_{right} \leftarrow \theta outer$
else if $\theta_{com} < 0$ **then**
  $\theta_{left} \leftarrow -\theta inner$
  $\theta_{right} \leftarrow -\theta outer$
else
  $\theta_{left} \leftarrow 0$
  $\theta_{right} \leftarrow 0$
end

Modeling of a basic four wheeled vehicle was done in Gazebo. Revolution joints were placed horizontally at each wheel and vertically at each steering block. Steering block rota-

tions are controlled separately by using a node running this algorithm. User input can be given to change the heading of the vehicle center of mass and a controller adjusts the wheels accordingly. Steering style is flexible and can be set from within the ackermann_controller node.

Taking note of the popularity of the move_base and ROS navigation packages for differential drive robots, the model and its controller were designed as part of a larger package with the goal of providing an easy to use, flexible ROS framework of computational nodes, the topics and messages they share, parameter files for nodes, and documentation supporting package use.

3.2.2 Package design

The ackermann_nav package is generally designed for the following information flow.

1. Robot state is initialized.
2. Obstacles are detected by sensors at positions relative to current position.
3. Global obstacle positions are calculated and fed to a global map.
4. Global path plan is produced for a defined goal and split into a list of checkpoints.
5. Local planner generates commands for robot to move to checkpoints while avoiding locally detected obstacles.
6. Controller turns planner commands into motor actions.
7. Robot state is updated using odometry and other sensor data.
8. Repeat steps 2-7 until goal is reached, though adjustments could be made to run global path planner a limited number of times throughout navigation process to improve efficiency.

The intention of this design is to allow different algorithms for global planners, local planners, state estimation, obstacle detection, SLAM, and controllers to be swapped out independently. Parameters can also be set and added/removed for individual nodes.

Other important design notes

- Drivers for specific sensors can be used by matching topic interfaces.
- obstacle_locations node classifies local and global obstacles to improve planner speed.
- global_map node can use premade SLAM maps or data only.
- Checkpoints fed to local planner to allow for more direct navigation.
- Using acceleration and heading commands for local planner allows for more general use.
- ackermann_controller expects input from only one source to prevent confusion.
- Other topics and nodes can be added easily using premade templates.

New message types and topics could also be made and published/subscribed to allow for passing other data between nodes. The topics were set up to allow with the intention of providing the most generality possible. Topic message types and the connecting nodes can be cleanly visualized in a block diagram intended to be updated with package changes.

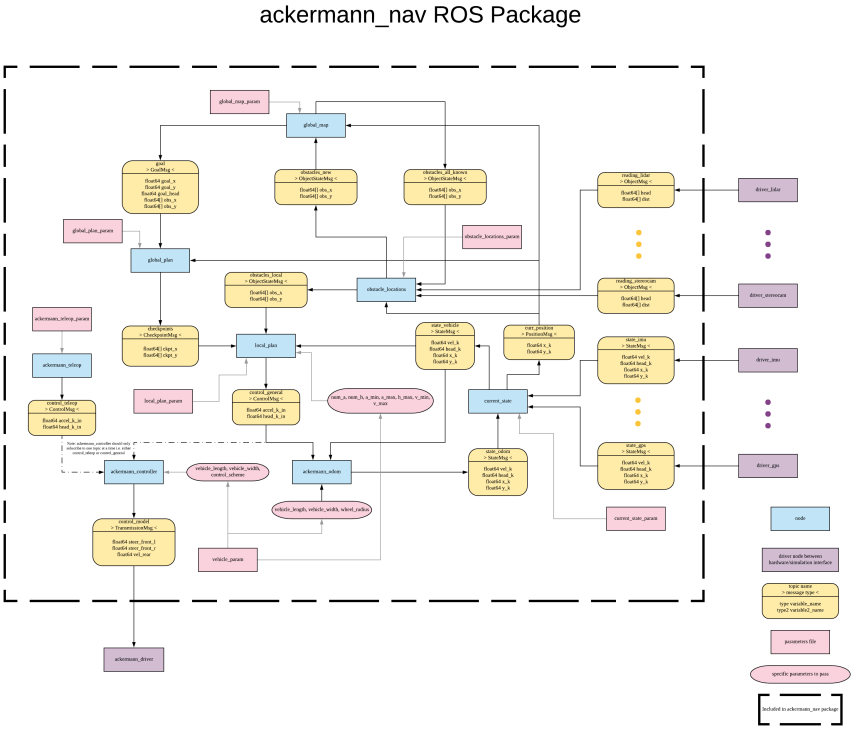

Figure 5: ackermann_nav ROS package structure.

3.3 MPC planner

3.3.1 Controller design

The adaptive MPC path planner that will be described here is designed with autonomous cars in mind, so the prediction horizon needs to be large enough to allow for braking to a stop but not so long that a car couldn't respond to obstacles appearing suddenly. Similarly, a fast sample and execution time are needed so the system can read and react to the environment even with limited reaction time.

An adaptive MPC was selected so available control actions and prediction/control horizons could be changed for fast or slow moving conditions. This was done to improve computation efficiency and give greater variety of control under varying velocities. When a car is moving slow, such as when parking, it may need to turn more and can likely stop in only one or two braking commands. When a car is moving fast, such as on a highway, it needs a larger prediction horizon because it will take longer to stop. Large steering changes are

not desired as they could compromise vehicle stability. Therefore, depending on a threshold velocity, the controller either samples command sequences with many control actions and a smaller prediction horizon or less control actions with a larger prediction horizon. This allows for improved control sequence selection while still maintaining efficient execution.

The MPC controller is designed to take in reference information of obstacles and checkpoints to navigate to, as well as a cost function. Control and prediction horizon parameters vary with vehicle state, making it an adaptive MPC. Acceleration and heading change commands can be passed to an actuation controller and converted to motor inputs before initiating vehicle motion and performing a state estimation update. Separating the MPC controller from the actuation controller allows the MPC controller to be used for more general applications.

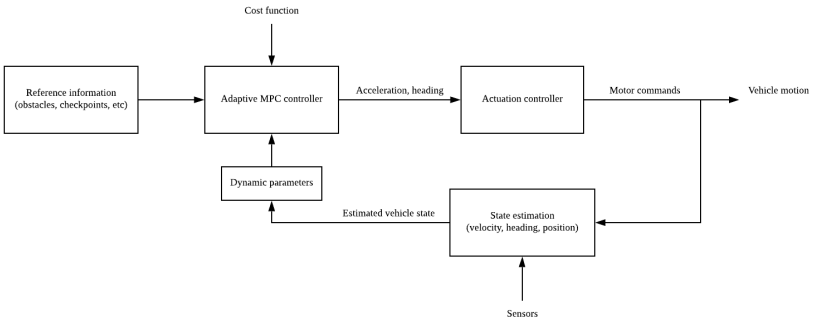

Figure 6: Online adaptive MPC controller block diagram.

3.3.2 Model

The vehicle being controlled is represented as a point mass whose state is described by a velocity, global heading, and x/y position. A single point was used so the controller can still be applied to more complex models, such as the ackermann_nav package described by this paper. This also simplifies calculations so the local planner can generate commands faster and thus operate on a finer time step scale.

Checkpoints are represented by lighter green circles with a buffer and the goal is represented by a darker green circle with a buffer.

State information is calculated after the control horizon is applied. The next state can

be predicted using a kinematic model and then combined with sensor data using a particle filter or some other technique. This state is used in a feedback loop for the next time step.

Obstacles are represented as either circles or lines where a buffer region is applied around the shape and used for obstacle avoidance. When evaluating control sequences, only buffer regions that could be reached within the prediction horizon are considered to reduce computation time. The buffer size can be designed as the maximum distance between the point mass and the outer contour of the vehicle to prevent any collisions.

3.3.3 Algorithm

At every time step, control sequences are generated from possible acceleration and heading control actions using a permutation of k items from a list of n elements. The cost over all time steps in the prediction horizon is calculated for each control sequence upon generation and compared to the optimal cost encountered thus far.

Algorithm 2: Control sequence generation and selection

input : L_{combos} list of possible control pair combinations for acceleration and heading change control actions, H_p prediction horizon
output : $S_{optimal}$ optimal control sequence to be applied

$c_{optimal} \leftarrow \infty$
Function `permutate`(L_{combos}, H_p)`:`
  while *all permutations not generated* **do**
    $S_{new} \leftarrow$ control sequence for new permutation
    $c_{new} \leftarrow \text{cost}(S_{new}, H_p)$
    if $c_{new} < c_{optimal}$ **then**
      $c_{optimal} \leftarrow c_{new}$
      $S_{optimal} \leftarrow S_{new}$
    end
  end
  return $S_{optimal}$
End Function

To implement for testing, a recursive algorithm for permutation generation was adopted and modified from [35]. The permutation function was designed to take on extra parameters to prevent having to use global variables.

The time complexity for this algorithm is

$$O(n \cdot H_p)$$

where

n is the Cartesian Product of acceleration and heading change action lists

H_p is prediction horizon length

and

$$n = a \cdot h$$

where

a is number of acceleration actions

h is number of heading change actions

The overall time complexity is then:

$$O((a \cdot h)^{H_p}) \tag{7}$$

Therefore, for maintaining a consistent computation time, the prediction horizon can be reduced for longer control action lists and extended for shorter control action lists. Using an adaptive MPC to vary the prediction horizon and control action lists with the vehicle state takes advantage of this observation.

3.3.4 Cost function

A linear cost function is used to calculate and sum scores for different variables being analyzed. Leaving the final cost unconstrained results in a soft constraint problem which allows the algorithm to always find a feasible, even if very costly, solution. This is important so that if an unexpected situation is encountered control can still be applied without a complete shut down.

The optimization problem can be written as:

$$\text{minimize} \quad C_{total}$$

$$\begin{aligned}
\text{subject to} \quad C_{total} =& \Lambda \cdot W^T \\
W =& \begin{bmatrix} W_a\, W_\phi\, W_{dist}\, W_{obs}\, W_{lineCross}\, W_{vmin}\, W_{vmax}\, W_{reverse}\, W_{fast}\, W_{prev} \end{bmatrix} \\
\Lambda =& \begin{bmatrix} \lambda_a\, \lambda_\phi\, \lambda_{dist}\, \lambda_{obs}\, \lambda_{lineCross}\, \lambda_{vmin}\, \lambda_{vmax}\, \lambda_{reverse}\, \lambda_{fast}\, \lambda_{prev} \end{bmatrix} \\
\lambda_a =& \sum_{i=0}^{H_p} a_i^2 \\
\lambda_\phi =& \sum_{i=0}^{H_p} \phi_i^2 \\
\lambda_{dist=} & \sum_{i=0}^{H_p} \sqrt{(x_i - goal_x)^2 + (y_i - goal_y)^2} \\
\lambda_{obs} =& \sum_{i=0}^{H_p} \sum_{j=0}^{l_{obs}} c_{obs} \cdot \delta(\sqrt{(x_i - obstacles_{jx})^2 + (y_i - obstacles_{jy})^2}) \leq r_{buffer}) \\
\lambda_{lineCross} =& \sum_{i=0}^{H_p} \sum_{j=0}^{l_{lines}} c_{lineCross} \cdot \delta(lineCrossed) \\
\lambda_{vmin} =& \sum_{i=0}^{H_p} c_{vmin} \cdot \delta(|v_i| < v_{min}) \\
\lambda_{vmax} =& \sum_{i=0}^{H_p} c_{vmax} \cdot \delta(|v_i| > v_{max}) \\
\lambda_{reverse} =& \sum_{i=0}^{H_p} c_{reverse} \cdot \delta(v_i < 0) \\
\lambda_{fast} =& \sum_{i=0}^{H_p} (v_i - v_{max})^2 \\
\lambda_{prev} =& \sum_{i=0}^{H_p} \sum_{j=l_m}^{l_p} c_{prev} \cdot \delta(\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}) \leq r_{prev}) \\
l_m =& p - min(m, p) \\
l_p =& p - s \\
& W, \Lambda \in R^{10}
\end{aligned}$$

where

$W_a, W_\phi, \ldots, W_{prev}$ are the weights for their respective cost variables (sub-scripted)

$\lambda_a, \lambda_\phi, \ldots, \lambda_{prev}$ are summed cost values (sub-scripted)

H_p is prediction horizon

a_i is acceleration at time step i of H_p

ϕ_i is heading at time step i of H_p

x_i, y_i are x and y position at time step i of H_p

$goal_x, goal_y$ are x and y position of goal

l_{obs} is number of locally detected obstacles

c_{obs} is cost of entering an obstacle buffer region

$obstacles$ is an array of locally detected obstacles

r_{buffer} is buffer radius assigned to obstacles

$\delta(u)$ is 1 when u True, 0 otherwise

l_{lines} is number of locally detected lines

$c_{lineCross}$ is cost of crossing a line

$lineCrossed$ is a boolean variable for whether a line has been crossed or not. Implementation to determine this may vary.

v_i is velocity at time step i of H_p

v_{min} is min desired velocity

c_{vmin} is cost of going below min desired velocity

v_{max} is max desired velocity

c_{vmax} is cost of going over max desired velocity

$c_{reverse}$ is cost of moving with negative velocity (moving in reverse)

λ_{fast} is cost of not moving fast (near max desired velocity)

c_{prev} is cost of moving near a previously visited location

x_j, y_j are x and y position at time step j along previous path

r_{prev} is avoidance radius given for moving near previous locations

l_m is beginning of previous path locations to check

l_p is end of previous path locations to check

m is number of previous locations to save in memory

p is overall number of previous locations

s is number of steps behind current step to finish checking previous locations

Weights, cost sum values, and other parameters need to be tuned for performance goals. Some sum values also scale with unit scales. For example, acceleration sum will have a larger value if using a scale of 10 instead of 0.1 so this may result in more cost than predicted. Future work could be done to make these sums values independent of unit scale or at least all relative to the same scale.

4 Findings and Data Analysis

4.1 TurtleBot

4.1.1 Gazebo performance

The simplest Gazebo environment is an empty world. Custom worlds ranging from simple to complex can also be created relatively easily. Many pre-made objects exist in the Gazebo library including cones, buildings, vehicles, street signs, and more.

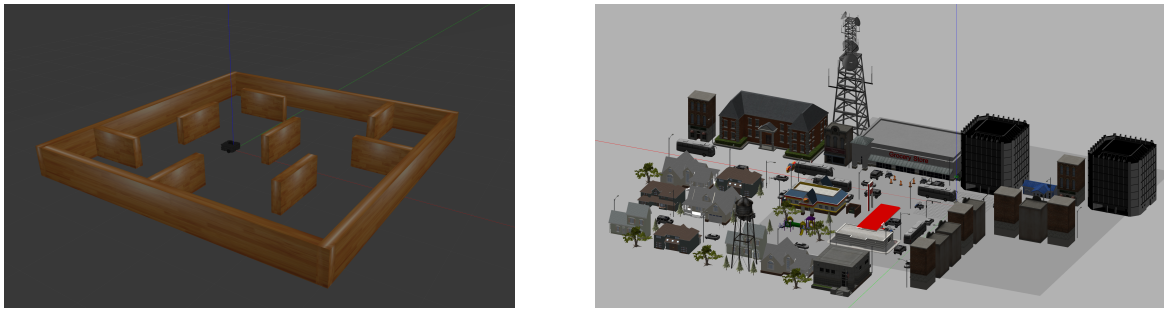

Figure 7: Examples of simple and complex Gazebo worlds

4.1.2 SLAM analysis

Among tested SLAM methods, Cartographer had superior map accuracy over gmapping but was found to be more difficult to implement with existing TurtleBot packages for performing SLAM. Therefore, gmapping was selected for future SLAM use because it was still reasonably accurate and more compatible for doing SLAM with multiple TurtleBots.

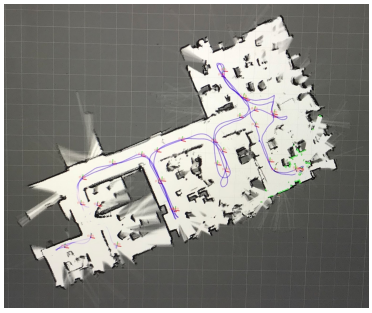

Figure 8: SLAM with Cartographer demo.

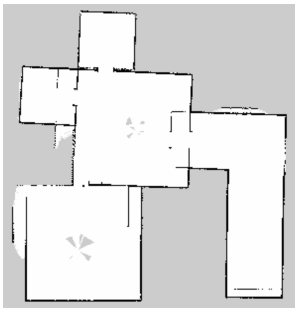

Figure 9: SLAM with gmapping.

There are numerous parameters involved with gmapping SLAM. By appropriately adjusting the values one can get a more accurate map. For example, one can clearly notice the difference between unset parameters and that of the adjusted parameters in figures 10 and 11.

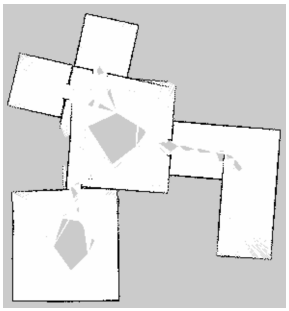

Figure 10: Before setting parameters.

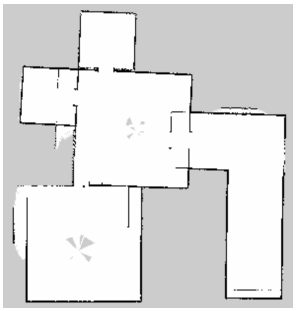

Figure 11: After setting parameters.

It is generally observed that rooms that are bigger and have more free space cause greater errors than maps of smaller rooms. One way of handling this problem is by increasing the value of the minimumScore parameter. The minimumScore parameter is the outcome of a good scan matching. The lstep and astep parameters can also be reduced for greater accuracy. The lstep is the linear step size for the scans while the astep is the angular step size

for the scans. It can also be beneficial to increase the resampleThreshold and the particles parameters. The resampleThreshold helps with the loop closure feature in gmapping SLAM while particles defines the number of particles in the filter for gmapping SLAM. Adjusting the values of these parameters allows one to get accurate maps using gmapping SLAM.

Upon tuning the gmapping parameters, it is possible to use multiple robots to perform SLAM. It would be beneficial to use multiple robots while trying to build the map of a location, because having more robots cover a greater area could help speed up the map making process. This would be especially useful in disaster situations when robots are searching for casualties in hard to reach areas.

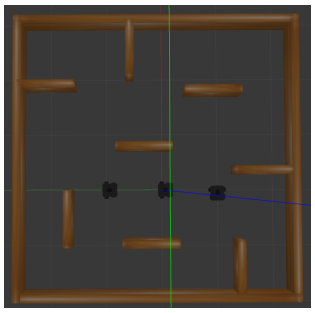

Figure 12: Multiple robots loaded in Gazebo.

Figure 13: Initial merged SLAM map.

Figure 14: Individual maps from each of the TurtleBots.

4.1.3 State estimation

The "2D Pose Estimate" button can be used by clicking/dragging a green arrow to specify robot pose. Setting robot pose helps align the robot and local costmap within a global environment. This alignment is essential for navigation techniques. It was observed in some

cases that driving the robot via teleop was needed to determine location in local costmap before a 2D Pose Estimate could be made by the user. In general, looking at the Gazebo simulation environment to see global robot pose was most helpful for complicated cases.

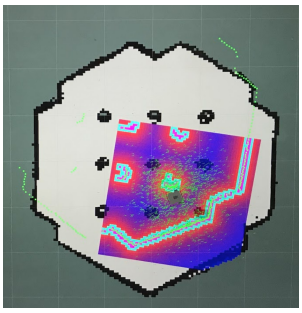

Figure 15: Map misaligned.

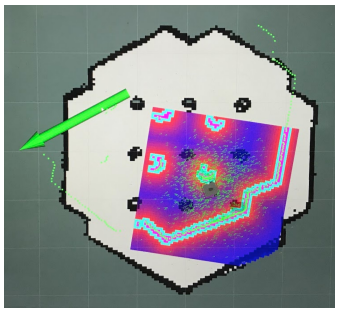

Figure 16: Set 2D Pose Estimate.

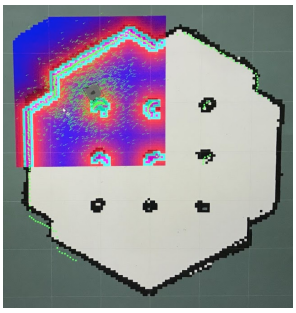

Figure 17: Map aligned.

AMCL state estimation accuracy improves automatically for a spinning robot. Predicted poses are shown as small green arrows in the local costmap. The AMCL state estimation variation decreases while accuracy increases as the robot spins. Thus, this technique could effectively provide localization in situations when a user cannot interact with the robot to give an initial pose estimate.

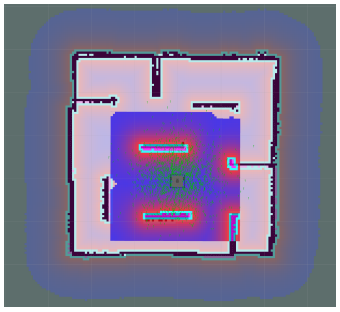

Figure 18: Initial pose

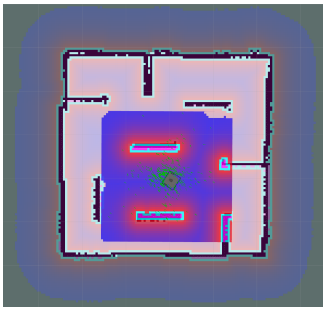

Figure 19: Robot spinning.

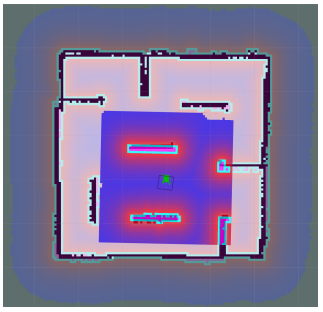

Figure 20: Final state estimate.

4.1.4 Navigation techniques

Using the rviz gui, a user can click the "2D Nav Goal" button and then click/drag green arrow into final position and heading. Robot is then controlled from initial position to goal using the move_base package. Both the global planner and local planner paths are displayed

in rviz as black and yellow lines, respectively. Obstacles are shown on global map as black and on local map as cyan with a red safety inflation zone around them. Blue on the local costmap represents safe navigation space. LiDAR data is visualized as green dots where obstacles are detected.

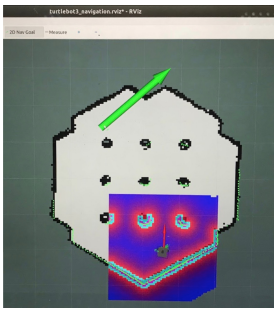

Figure 21: Set 2D Nav Goal.

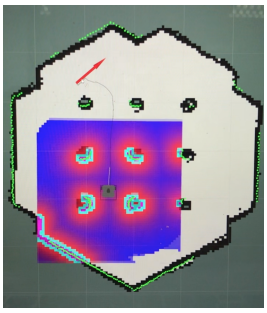

Figure 22: Robot begins following path.

A system was designed for allowing user input to navigate a robot between preset locations on a map. Robot can travel between any two locations. Functionality could be built for patrolling between locations. Work was attempted on leader-follower implementation but not completed at the time of writing. In all cases, the node handles the method to be employed and then publishes a message to the "move_base_simple/goal" topic. Any form of publishing goals to the topic is highly effective with the TurtleBot navigation stack.

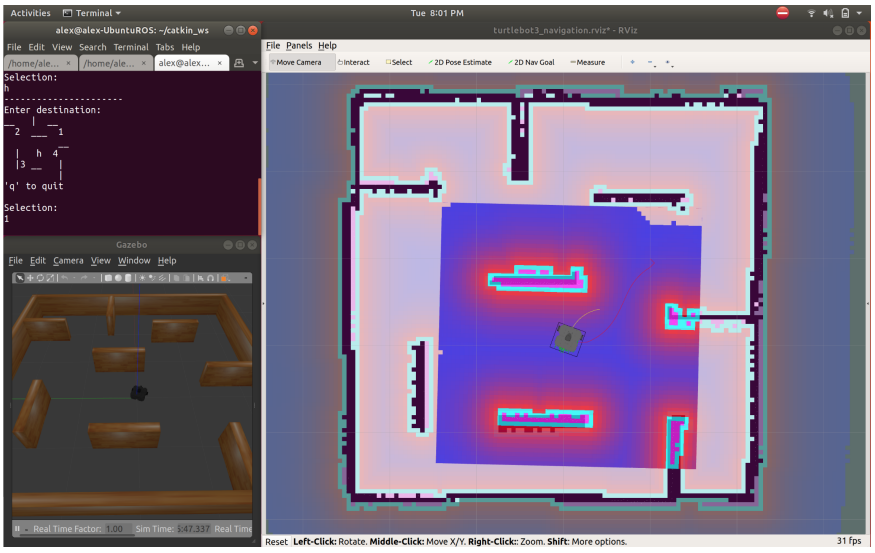

Figure 23: Navigation performed by allowing a user to specify preset destination locations.

4.2 ackermann_nav package

4.2.1 Model validation

Joint locations for Ackermann steering Gazebo model. Rear wheels spin around the same axis and front wheels spin about independently controlled steering blocks. Red arrows indicate joint locations. The example shown is for a left turn, following the views for previous Ackermann steering geometry.

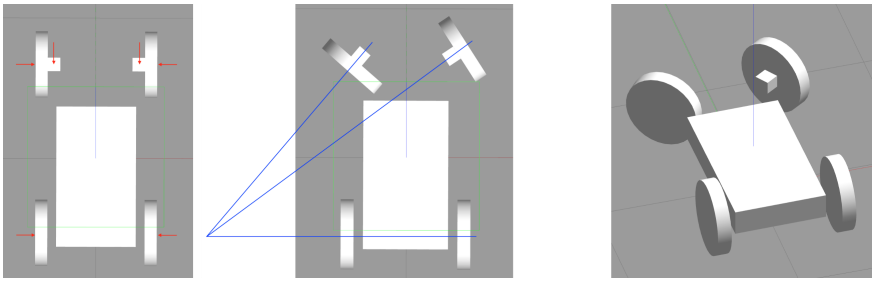

Figure 24: Ackermann model and geometry validation.

4.2.2 Package construction

All nodes and topics were set up and launched with their associated parameter files according to the block diagram specified.

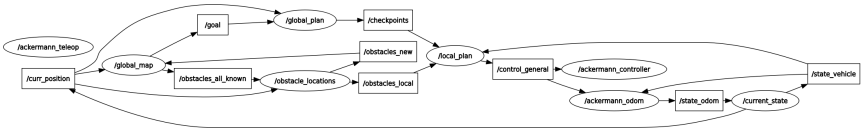

Figure 25: ackermann_nav rqt graph of connected nodes and topics.

Templates for nodes that publish and subscribe to varying numbers of topics were also created and designed so that topics and their message types can be easily adjusted.

4.2.3 Obstacle detection

One feature that was tested on the Ackermann-steering autonomous car was obstacle detection. This was done by adding a simulated Hokuyo LiDAR to the car to detect the distance and the heading of objects in meters. For example, once an object comes within 3m of

the car, an alert is displayed that an obstacle is detected and that the car should change directions.

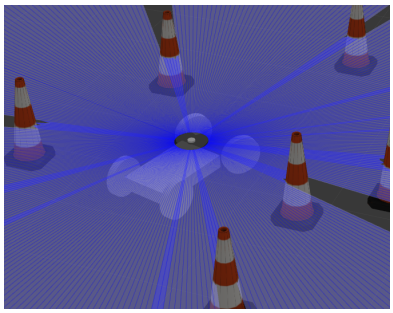

Figure 26: Ackermann model in custom world with Hokuyo LiDAR sensor.

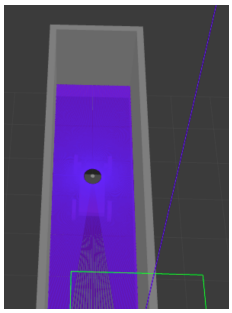

Figure 27: Vehicle surrounded by walls.

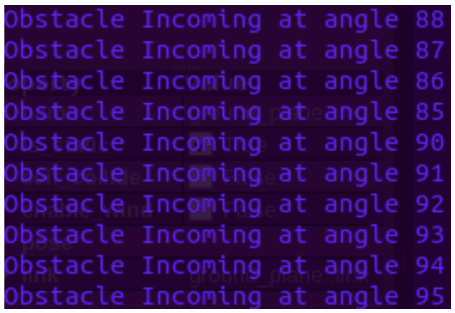

Figure 28: Incoming obstacle alert message.

4.3 MPC planner

4.3.1 Model analysis

To assist with simulation, a box was drawn around the point mass for visualization. The prediction horizon is shown as orange "x" marks extending ahead of the vehicle. The detection zone for local obstacles is shown as a light blue circle. Past locations kept in memory are shown as blue dots behind the vehicle (while moving) and the global path is saved as well (plotted once goal is reached).

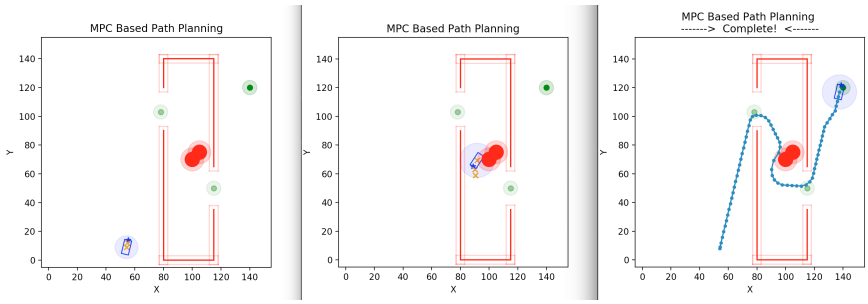

Figure 29: Navigation through simulation environment.

As velocity increases, the prediction horizon and local obstacle detection zone extend. This behavior allows for increased vehicle control while moving slow and increased detection range while moving fast.

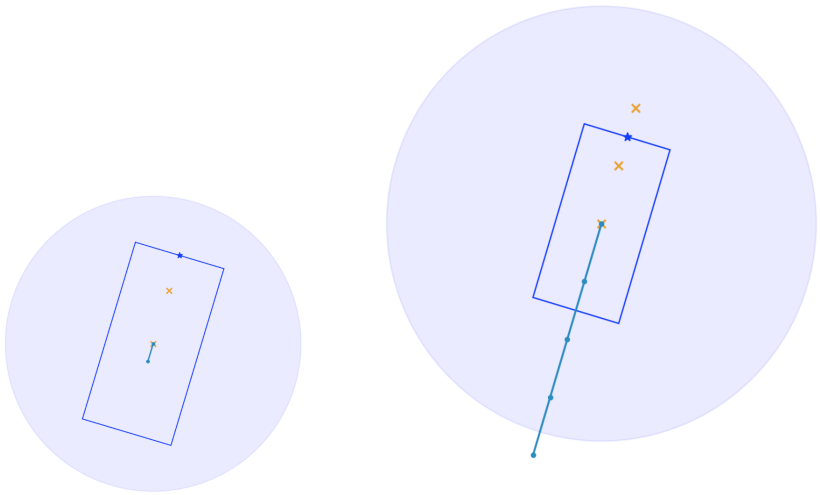

Figure 30: Adaptive prediction horizons and local obstacle range.

4.3.2 Parameter tuning

Position and path planning efficiency can change drastically using different parameters such as parameter combinations and checkpoints. Resulting paths are for the same map but different memory levels.

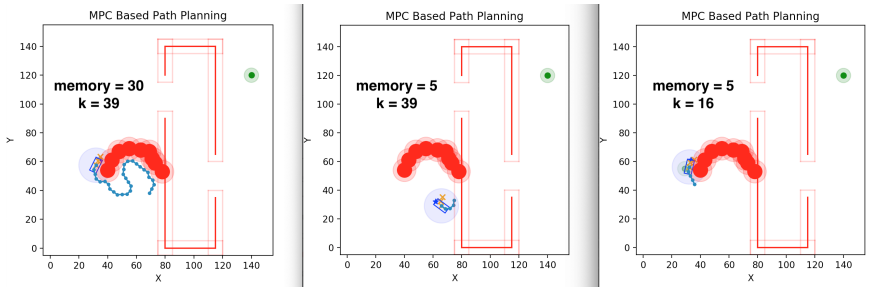

Figure 31: Simulation testing at different time steps with varying memory levels.

For the example set of obstacles that turn the vehicle away from the goal, using a large memory allows a successful path to be found but takes many timesteps (k). However, by using a checkpoint the same position can be reached in far fewer timesteps and while saving computation time. Thus, for complete navigation it would be best to employ some sort of global planner to place these checkpoints prior to beginning this local planner. In situations where checkpoints are not possible, a certain number of previous locations can be saved in memory and used to prevent traveling in loops, though this can take extra computation

time.

4.3.3 Obstacle representation

It was found that with a large enough time step relative to the circle sizes, the planner could make a path that jumps over the smallest section between circles.

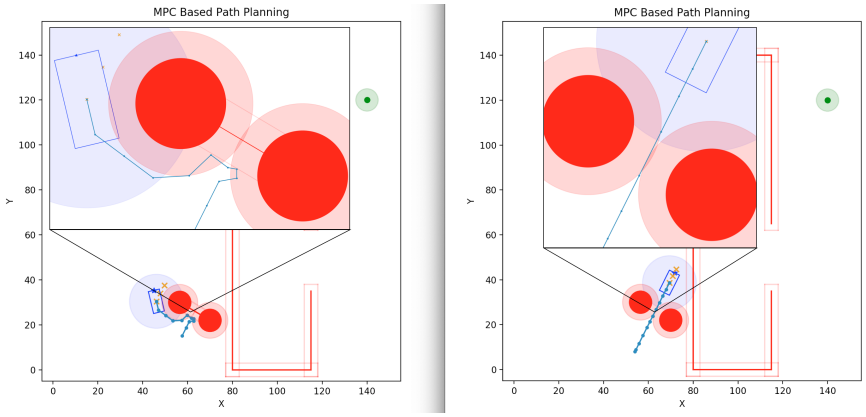

Figure 32: Planner able to pass between close circles but not when line is present.

Using lines prevents the planner from finding valid but undesired paths that jump over small overlaps in circular buffer regions. They also allow long obstacles can be represented with only one variable, reducing computation time. Future designs could use lines to represent more complex polygon obstacles.

4.3.4 Algorithm performance

Computation time is reduced by over an order of magnitude by only checking for collisions with local obstacles that could be hit within a local obstacle detection zone. Comparison shown is for first time step execution of simulation checking all global lines (left) and only local lines (right). This performance improvement is observed for both circular and line obstacles.

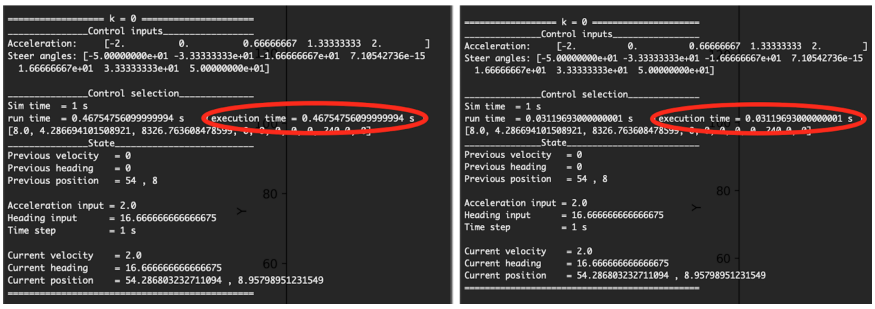

Figure 33: Execution time comparison for different obstacle detection techniques.

Run time was analyzed for varying sizes of acceleration actions, heading actions, and prediction horizon and compared to expected time complexity.

Trial	Acceleration actions (a)	Heading actions (h)	Prediction horizon (Hp)	Predicted complexity	Execution time [sec]			Average execution time
Small Hp, large a & h	5	7	2	1225	0.0333	0.0339	0.0319	0.03303333333
Large Hp, small a & h	4	3	3	1728	0.0596	0.0624	0.0612	0.06106666667
Double a only	10	7	2	4900	0.1285	0.133	0.1274	0.1296333333
Double h only	5	14	2	4900	0.1287	0.1285	0.1288	0.1286666667
Larger Hp only	5	7	3	42875	1.4882	1.5004	1.5482	1.512266667

Figure 34: Data table for algorithm execution time testing.

The resulting data was normalized to the largest observed time values and compared to observe relative trends.

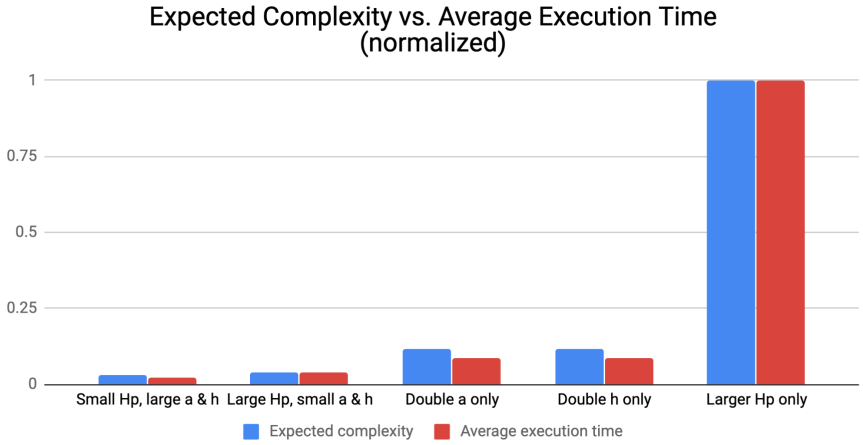

Figure 35: Chart of expected vs measured run times.

Comparing the two data sets shows the same relative trends in actual run time as would be expected from a predicted time complexity analysis using eq. (7). This proves that

the time complexity analysis is a good upper bound model for evaluating how changes in control actions and prediction horizon can affect controller performance. More data could be collected to further optimize different combinations of control actions and prediction horizon.

5 Discussion

5.1 TurtleBot

Tuning the ROS navigation stack is a non-trivial process that involves understanding the effects of and relations between many different parameters. These parameters include the global and local planners, costmap, localization (specifically for AMCL), recovery behaviors, and robot, sensor, and object representation. A more complete tuning review can be found at [27].

5.2 ackermann_nav package

The impact and usefulness of this package remains to be seen. Dependent factors include the algorithms used, public awareness of package, and code/documentation maintenance. It is not the intent to provide a full navigation system yet, rather the framework for users to implement their own algorithms. However, it is likely that some of the first implementations will be added to the package so users can adopt a working system early on similar to the move_base and ROS navigation packages for differential drive robots.

Future work can be done to make the user experience even more friendly for making package adjustments, such as changing connecting topics and algorithms via parameter files and providing tutorials on use. Feedback from users will drive adjustments to the framework and package documentation with the primary goal of making the package easy and friendly to use.

The source code for the ackermann_nav package has been made open source on GitHub, along with more in-depth supportive documentation, at [18].

5.3 MPC planner

By inserting the cost function into the permutation generation algorithm and passing along the single best control sequence thus far, a global optimum can be found every iteration without need for other optimization solvers. This reduces overall computation time so the controller can be used at smaller time step sizes for finer control.

Compared to existing local planners DWA and TEB methods, this proposed MPC has numerous advantages. First off, the full control space can be sampled at each step of the prediction horizon whereas DWA only extends constant actions. This allows for more complex control sequences that can better adapt to changing conditions. While TEB requires a given trajectory to modify, MPC needs only a checkpoint to simultaneously find a globally optimal local trajectory and the associated control sequence. The cost function used allows for tuning of more parameters to better adjust planner performance. Computation time is reduced by passing a globally optimal control sequence through permutation generation, removing the need to apply an optimizer. Generating lateral acceleration and heading change commands and allowing a vehicle-specific controller to convert commands to motor inputs means this planner can be used for a wide variety of applications.

6 Conclusion

6.1 TurtleBot

Existing packages and support for TurtleBot was explored and extended for custom simulation worlds, multiple robot SLAM and navigation to preset locations using via user input. Further exploration could be done on finding optimal parameters for the global_planner package, implementing leader-follower navigation, and adjusting the navigation package for varied applications including personal companion robots, delivery robots, swarm navigation, and for searching space for objects within a map.

6.2 ackermann_nav package

It is the intention of this project that the contribution of this package allows future researchers and other users to more easily experiment with navigation techniques as they relate to four wheeled vehicles, in a similar spirit to how there is lots of existing support for differential drive robots. This will hopefully allow more efficient, safer, and advanced navigation algorithms and techniques to be conceived and implemented so autonomous navigation can better accomplish real world tasks.

6.3 MPC planner

The MPC control algorithm developed in this project can be used in online applications to avoid moving obstacles and has improved efficiency due to cost calculation upon control sequence generation. The planner is also made robust to unexpected inputs by using a soft constraint cost function and adaptive prediction horizon and potential control actions based on system state. While designed for four wheeled autonomous vehicles, it could be used more generally for any robot using a controller for 2D acceleration and heading change commands. This algorithm could be improved by making the permutation algorithm more efficient and using advanced neural nets or other optimization techniques for tuning weights and cost sum values to meet performance goals.

6.4 Project summary

This project explored robotic navigation techniques in Gazebo and ROS through existing TurtleBot packages. This confirmed ROS as an excellent open source and well supported resource for researching and developing robotics software. It was observed that there was limited support for four wheeled vehicles, so a custom package was created with the intention of providing a navigation framework for other ROS users. Finally, a local planner was designed as an online adaptive MPC controller with improved speed and control performance compared to other existing methods.

References

[1] Biswal, S., Prasanth, A., Udayakumar, R., Sankaram, M. N., and Patel, D. (2017). *Design of steering system for a small Formula type vehicle using tire data and slip angles*. In MATEC Web of Conferences (Vol. 124, p. 07007). EDP Sciences.

[2] Bourke, Paul (1988, October). *Minimum Distance between a Point and a Line*. Retrieved from http://paulbourke.net/geometry/pointlineplane/

[3] Fox, D., Burgard, W., and Thrun, S. (1997). *The dynamic window approach to collision avoidance*. IEEE Robotics & Automation Magazine, 4(1), 23-33.

[4] Frasch, J. V., Gray, A., Zanon, M., Ferreau, H. J., Sager, S., Borrelli, F., and Diehl, M. (2013, July). *An auto-generated nonlinear MPC algorithm for real-time obstacle avoidance of ground vehicles*. In 2013 European Control Conference (ECC)(pp. 4136-4141). IEEE.

[5] Gerkey, Brian P. (2019, June). *amcl*. Retrieved from http://wiki.ros.org/amcl

[6] Guo, H., Shen, C., Zhang, H., Chen, H., and Jia, R. (2018). *Simultaneous trajectory planning and tracking using an MPC method for cyber-physical systems: A case study of obstacle avoidance for an intelligent vehicle*. IEEE Transactions on Industrial Informatics, 14(9), 4273-4283.

[7] Gutjahr, B., Gröll, L., and Werling, M. (2016). *Lateral vehicle trajectory optimization using constrained linear time-varying MPC*. IEEE Transactions on Intelligent Transportation Systems, 18(6), 1586-1595.

[8] Hess, W., Kohler, D., Rapp, H., and Andor, D. (2016, May). *Real-time loop closure in 2D LIDAR SLAM*. In 2016 IEEE International Conference on Robotics and Automation (ICRA)(pp. 1271-1278). IEEE.

[9] Lu, David (2019, June). *global_planner*. Retrieved from http://wiki.ros.org/global_planner?distro=melodic

[10] Marder-Eppstein, Eitan (2019, June). *carrot_planner*. Retrieved from http://wiki.ros.org/carrot_planner?distro=melodic

[11] Marder-Eppstein, Eitan (2019, June). *dwa_local_planner*. Retrieved from http://wiki.ros.org/dwa_local_planner?distro=melodic

[12] Marder-Eppstein, Eitan (2019, June). *move_base*. Retrieved from http://wiki.ros.org/move_base

[13] Marder-Eppstein, Eitan (2019, June). *navfn*. Retrieved from http://wiki.ros.org/navfn

[14] Marin-Plaza, P., Hussein, A., Martin, D., and Escalera, A. D. L. (2018). *Global and local path planning study in a ros-based research platform for autonomous vehicles*. Journal of Advanced Transportation, 2018.

[15] Peng, Z., Wang, D., Chen, Z., Hu, X., and Lan, W. (2012). *Adaptive dynamic surface control for formations of autonomous surface vehicles with uncertain dynamics.* IEEE Transactions on Control Systems Technology, 21(2), 513-520.

[16] Phillips, Michael (2019, March). *sbpl_lattice_planner.* Retrieved from http://wiki.ros.org/sbpl_lattice_planner

[17] Phung, D. K., Hérissé, B., Marzat, J., and Bertrand, S. (2017). *Model predictive control for autonomous navigation using embedded graphics processing unit.* IFAC-PapersOnLine, 50(1), 11883-11888.

[18] Pletta, Alex and Sundaram, Karthikeyan (2019, July). *ackermann_nav-ROS.* Retrieved from https://github.com/apletta/ackermann_nav-ROS

[19] Pletta, Alex and Sundaram, Karthikeyan (2019, June). *Turtlebot Navigation.* Retrieved from https://github.com/KarthikeyanS27/Turtlebot-Navigation

[20] Rösmann, C., Hoffmann, F., and Bertram, T. (2015, September). *Planning of multiple robot trajectories in distinctive topologies.* In 2015 European Conference on Mobile Robots (ECMR) (pp. 1-6). IEEE.

[21] Rösmann, Cristoph (2019, July). *teb_local_planner.* Retrieved from http://wiki.ros.org/teb_local_planner

[22] Santos, J. M., Portugal, D., and Rocha, R. P. (2013, October). *An evaluation of 2D SLAM techniques available in robot operating system.* In 2013 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR) (pp. 1-6). IEEE.

[23] Soltani, A. R., Tawfik, H., Goulermas, J. Y., and Fernando, T. (2002). *Path planning in construction sites: performance evaluation of the Dijkstra, A, and GA search algorithms.* Advanced engineering informatics, 16(4), 291-303.

[24] Thrun, S., Burgard, W., and Fox, D. (2005). *Probabilistic robotics* MIT press.

[25] Ulusoy, Melda. (n.d.). *Understanding Model Predictive Control, Part 1: Why Use MPC?.* Retrieved from https://www.mathworks.com/videos/understanding-model-predictive-control-part-1-why-use-mpc–1526484715269.html

[26] Ye, Y., He, L., and Zhang, Q. (2016). *Steering control strategies for a four-wheel-independent-steering bin managing robot.* IFAC-PapersOnLine, 49(16), 39-44.

[27] Zheng, K. (2016, September). *ROS Navigation Tuning Guide*

[28] (2008, April). *Ackermann steering geometry.* Retrieved from https://en.wikipedia.org/wiki/Ackermann_steering_geometry

[29] (2011, February). *How can I use the navigation stack on a carlike robot.* Retrieved from https://answers.ros.org/question/9059/how-can-i-use-the-navigation-stack-on-a-carlike-robot/

[30] (2014, March). *Why is ROS not real time?*. Retrieved from https://answers.ros.org/question/134551/why-is-ros-not-real-time/

[31] (2014, April). *Heap's algorithm*. Retrieved from https://en.wikipedia.org/wiki/Heap%27s_algorithm

[32] (2015, September). *Planner for Car-Like Model*. Retrieved from https://answers.ros.org/question/217091/planner-for-car-like-robot/

[33] (2017). *Ackermann Interest Group*. Retrieved from http://wiki.ros.org/Ackermann%20Group

[34] (n.d.). *Which combination of ROS/Gazebo versions to use*. Retrieved from http://gazebosim.org/tutorials/?tut=ros_wrapper_versions

[35] (n.d.). *Print all combinations of length k that can be formed from a set of n characters*. Retrieved from https://www.geeksforgeeks.org/print-all-combinations-of-given-length/

[36] (n.d.). *How to check if two given line segments intersect?*. Retrieved from https://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/