# Precision Navigation for Indoor Mobile Robots

by

ERIC MICHAEL PERKO

Submitted in partial fulfillment of the requirements
For the degree of Master of Science

Thesis Advisor: Dr. Wyatt S. Newman

Department of Electrical Engineering and Computer Science
CASE WESTERN RESERVE UNIVERSITY

January, 2013

# Table of Contents

# List of Tables

# List of Figures

# List of Algorithms

# Acknowledgements

First I would like to thank my thesis advisor, Dr. Wyatt S. Newman for his guidance and patience throughout this thesis. I would like to thank Chad Rockey for his help keeping HARLIE operational during this thesis. I would also like to thank Tony Yanick and Megan Gorrow for their help in collecting experimental data and videos. I must also thank Invacare Corporation for their generous support that enabled the research that went into this thesis.

Precision Navigation for Indoor Mobile Robots

Abstract

by

ERIC MICHAEL PERKO

This thesis describes a precision navigation system for indoor mobile robots. It includes a precision localization subsystem, based on a laser scanner, wheel encoders, gyroscope and *a priori* map, and a precision path execution system made up of a steering algorithm, a trajectory generator and a simplistic path planner. Geometric parameterizations for path segments were developed for use by those components. The precision navigation system was evaluated using a physical robot, CWRU's HARLIE, as well as in simulation. This precision navigation system allowed HARLIE to precisely navigate indoors, following paths with as little as just over three centimeters of lateral offset.

# 1 Introduction

Mobile robotics is a growing area in the field of robotics. The key difference between mobile robotics and other types of robotics, such as industrial automation, is that mobile robots can navigate their environment. While there have been mobile robots such as tour guides or self-driving cars for years now, new classes of mobile robots such as service robots or smart wheelchairs are pushing the limits of current navigation technologies.

While there are mature technologies for outdoor navigation, as is the case with field robotics or self-driving cars, as well as simply moving from point A to point B indoors while avoiding obstacles, as is the case with tour guide robots, there is relatively little available for doing precision navigation indoors or outdoors. Whereas traditional navigation approaches often include path followers that simply make a "best effort" to precisely follow the overall path plan while avoiding obstacles, a precision navigation system is designed to precisely follow the overall path plan at all times while still avoiding obstacles.

Precision navigation allows mobile robots to function in situations that are outside the capabilities of standard robot navigation techniques. For example, with precision guarantees, a smart wheelchair can smoothly and reliably pass through doorways with little margin for error or pull up parallel and close enough to a wall so that the user can press a handicap door assist button. Standard navigation techniques do not make strong guarantees about achieving goal positions via a precise path and it would be very difficult for many navigation systems to approach a wall so closely without making many attempts. Even though a precision navigation system must always precisely follow the path, it must also deal with dynamic obstacles that appear in the path. Because a precision navigation system does not allow the robot to intentionally deviate from the planned path, the planned path must be updated via dynamic replanning in order to account for and avoid any dynamic obstacles.

The precision navigation system described in this thesis is designed to do all of those things and such behaviors were experimentally confirmed both in simulation and with a physical robot, Case Western Reserve University's HARLIE. The remainder of this thesis is organized as follows. Section 2 describes the experimental systems used, HARLIE and the Gazebo simulation environment, to test the precision navigation system. Section 3 describes related navigation systems, specifically open source navigation systems that were evaluated on HARLIE. Section 4 describes the precision localization subsystem used by this precision navigation system to determine the robot's current pose. Section 5 describes the steering component of this precision navigation system, which outputs driving commands to the robot. Section 6 describes the trajectory generation component of this precision navigation system, which takes a planned path and outputs the next state the robot needs to achieve in order to follow that path. Section 7 describes the simplistic path planner developed for this thesis. Section 8 describes a number of experimental results for the different components of the precision navigation system. Section 9 describes possible avenues for extending the precision navigation system described in this thesis.


Figure 1: HARLIE, the robot used for experimental testing

# 2 Experimental System

Two separate experimental systems were used when testing and gathering results for localization and motion planning. The first of these is the physical robot, HARLIE, which will be described in further detail in Section 2.1. The second experimental system is a simulation environment which will be described in detail in Section 2.2.

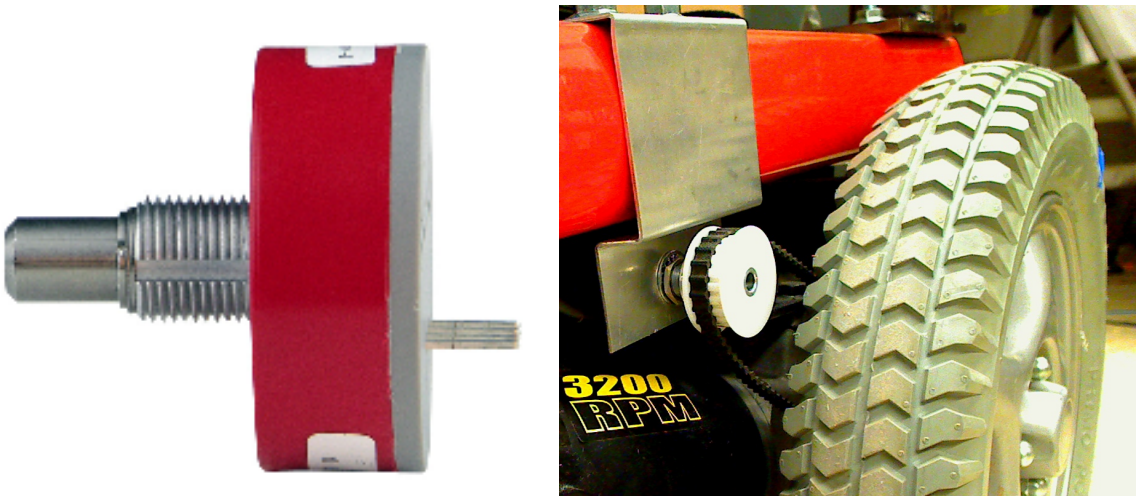
## 2.1 HARLIE

The physical robot used for these experiments was HARLIE (see Figure 1). HARLIE is a fully autonomous robot built on top of a wheelchair base donated by the Invacare Corporation. The robot is powered via a pair of car batteries in series, providing a nominal twenty-four volts to the system. The wheelchair base has two electric motors

powering the two drive wheels; with one motor per wheel, HARLIE is able to vary the velocity of each drive wheel independently. Because of this independent velocity control, HARLIE can move forwards and backwards, both straight and in arcs, as well as spin in place. This drive setup is known as "differential drive" and is one of the most common drive setups used in mobile robotics [17].

On top of this differential drive base, HARLIE is equipped with numerous sensors and computing systems. HARLIE is equipped with three sensors that are used for indoor localization: quadrature encoders, a MEMS gyroscope and a LIDAR. The quadrature encoders on HARLIE are used to measure the speed of each of HARLIE's drive wheels; these speeds are then integrated over time to get an estimate of the total distance moved by each wheel. HARLIE also has encoders attached to the two drive shafts, though these are used in the velocity control loops and not for localization. The encoders used on HARLIE (see Figure 2a) are high-resolution optical encoders, providing over 1000 encoder ticks per wheel revolution (or approximately 58 $\mu$m per tick), which allows for precise measurement of the wheel's speed and direction. The encoders are not directly connected to the wheel's axle - they are connected via a pair of sprockets and a rubber, toothed belt (see Figure 2b).

The second sensor on HARLIE used for indoor localization is an Analog Devices MEMS gyroscope (see Figure 3). This is a single-axis gyro used to measure the yaw rate (or angular velocity about the origin of rotation) of the robot. This sensor can measure a yaw rate of up to $\pm 2.6$ radians per second and includes both a temperature sensor output as well as reference voltage output [7]. These latter outputs are important for correcting the sensor bias automatically - without bias correction, the yaw rate reading from the gyro will drift and give erroneous readings. With bias correction (see Section 4.1 for how this correction is done on HARLIE), the gyro is a very accurate estimator of HARLIE's yaw rate, even when there is wheel slippage that can cause errors in a yaw rate estimate derived from differencing the velocities




(a) Grayhill 61K Optical Encoder (b) Wheel Encoder Setup

Figure 2: Encoder & Encoder setup

of each drive wheel as reported by the encoders.

The third sensor on HARLIE used for indoor localization is the SICK LIDAR (see Figure 4a). This LIDAR (LIght Detection And Ranging) unit uses a beam of infrared light (905nm wavelength) to scan a 180° arc in 1° increments in front of the sensor, returning a complete 180° scan at a rate of 75Hz (see Figure 4b). The LMS291 can detect objects out to a range of 80 meters with an accuracy of ±1 centimeter [2]. With this long range and excellent accuracy, the SICK is able to provide enough information to localize HARLIE precisely in an indoor environment (see Section 4.2 for details).

Harlie includes a number of other sensors, including cameras, a GPS receiver and sonar. Some like the GPS receiver, which allows for precise localization outdoors, would allow the methods described in the remainder of this thesis to function outdoors as well as indoors. Others such as the camera could augment the LIDAR for indoor localization [13]. Still others such as the sonar could be augmented and used to assist in detecting obstacles that the LIDAR cannot sense such as glass walls or materials with low infrared reflectivity.

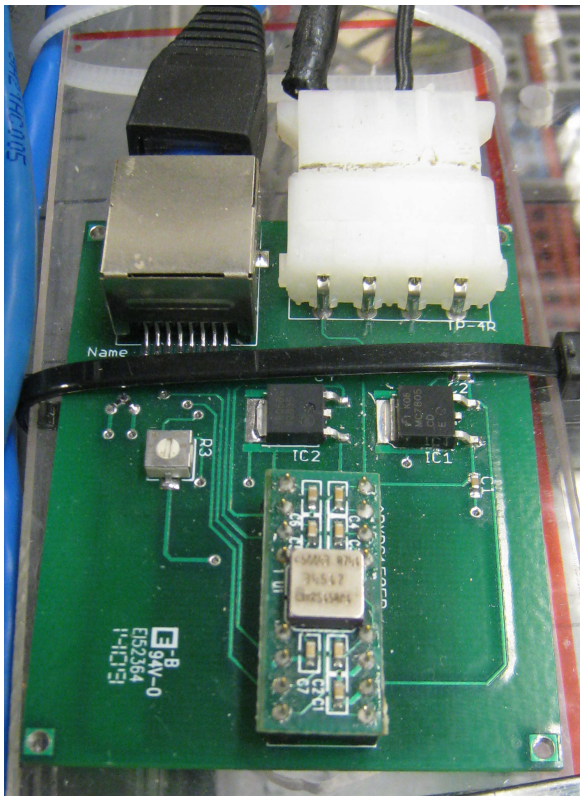
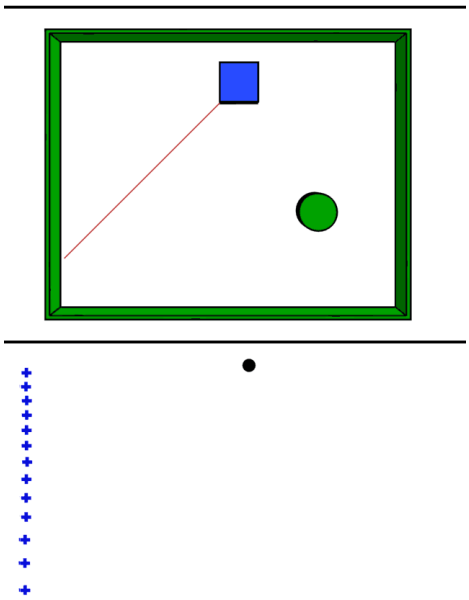



Figure 3: Analog Devices ADXRS150 Gyro as mounted on HARLIE

HARLIE also includes a pair of computers. The first of these is a National Instruments CompactRIO (cRIO) that consists of a 400MHz PowerPC running VxWorks (a hard real-time operating system), a Xilinx FPGA and a large number of input/output modules for connecting to the different sensors [24] (see Figure 5). HARLIE's cRIO is used for two main functions: relative localization (see Section 4.1 for details) and providing velocity control of the two drive wheels. The cRIO takes a translational (forwards/backwards) velocity as well as rotational velocity and converts them into a velocity for each drive wheel (see Section 4.1.2 for the equations used in this conversion). These individual wheel velocities are then used as the setpoint for the PID controller [25] for each wheel – by using the wheel encoders for feedback to these PID algorithms, velocity commands are executed promptly and accurately even on different types of surfaces. Accurate velocity control offloads some work that would otherwise have to be done in the higher-level motion planning subsystems to ensure


(a) SICK LMS291



(b) Sample LIDAR Scan [29]. Top: Sensor in blue, laser beam in red, environment in green. Bottom: Sample data points for a partially complete scan of the environment shown above (beam moves counterclockwise).

Figure 4: SICK LIDAR & Sample LIDAR Scan

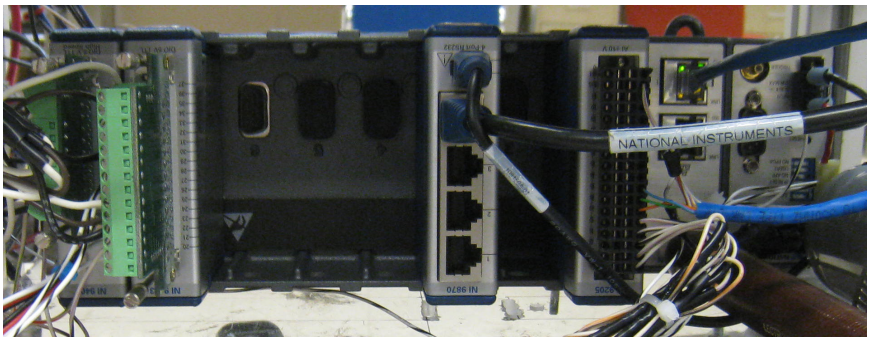



Figure 5: NI cRIO with 4 IO modules installed and connected to on-board sensors

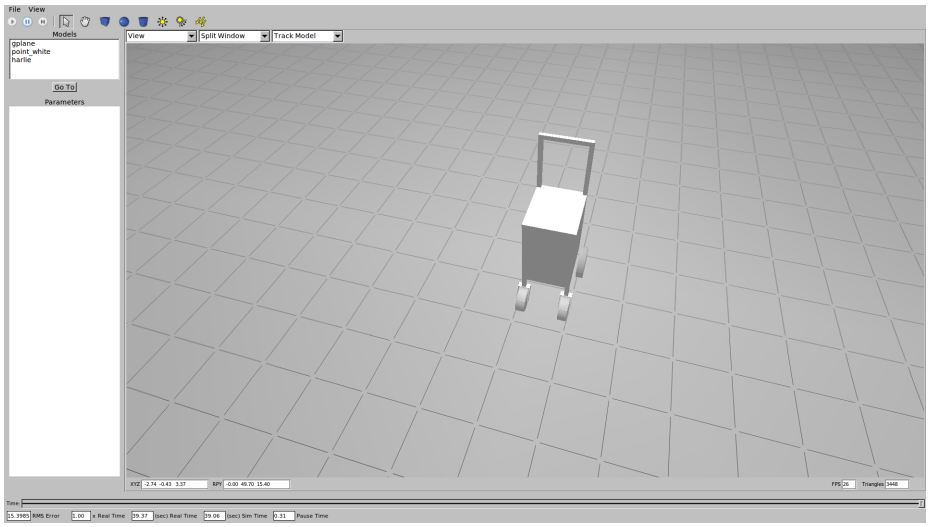



Figure 6: HARLIE in the Gazebo Simulator

that the robot base is executing commands properly. The cRIO receives velocity commands and sends localization and status information to HARLIE's main computer via UDP over an Ethernet network.

The second computer on HARLIE is a custom-built Linux computer that includes an Intel Core i7 CPU clocked at 2.66GHz with up to eight simultaneous threads of execution and six gigabytes of RAM. This computer runs all of the obstacle mapping, absolute localization and path execution algorithms discussed in later chapters. Since this computer is a standard Linux computer, many open-source libraries, programming languages and tools are available for use in those subsystems.

## 2.2 Simulation

While a physical platform is required for real world testing of a full navigation stack, field testing is impractical for gathering the amount of data required to make any sort of generalization about the performance characteristics of the algorithms involved. In order to gather such a large amount of data practically, a simulated robot was setup to mimic the characteristics of HARLIE. To do this, a number of steps were carried out.

First, a reasonably accurate model of HARLIE's physical structure and characteristics was created. This includes things such as the locations of the wheels and LIDAR, HARLIE's mass and estimates of the inertial constants of the wheels and the configuration and noise model for HARLIE's LIDAR (see Figure 6). Next, all of this information was fed into the Gazebo simulator [14], which then uses those characteristics to do a 3D full-physics simulation of HARLIE. This setup was used to validate changes to the algorithms before testing on HARLIE as well as for gathering the data used to analyze the steering algorithms' attraction regions.

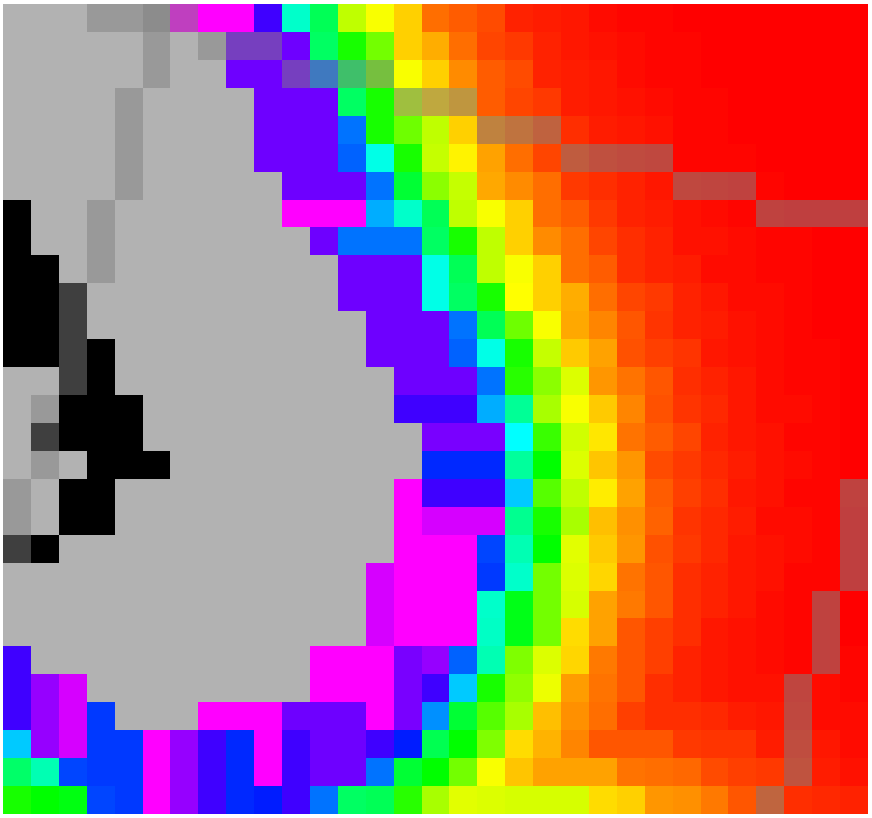


Figure 7: costmap_2d Sample

# 3 Related Work

Before delving into the methods and algorithms developed as part of this thesis, a survey of existing open source navigation packages is necessary. This will provide background for the design choices in this thesis, as deficiencies in these existing packages were areas that this thesis focused on improving upon. Specifically, the most mature and complete open source navigation package is that available in the Robot Operating System (ROS)[21]; due to this maturity and completeness, the ROS navigation stack will be the primary focus of this examination of related work. The ROS navigation stack is made up into three distinct parts that will be detailed in individual sections: obstacle mapping, local planning, and global planning.

## 3.1 Obstacle Mapping

For obstacle mapping, the ROS navigation stack uses a software package called "costmap_2d" [5]. This package takes in sensor information about the environment, builds a 2D or 3D fixed-resolution occupancy grid by raytracing that sensor information and inflates obstacles to facilitate navigation based on user provided robot parameters. For a visualization of a small portion of a costmap built from actual sensor data, see Figure 7. In Figure 7 black pixels are the actual sensed obstacles, gray pixels are marked as "lethal" cost and the rest are a gradient between high cost in violet and low cost in red. Lethal cells are obstacles that have been inflated according to the robot's geometry - specifically these are any cells that if the control point (or origin/center) of the robot were to enter one of these cells it would be in collision with the actual sensed obstacle. By inflating obstacles based on the robot's geometry, planning and control can approximate the robot as a single point as opposed to working with its true geometry [27]. Outside of this inflated radius, the cost gradient expresses a preference for the path planners to stay away from obstacles unless other factors force the robot near to the obstacle.

While this approach works in many environments, it has drawbacks when used for precision navigation. The most serious drawback is the fixed resolution of the grid when very fine resolution is required for some areas, but not others. For example, imagine navigating through a narrow doorway that opens to wide open areas on either side. In this situation, there is generally no good overall choice for the resolution of the grid. Choosing a resolution that is fine enough to ensure that the robot can always fit through the doorway leads to an unnecessarily large number of empty cells in the wide open areas, wasting both memory and computation time evaluating these cells; choosing a resolution that reduces the number of cells in the open areas may not guarantee that the doorway will always be sensed as traversable, as even a sensor reading along an edge of a cell will cause the entire cell to be marked as an obstacle.

While prior knowledge about an environment allows one to choose a good resolution for that environment, that choice may not be optimal for a different environment; even if an "adaptive" fixed resolution were used that was always optimal for representing a given environment, path planning performance, both how long it takes to find an optimal plan and the actual optimal plan generated, can be affected by changing the obstacle map resolution, which is undesirable for a navigation system that seeks to perform consistently in different environments.

Another issue with the implementation used by costmap_2d is that, with a very fine grid cell resolution, obstacles can leave "droppings" in the map. These "droppings" occur when an obstacle such as a human moves across the sensor's field of view and is not completely cleared out of the map because the raytraced sensor beams do not intersect all of the cells that had already been marked as occupied. In order to resolve this problem, either a large enough grid cell resolution must be used to ensure that sensor beams will never fall to either side of an occupied cell (but, as discussed previously, large grid cell resolutions have other significant drawbacks) or an occupancy grid implementation that can clear cells even if a sensor beam does not raytrace through a particular cell must be used. For example, an occupancy grid implementation could slowly "fade" obstacle cells so that obstacles that have not been observed for some time would automatically be cleared. Other possibilities include explicitly modeling the probability of each cell given all of the previous sensor measurements, such as the occupancy grid mapping techniques introduced in [22]. The proposed mapping solution used in this thesis is detailed in Section 6.4.

## 3.2 Local Planning

For local planning, the ROS navigation stack uses a software package called "base_local_planner" [4]. This package uses the occupancy grid built by the package described in Section 3.1 to generate steering commands (translational velocity and rotational velocity) such

that the robot follows a given path through the environment without collisions. To do this, base_local_planner, as configured for HARLIE, uses an algorithm called Trajectory Rollout [11]. The algorithm can be described as a sequence of four steps that are repeated until the robot reaches the end of the path. The steps are as follows:

1. Sample the robot's control space, generate translational (forward) velocity and rotational velocity pairs. These pairs are the possible commands that could be sent to the robot, given it's current state and dynamics constraints (e.g. acceleration).

2. Forward simulate each of those control pairs for some short, fixed amount of time to determine what would happen if the robot were given that control pair. One forward simulated control pair creates one possible trajectory.

3. Score each trajectory using a cost function (3.1) to determine the "goodness" of a trajectory. Discard any trajectories that result in collisions.

4. Pick the trajectory that scores the best (this could be lowest or highest score depending on cost function) and send the control pair that generated that trajectory to the robot.

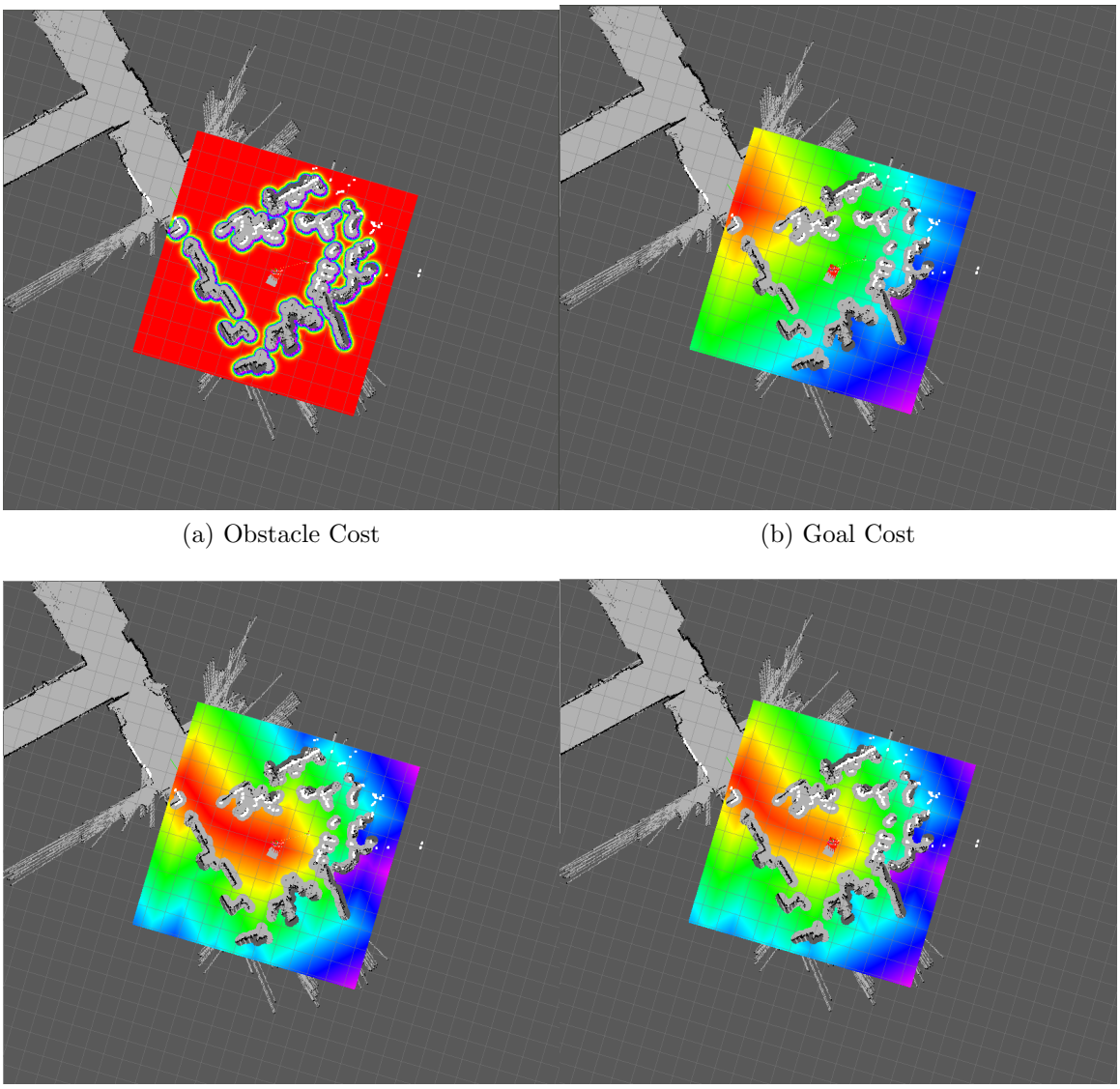
The scoring function used by the Trajectory Rollout implementation in base_local_planner is as follows:

$$k_{path} \cdot pathDist + k_{goal} \cdot goalDist + k_{obs} \cdot obs \tag{3.1}$$

In (3.1), the three $k$ terms are weights that control how much each of the individual cost components influence the overall score. The three cost components are $pathDist$, $goalDist$ and $obs$. $pathDist$ is the distance between the endpoint of the trajectory and the closest point on the path. $goalDist$ is the distance between the endpoint of the trajectory and the end of the given path. $obs$ is the maximum cost for any point

along the trajectory as returned by the occupancy grid. A visualization of the cost function components and sum for one cycle is in Figure 8.

There are a number of issues that arise from using this type of method when attempting to do precision navigation on a differential drive robot. One such issue is the simplistic generation of trajectories. In base_local_planner, trajectories are generated using a fixed number of samples between the minimum and maximum translational velocities that could be commanded to the robot, given the current translational velocity and acceleration and deceleration limits, and similarly sampling from the possible rotational velocity space. The trajectories are then generated by forward simulation of the Cartesian product of these two sets of velocities (see Figure 9 for an example set of trajectories). The problem with this method of generating trajectories is two-fold. First, the number of samples must remain relatively small compared to the overall possible controls space in order to forward simulate and score all trajectories quickly (commands should be sent to the robot at 20Hz, so a control pair must be chosen at least every 20Hz), so a trajectory that may actually be better in the long term may not be evaluated if it was not one of the samples. Second, because one control pair is chosen per forward simulation, only short, constant curvature trajectories are evaluated. While these two problems do not generally affect navigation in environments where the circumscribed radius of the robot can pass through the smallest gap it must navigate (i.e. for a rectangular robot it could fit through sideways), they quickly lead to undesirable behavior in situations such as robotic wheelchairs passing through ADA compliant doorways or any other case where the robot must precisely follow the given path. In such situations, these simplistic trajectories do not ensure that the robot will be able to pass through the doorway smoothly — for example, base_local_planner often needs to back up and re-align the robot once it gets close to the doorway because the trajectories that were scored and chosen did not lead to the robot being lined up such that it could pass straight



(a) Obstacle Cost

(b) Goal Cost

(c) Path Cost

(d) Total Cost

Figure 8: Sample Trajectory Rollout Cost Visualization. The robot is in the center of the cost field. The end of the commanded path is visible as a green line to the upper left of the cost field, heading towards the open doorway. Red is low cost. Violet is high cost.

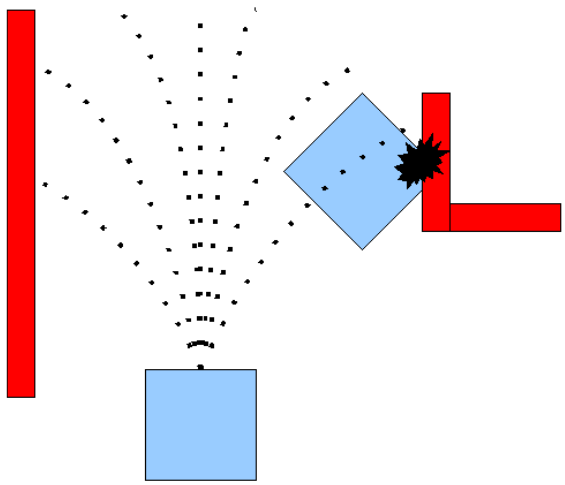


Figure 9: base_local_planner Sample Trajectories [4]

through the doorway.

Another issue with using Trajectory Rollout for precision navigation is the simplistic scoring function used. The specific cost function used by base_local_planner, (3.1), has only three components, which should cause the planner to choose trajectories that remain far away from obstacles, end near the goal and remain near the path. In practice, these few components are insufficient for precision navigation. For example, it is entirely possible for the planner to choose a trajectory that avoids obstacles and gets close to the goal, but doesn't follow the given path very closely even though it has the best score. This comes about because the distance to the path is only calculated from the endpoint of the trajectory and therefore is not some measure of overall fit of the entire trajectory to the given path. Another issue with this scoring function is its inability to handle spin-in-place actions. For a differential drive robot, these are an important ability when attempting to navigate in constrained environments (for example, a spin-in-place could be used for realigning to a doorway) but, because this scoring function prefers trajectories that move towards the goal and/or path, spin-in-place trajectories are rarely chosen, even if a spin-in-place would actually be a more optimal choice for getting to the goal sooner. Spin-in-place actions are not as important for a holonomic robot, as it can often just strafe instead of realigning or

while realigning; the preference of the cost function towards trajectories that move the robot closer to the goal and/or path still works when strafing, since strafing is more than just a change in heading.

## 3.3 Global Planning

For global planning, the ROS navigation stack uses a package called "navfn" [23]. This package uses the occupancy grid built by the package described in Section 3.1 to generate the minimum cost plan between the robot's starting position and a given goal location. To do this, navfn uses Dijkstra's shortest path algorithm as described in [17], assuming the robot has a circular projection onto the costmap. While this method does generate optimal paths that avoid obstacles, it has some shortcomings when used for precision navigation on robots such as HARLIE.

The most serious of these shortcomings is the circular robot assumption. For a robot such as HARLIE, which has an approximately 3:2 length:width footprint for the base, there are two choices for the radius of this circular robot approximation: the inscribed radius or the circumscribed radius. If the inscribed radius is chosen, navfn is planning paths between obstacles that the robot can actually pass through, but the planner may generate paths that cause the back end to hit obstacles. An example of this situation is when planning an L-shape through a narrow doorway – after the path passes through the midpoint of the doorway, it will turn too early and, if the path is followed precisely, cause the back of the robot to swing into the door as it turns. On the other hand, if the circumscribed radius is chosen navfn will not be able to plan a path through all environments that the robot can actually navigate because the planner will, essentially, be trying to fit the robot through sideways. Any global planner used for precision navigation must use the actual footprint of the robot so that it plans paths in any environment the robot can safely navigate while not underestimating the size of the robot and generating paths that may lead

to collisions.

Another issue with navfn as the global planner in a precision navigation application is that it is not a dynamic planning algorithm. Dijkstra's algorithm (and other static path planning algorithms such as A*) must replan from scratch if the environment changes during execution; to achieve precise navigation in arbitrary environments, any planner used must handle dynamic obstacles such as people or other robots. Dynamic planning algorithms such as Field D* [8] or ADA* [19, 20] are designed to adjust the planned path to account for dynamic obstacles with a minimum amount of recomputation. Because of this, the plan can continuously be refined or a larger, more flexible state space used, leading to smoother, better overall plans; Dijkstra's would need a relatively small state space in order to recompute the entire path quickly to account for dynamic obstacles.

# 4 Localization

One of the major components of any navigation system is the localization subsystem. The job of the localization subsystem on any robot is to determine where the robot is. Knowing where the robot is is essential to being able to actually navigate to a specific goal position, otherwise the robot won't know once it reaches that goal position. The pose estimates generated by the localization subsystem may also be used in control loops or otherwise by the planning and control subsystems. For example, the local planner described in Section 3.2 uses the pose (2D position and heading) as well as the translational and rotational velocities of the base when generating possible trajectories to evaluate. The precision navigation system described in this thesis uses the pose and velocity estimates as control variables to generate commands that follow the desired path (see Section 5).

For precise navigation, the robot not only needs to know where it is, but that pose estimate must be at least as accurate as the precision navigation system as a whole is designed to be. For example, a robot that must pass through a doorway with less than seven centimeters of clearance on either side must know where it is at any given time to at least seven centimeters or it is unlikely to reliably (and smoothly) get through the doorway. One important consideration is that the localization for precise navigation does not need to be accurate, only precise. As long as the coordinates for a given real world position are estimated to be the same over multiple runs, it is unimportant if those estimates do not match something like a floorplan. The localization subsystem used by the precision navigation system described in this thesis is broken up into two separate components, depending on what type of reference frame their pose estimates are in: the relative localization component and the absolute localization component.

## 4.1 Relative Localization

The first major component of the localization subsystem used in this thesis is the relative frame localization system. This component generates pose estimates relative to wherever the robot was powered on. The relative localization subsystem also generates estimates of the translation and rotational velocities. While these estimates are important, the relative frame position estimate is only good for uses that can tolerate a reference frame that drifts over long periods of time and is not fixed between times when the robot is powered on and off. For these reasons, it is not useful for describing goal points or global planning. It is useful for local planning and collision avoidance, as both of those are tolerant to drift in the reference frame. The estimates generated by the relative frame localization are also useful for the control algorithms used in this thesis because they can be generated at a high rate thanks to the computational simplicity of the algorithms used compared to absolute frame localization algorithms as described in Section 4.2.

HARLIE's relative frame localization is generated by using an Extended Kalman Filter (EKF) [16, 32, 31]. The EKF is an extension of the standard Kalman Filter algorithm, used for optimal state estimation in linear systems, to non-linear systems such as a differential drive robot. Using a model of the dynamic system and measurements from a variety of sensors, the EKF is able to produce a state estimate that is more accurate than any sensor individually. The EKF on HARLIE uses two of the sensors described in Section 2.1 to produce these relative frame state estimates: the encoders and the gyroscope. Algorithm 1 is a description of the exact EKF algorithm used on HARLIE. The values of each set of matrices and functions used are detailed afterwards.

The inputs to Algorithm 1 are the previous state estimate ($x_{t-1}$), the covariance of the previous state estimate ($P_{t-1}$) and a set of new measurements ($z_t$). The filter outputs an estimate of the current state ($x_t$) and the covariance of the current

**Algorithm 1:** HARLIE's Extended Kalman Filter

**Input**: $x_{t-1}$, $P_{t-1}$, $z_t$
**Output**: $x_t$, $P_t$

1 $\hat{x_t} = f(x_{t-1})$
2 $\hat{P}_t = G \cdot P_{t-1} \cdot G^T + Q$
3 $\hat{P}_t = \texttt{ZeroOutBiasXYThetaCov}(\hat{P}_t)$
4 **foreach** $z_t^i \in z_t$ **do**
5 $\quad y_t^i = z_t^i - h(\hat{x_t})$
6 $\quad S = H \cdot \hat{P}_t \cdot H^T + R$
7 $\quad K = \hat{P}_t \cdot H^T \cdot S^{-1}$
8 $\quad \hat{x_t} = \hat{x_t} + K \cdot y_t^i$
9 $\quad \hat{P}_t = (I - K \cdot H) \cdot \hat{P}_t$
10 $x_t = \hat{x_t}$
11 $P_t = \hat{P}_t$

state ($P_t$). In Algorithm 1, lines 1-3 are known as the prediction step and depend only on previous state and a model of the system. Lines 4-10 are the measurement updates where the EKF uses sensory information to refine the state estimate. The matrices used in the measurement update step are specific to which sensor generated the $z_t^i$ information being used, since different sensors generate different types of measurements. The following sections detail the prediction and two different sets of measurement updates that HARLIE's EKF uses.

### 4.1.1 Prediction Step

The prediction step is run every filter update, whether there are new measurements or not. This generates state estimates at a predictable, high fixed rate (50 Hz). The filter runs at this high fixed rate so that the navigation control algorithms always have fresh state information for their control loops.

$$x_{t-1} = \begin{bmatrix} x & y & \theta & v & \omega & b_\omega \end{bmatrix}^T \tag{4.1}$$

(4.1) is the state estimate used and generated by the filter. $x$ and $y$ are the

Cartesian coordinates of the robot on the ground plane, relative to where the robot was powered on. $\theta$ is the robot's heading relative to its heading when it was powered on. $v$ is the translational velocity of the robot and $\omega$ is the rotational velocity (yaw rate) of the robot. $b_\omega$ is a bias term to represent the drift present in the gyroscope's rotational velocity estimates.

$$P_{t-1} = \begin{bmatrix} \sigma_x^2 & \sigma_x\sigma_y & \dots & \sigma_x\sigma_{b_\omega} \\ \sigma_y\sigma_x & \sigma_y^2 & \dots & \sigma_y\sigma_{b_\omega} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{b_\omega}\sigma_x & \sigma_{b_\omega}\sigma_y & \dots & \sigma_{b_\omega}^2 \end{bmatrix} \tag{4.2}$$

(4.2) is the covariance of the state estimate. It is an $n \times n$ matrix, where $n$ is the number of elements in the state vector and arranged such that element $m, n$ of the covariance matrix is the covariance of the $m$-th element of the state vector with the $n$-th element of the state vector.

$$f\left(x_{t-1}\right) = \begin{bmatrix} x + v \cdot dt \cdot \cos\left(\theta\right) \\ y + v \cdot dt \cdot \sin\left(\theta\right) \\ \theta + \omega \cdot dt \\ v \\ \omega \\ b_\omega \end{bmatrix} \tag{4.3}$$

(4.3) is the process model used in line 1 of Algorithm 1. This function generates a prediction of the state at time $t$ given an estimate of the state at time $t-1$, where $dt$ is the time between filter updates (50 Hz or 0.02 seconds in this case). This particular process model assumes that the robot maintains constant translational and rotational velocities, as well as a constant bias term; since these terms are not truly constant, the process noise must be large enough to include the changes in

these values between filter update cycles. The model for the change in location and heading assumed that any rotations are pivot turns at the end of the motion; as such, it doesn't model constant curvature arcs. While there are other odometry models that do address constant curvature arcs (such as those in [31]), the model in (4.3) is sufficiently accurate when run at the 50 Hz rate of the filter and computationally simpler than models that take into account constant curvature arcs. If the filter were slowed down, a more complex odometry model might become necessary to maintain accuracy.

$$G = \begin{bmatrix} 1 & 0 & -v \cdot dt \cdot \sin(\theta) & dt \cdot \cos(\theta) & 0 & 0 \\ 0 & 1 & v \cdot dt \cdot \cos(\theta) & dt \cdot \sin(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 & dt & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.4}$$

(4.4) is the Jacobian of the process model (4.3) with respect to the state vector (4.1).

$$Q = \begin{bmatrix} \sigma_x^2 & 0 & 0 & 0 & 0 & 0 \\ 0 & \sigma_y^2 & 0 & 0 & 0 & 0 \\ 0 & 0 & \sigma_\theta^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & \sigma_v^2 & 0 & 0 \\ 0 & 0 & 0 & 0 & \sigma_\omega^2 & 0 \\ 0 & 0 & 0 & 0 & 0 & \sigma_{b_\omega}^2 \end{bmatrix} \tag{4.5}$$

(4.5) is the noise in the process model. It represents the uncertainty of the model and must account for any unmodeled characteristics of the system (such as that the velocity is not constant). For HARLIE, $\sigma_x = 0.001$, $\sigma_y == 0.001$, $\sigma_\theta = 0.001$,

$\sigma_v = \sqrt{10}$, $\sigma_\omega = \sqrt{10}$ and $\sigma_{b_\omega} = 1.0 \times 10^{-5}$.

$$\texttt{ZeroOutBiasXYThetaCov}\,(P_t) = \begin{bmatrix} \sigma_x^2 & \dots & \dots & \dots & \sigma_x\sigma_\omega & 0 \\ \vdots & \ddots & \dots & \dots & \sigma_y\sigma_\omega & 0 \\ \vdots & \vdots & \ddots & \dots & \sigma_\theta\sigma_\omega & 0 \\ \vdots & \vdots & \vdots & \ddots & \dots & \sigma_v\sigma_{b_\omega} \\ \sigma_\omega\sigma_x & \sigma_\omega\sigma_y & \sigma_\omega\sigma_\theta & \vdots & \ddots & \sigma_\omega\sigma_{b_\omega} \\ 0 & 0 & 0 & \sigma_{b_\omega}\sigma_v & \sigma_{b_\omega}\sigma_\omega & \sigma_{b_\omega}^2 \end{bmatrix} \tag{4.6}$$

(4.6) is the function applied to the prediction covariance in line 3 of Algorithm 1. This function zeros out the covariance between the yaw rate bias $b_\omega$ and $x$, $y$ and $\theta$. This is necessary to prevent the $x$, $y$ or $\theta$ terms of the prediction from adjusting the bias – which is a reasonable assumption, as the bias term cannot physically be influenced by $x$, $y$ or $\theta$. Without applying this function, bias estimation would become unstable quickly due to the influence of those variables.

### 4.1.2 Encoder Measurement Update

The encoder measurement update runs whenever there are new encoder readings. On HARLIE, the encoders are sampled once every filter update, so there is always a new encoder reading after completing the prediction step.

$$z_t^i = \begin{bmatrix} d_{left} \\ d_{right} \end{bmatrix} \tag{4.7}$$

(4.7) is the form of the measurement returned by measuring the encoders. $d_{left}$ and $d_{right}$ are the change in the left and right wheel encoders, respectively.

$$h\left(\hat{x}_t\right) = \begin{bmatrix} \frac{1}{2} \cdot dt \cdot (2 \cdot v + b \cdot \omega) \\ \frac{1}{2} \cdot dt \cdot (2 \cdot v - b \cdot \omega) \end{bmatrix} \tag{4.8}$$

(4.8) is the function used in line 5 of Algorithm 1 that generates a prediction of the measurement, given the estimated state $\hat{x}_t$. (4.8) is only used when $z_t^i$ comes from the encoders. $dt$ is again 0.02 seconds (50 Hz), since the encoders are sampled every filter update. $b$ is the track width of the base.

$$v = \frac{d_{right} + d_{left}}{2} \tag{4.9}$$

$$\omega = \frac{d_{right} - d_{left}}{b} \tag{4.10}$$

(4.9) and (4.10) are the odometry equations that (4.8) is based on. Using these two equations, the translational and rotational velocities can be related to the change in the individual encoders. $b$, the track width, is the distance along the axle between the two wheels.

$$H = \begin{bmatrix} 0 & 0 & 0 & dt & \frac{1}{2} \cdot b \cdot dt & 0 \\ 0 & 0 & 0 & dt & -\frac{1}{2} \cdot b \cdot dt & 0 \end{bmatrix} \tag{4.11}$$

(4.11) is the Jacobian of (4.8) with respect to (4.1).

$$R = \begin{bmatrix} d_{left}^2 \cdot \alpha_{left} + \epsilon & 0 \\ 0 & d_{right}^2 \cdot \alpha_{right} + \epsilon \end{bmatrix} \tag{4.12}$$

(4.12) is the noise in the encoder measurement. The noise model used has two components, one constant and one variable. $\epsilon$ is a small constant value to represent an error of $\pm 1$ encoder tick, so that even when there was no change in the encoders, the measurement noise accounts for possible mismeasurement. $\alpha_{right}$ and $\alpha_{left}$ are

parameters used to control the amount of variable noise applied to the measurement. This models the fact that the encoders become more unreliable when the robot is moving more quickly (or the wheels are slipping). For HARLIE, $\epsilon = 1.0 \times 10^{-8}$, $\alpha_{right} = 0.0002$ and $\alpha_{left} = 0.0002$.

### 4.1.3 Gyroscope Measurement Update

The gyroscope measurement update runs whenever there is a new gyroscope reading. On HARLIE, the gyroscope is sampled once every filter update, so there is always a new reading after completing the prediction step.

$$z_t^i = \begin{bmatrix} \omega_{sample} \end{bmatrix} \tag{4.13}$$

(4.13) is the form of the measurement returned by the gyroscope sampling component.

$$h\left(\hat{x}_t\right) = \begin{bmatrix} \omega + b_\omega \end{bmatrix} \tag{4.14}$$

(4.14) is the function used in line 5 of Algorithm 1 that generates a prediction of the gyroscope measurement, given the estimated state $\hat{x}_t$. (4.14) is only used when $z_t^i$ comes from the gyroscope.

$$H = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \tag{4.15}$$

(4.15) is the Jacobian of (4.14) with respect to (4.1).

$$R = \begin{bmatrix} \omega_{sample}^2 \cdot \alpha_{gyro} + \gamma \end{bmatrix} \tag{4.16}$$

(4.16) is the noise in the gyroscope measurement. Analogously to the encoder noise, this noise model also has a variable ($\alpha_{gyro}$) and constant ($\gamma$) component. For

HARLIE, $\alpha_{gyro} = 0.000048345$ and $\gamma = 0.00000019$.

## 4.2 Absolute Localization

The second major component of the localization subsystem used in this thesis is the absolute frame localization system. This component generates pose estimates relative to some fixed reference pose, such as the origin of a map or, for outdoor robots, GPS coordinates. As this thesis focused on indoor robots, the absolute localization system generates pose estimates relative to the origin of the map the robot is currently operating in. Unlike the relative localization system described in Section 4.1, the origin of the map (and therefore the pose estimates) does not drift over time or whenever the robot is powered on and off. Because the absolute reference frame does not drift, it is suitable for describing things like goal points or long-term paths. While the absolute reference frame position estimates don't drift, they are not suitable for things that need position updates at high rates due to the computational complexity of the algorithms involved.

HARLIE's absolute frame localization is generated by an algorithm called Adaptive Monte Carlo Localization (AMCL) [6, 10, 31]. AMCL is a type of probabilistic localization algorithm, based on a particle filter, that adaptively varies the number of particles used to represent the posterior. Each particle represents an estimate of the robot's pose relative to the fixed origin of the *a priori* map. These estimates are updated each cycle based on the pose and velocity estimates from the relative localizations algorithm described in Section 4.1, the laser data from the SICK LMS291 and the *a priori* map. The output of AMCL is a set of possible poses (each particle) with an associated weight; since weights are proportional to the likelihood of each pose being the true pose, the best estimate of the robot's current absolute position is simply the particle with the highest likelihood. The *a priori* map required by AMCL can be generated in a number of ways, such as the output of a SLAM algorithm like

GMapping [12] or the building's blueprints [9]. This thesis relied on a map generated by the ROS GMapping package with a five centimeter resolution for use while generating precision localization estimates.

The AMCL algorithm used on HARLIE was provided by the ROS "amcl" package [3]. The amcl package has a large number of parameters that control it's behavior. As configured, it used the algorithms *KLD_Sampling_MCL*, *sample_motion_model_odometry* and *likelihood_field_range_finder_model* as described in [31]. *KLD_Sampling_MCL* implements the adaptive particle filter localization described previously. *sample_motion_model_odometr* draws samples from the distribution given by sampling based on the change in pose and adding those samples to the last computed pose. *likelihood_field_range_finder_model* precomputes the likelihood of a laser "hit" for each cell in the map - at runtime, getting the probability of a given laser range reading is as simple as looking up the 2D coordinates of the hit location in the precomputed likelihood field, which makes it much faster than alternative methods based on raytracing each beam through the entire map. This likelihood field model can be slightly less accurate though, for example if the beam should have intersected with a known obstacle prior to the hit on another known obstacle, the likelihood field model will only return the probability of the hit on the second known obstacle, not the first. However, in our indoor environment, these situations are not very likely to occur and so the speed advantages of the likelihood field model outweigh the accuracy advantages of the raytracing based model.

As mentioned previously, the ROS amcl package has a large number of parameters available. All but six of these parameters were left at their default values. Four odometry motion model related parameters were changed: $odom\_alpha1 = 1.2$, $odom\_alpha2 = 1.2$, $odom\_alpha3 = 1.8$, $odom\_alpha4 = 1.2$. These increased the expected noise levels in the odometry estimates from the relative localization algorithm from their default values of 0.2 to values that were more accurate for the actual noise

in those estimates. Essentially, this allowed the AMCL algorithm to rely more on the laser than on the odometry estimates; because HARLIE operated indoors with a high quality laser and high resolution *a priori* map, this led to more precise absolute pose estimates than the default parameters. Two parameters that control how often AMCL recomputes the absolute pose estimate were also modified: $update\_min\_d = 0.05$ from 0.2 meters and $update\_min\_a = 0.1$ from $\pi/6$ radians. Modifying these parameters caused AMCL to recompute the position estimate every five centimeters or 0.1 radians; recomputing the pose more often while moving (the decrease in $update\_min\_d$) led to smoother absolute pose estimates which prevented any large jumps in absolute position in between trajectory generation loops.

# 5 Steering

Another major component of a navigation subsystem is the path execution component. There are a number of parts that make up a path execution component; in this thesis, there are three such components: steering, trajectory generation and path planning. The lowest-level component is steering. Steering's task is to take a desired robot state (this represents the ideal state of the robot at a given time) and the current robot state and, from those two inputs, produce a set of commands that will make the robot's current state converge on the desired state. The state of the robot is not simply a position and orientation, but also includes parameters such as translational and angular velocities and curvature.

## 5.1 State Description

In the precision steering algorithms developed for this thesis, the final version of the state description was made up of the fields described in Table 1.

| Name | Description |
|---:|---|
| header | This is a standard ROS header type that contains information such as the state's timestamp as well as what reference frame the desired pose is in. |
| segment_type | An integer enum representing the type of segment that generated this desired state, such as a straight line segment, constant curvature arc segment or spin-in-place segment. |
| segment_number | The ID number of the segment that generated this state. |
| pose | A six degree of freedom pose, containing x, y and z coordinates as well as roll, pitch and yaw angles. |
| velocity | The desired velocity for this state. Whether the velocity is in meters per second or radians per second depends on the *segment_type*. |
| rho | The desired curvature for this state. Not used for spin-in-place segments. |
| segment_distance | How far along the path segment this state corresponds to. |

Table 1: Steering Desired State Field Description

Of the fields listed in Table 1, most of the fields listed are obvious choices for

inclusion in the state. For example, if part of the steering algorithm's task is to move the robot to a certain position and orientation in the environment, then obviously the steering algorithm needs that pose in order to move the robot to it. Other fields in this parameterization of the desired state are mainly for informational purposes or consistency with the structures used by the trajectory generator, such as the segment number or segment distance, and were not used by the steering algorithms themselves. One important choice is the parameterization of the segment type field used in this final version.

Originally, only two segment types were used: line segments and constant curvature arc segments. For the steering algorithms, there was no difference in interpretation of those two segment types. While these worked well for most path segments, representation of spin-in-place segments was somewhat hacky, simply representing them as arcs with very small radii (e.g. less than 1 $\mu m$). Such a representation was acceptable at a theoretical level, as arcs with very small radii are reasonable approximations of a spin-in-place, but led to numerical stability issues when using such small numbers in the steering algorithms due to floating point math behavior on modern computers. In order to solve those numerical stability issues, a third segment type was introduced: the spin-in-place segment. These segments were assumed to have no translational offset and changed the interpretation of the desired velocity from a translational velocity to a rotational velocity.

## 5.2 Steering Algorithms

Two different steering algorithms were developed and evaluated for use in this precision navigation system. The first of these is an algorithm called *second-order steering* (see Section 5.2.1) and the second, developed to address deficiencies in the first, is called *phase space steering* (see Section 5.2.2). Both of these algorithms have the same form, as laid out in Algorithm 2.

**Algorithm 2:** Generic Steering Algorithm

**Input**: $\hat{x}_t$, $x_t$
**Output**: $v$, $\omega$
1 $v$, $\omega$ = executeSteeringAlgorithm($\hat{x}_t$, $x_t$)

The inputs to Algorithm 2 are the desired robot state ($\hat{x}_t$) and the current robot state ($x_t$) as estimated by the precision localization system. The steering algorithm outputs are the translational velocity ($v$) and rotational velocity ($\omega$) that, when commanded to the robot base, are the best commands to ensure convergence to the desired state. Since these are only individual commands, not a sequence of them, the steering algorithm must be run at a regular rate, recomputing the commands each time, to ensure convergence. In this thesis, the steering algorithms were tuned to run at 20 Hz.

Both steering algorithms use the same equation to determine the output velocity command, $v$. (5.1) gives this equation.

$$v = v_{des} + k_v \cdot Lfollow \tag{5.1}$$

In (5.1), $v$ is the output desired velocity, $v_{des}$ is the desired velocity according to the input desired robot state and $k_v$ is a tunable gain constant. $Lfollow$ is the error between the robot's state and the desired state along the desired path. $Lfollow$ represents how far in front or behind the robot is from the desired state and, accordingly, modifies the desired velocity to correct this error along the desired path. For example, if the robot has fallen "behind" the desired state, the output velocity is increased such that it can reduce the tangential error (and hence $Lfollow$ term). See Section 5.3 for more discussion of the $Lfollow$ term.

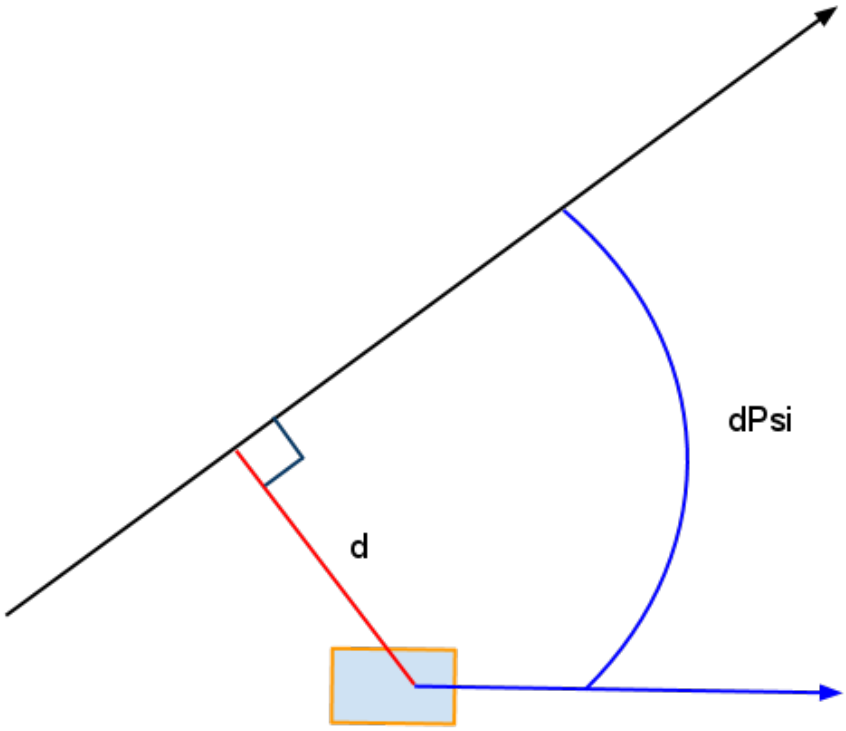



Figure 10: Steering Variable Diagram

### 5.2.1 Second-Order Steering

The initial steering algorithm used in this thesis is called *second-order steering*. This algorithm implements (5.1) and (5.2).

$$\omega_{cmd} = v \cdot (k_d \cdot d + k_\Psi \cdot \delta\Psi + \rho_{des}) \tag{5.2}$$

In (5.2), $\omega_{cmd}$ is the output rotational velocity, $v$ is the output of (5.1), $\rho_{des}$ is the curvature from the desired state and $k_d$ and $k_\Psi$ are tunable gains. $\delta\Psi$ is the difference between the orientation of the desired state and the robot's current state (constrained to the interval $[-\pi, \pi]$). $d$ is the perpendicular offset of the robot's position from the desired state and represents how far "off" the desired path the robot is. Note that $d$ is calculated relative to the desired state extended along its tangent vector in both directions and not necessarily just the desired state's *pose* member described in Table 1.

Figure 10 is a graphical representation of $d$ and $\delta\Psi$ for a sample robot state and desired state. The black arrow in Figure 10 is the desired state extended along its tangent vector in both directions, the rectangle with orange outline is the robot's current footprint, the blue arrow represents the robot's current position (base of the arrow) and heading (direction of the arrow). $d$ in Figure 10 is the $d$ term described above and used in (5.2). $dPsi$ in Figure 10 is the $\delta\Psi$ term described above and used in (5.2).

While *second-order steering* is a straightforward control algorithm both theoretically and in implementation, it does have some significant drawbacks for precision navigation on an indoor mobile robot. The most significant drawback is that the attraction region of the controller (the contour in control space where the controller will converge to zero error) varies strongly with $k_d$ and $d$. When either of those values is too large, (5.2) will be dominated by the $k_d \cdot d$ term and lead to very high rate rotational velocity commands. Because of their large magnitude, these commands will basically cause the robot to spin-in-place extremely fast or do large loops far from the desired state; either of these behaviors will not lead to convergence. For precision navigation, the steering algorithm should converge quickly to the desired path, necessitating larger values of $k_d$, but the larger the $k_d$ used, the less lateral offset $d$ required to put *second-order steering* outside of its region of attraction.

Solving this problem required a re-examination of the *second-order steering* algorithm. Specifically, there are two error terms used in (5.2): $k_d \cdot d$ and $k_\Psi \cdot \delta\Psi$. The $k_\Psi \cdot \delta\Psi$ term, which zeros out error between the desired state's orientation and the robot's current orientation, is relatively stable because the error between desired orientation and actual robot orientation, $\delta\Psi$, is bounded to the interval $[-\pi, \pi]$. Because of that bounding constraint, $k_\Psi$ can be tuned relative to that interval to be both aggressive in converging to zero orientation error while not having a detrimental effect on the controller's attraction region. The other error term in (5.2), $k_d \cdot d$, zeros

out lateral offset error between the desired state and the robot's current state. In effect, it produces a desired heading error between the robot's current orientation and the desired state's orientation such that the robot will move towards the desired state and not simply along the desired state's orientation. This insight led the second steering algorithm developed for this thesis, *phase space steering* (see Section 5.2.2).

### 5.2.2 Phase Space Steering

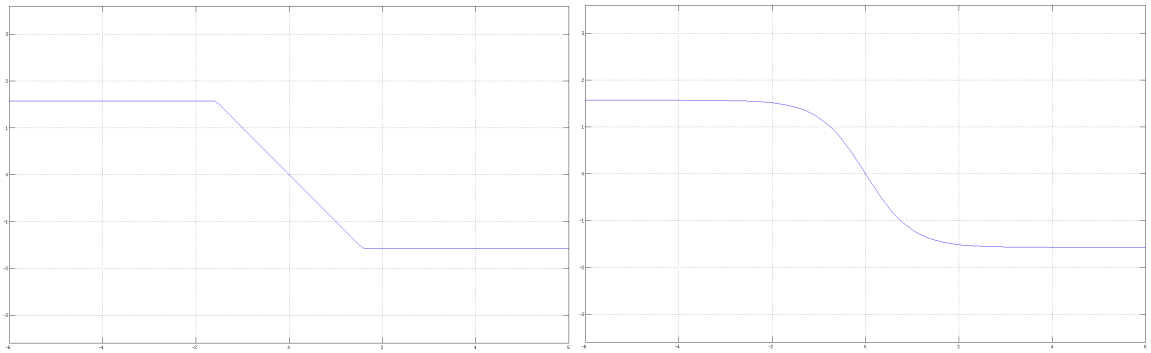
The second, improved steering algorithm used in this thesis is called *phase space steering*. This algorithm implements (5.1) and (5.3).

$$\omega_{cmd} = k_{\Psi} \cdot (\delta\Psi - f(d)) + v * \rho_{des} \tag{5.3}$$

In (5.3), $\omega_{cmd}$ is the output rotational velocity, $v$ is the output of (5.1), $\rho_{des}$ is the curvature from the desired state and $k_{\Psi}$ is a tunable gain. $\delta\Psi$ and $d$ have have the same meaning as previously described in Section 5.2.1 and shown in Figure 10, with $\delta\Psi$ constrained to the interval $[-\pi, \pi]$. The key difference between (5.2) and (5.3) is in the treatment of the lateral offset $d$. In (5.3), $f$ is a non-linear mapping function that maps a given lateral offset $d$ to a desired $\delta\Psi$, such that that desired $\delta\Psi$ would cause the robot to head back towards the desired state and reduce the lateral offset $d$. The particular non-linear mapping function used in this thesis is a simple piecewise linear function, shown in Figure 11a and given in (5.4).

$$f(d) = \begin{cases} -\pi/2 & \text{if } s \cdot d < -\pi/2 \\ s \cdot d & \text{if } -\pi/2 \leq s \cdot d \leq \pi/2 \\ \pi/2 & \text{if } s \cdot d > \pi/2 \end{cases} \tag{5.4}$$

In (5.4), $d$ is the lateral offset that needs to be mapped to a desired $\delta\Psi$. $s$ is the slope of the line used and is the only tunable parameter of this mapping function. The



(a) Linear Function (b) Hyperbolic Tangent Function

Figure 11: Sample Phase Space Non-Linear Mapping Functions. The horizontal axis in both of these graphs is the lateral offset $d$ and the vertical axis is the desired $\delta\Psi$

output is bounded to the interval $[-\pi/2, \pi/2]$ as, for any robot orientation and lateral offset, no more than a ninety degree rotation is required to point the robot towards the desired state. Limiting the output to that interval also prevents arbitrarily large lateral offsets from removing the controller from its attraction region. For any $d$ and $\delta\Psi$, a properly tuned *phase space steering* controller will eventually converge to the desired state vector given enough obstacle-free space.

While the simple piecewise linear function used in this thesis worked well enough, different non-linear mapping functions could be even better. For example, (5.4) has a pair of discontinuities at the transitions between pieces, easily seen in Figure 11a. These discontinuities did not lead to any observed problems, but could be removed while maintaining the same general shape by using a mapping function based on a sigmoid function (see [28] for more details on sigmoid functions). See Figure 11b for an example of a sigmoid function based non-linear mapping that uses the hyperbolic tangent function. The hyperbolic tangent does not have any discontinuities as it approaches its limits and is easily shifted to the interval $[-\pi/2, \pi/2]$ by multiplying with $\pi/2$, so may be an attractive alternative to the simple piecewise linear function.

## 5.3 Lfollow Discussion

As mentioned in Section 5.2, the $Lfollow$ term of (5.1) deserves further discussion, specifically the interaction with HARLIE's lower-level velocity control algorithms. For each drive wheel, HARLIE runs an independent proportional-integral-derivative (PID) controller [25]. The job of HARLIE's PIDs is to ensure that each wheel's actual velocity converges on its commanded velocity, even on different surfaces or with different amounts of resistance. This makes for much more accurate motion than simple open-loop control (essentially just the proportional part of the PID), but is sensitive to tuning of the P, I and D parameters. There were two different sets of PID parameters used on HARLIE: one set of parameters made the wheel velocity controllers overdamped systems while another set made them critically damped systems. In an overdamped system, the PID controller doesn't take the process variable (wheel velocity in this case) above the setpoint (commanded wheel velocity in this case); in a critically damped system, the PID controller quickly ramps the process variable up to the setpoint but will oscillate (with damping) around the setpoint until eventually converging on the setpoint.

For $Lfollow$, the important difference between the overdamped and critically damped controllers is the different time constant (how long it takes for the PID to achieve the setpoint). The overdamped controller takes longer to achieve the commanded velocity than the critically damped controller. Where this becomes a problem is that the $Lfollow$ term also has a sort of integral effect if the time constant on the underlying velocity controller is long (note that HARLIE's PIDs ran at 100 Hz, so updated about 5 times per steering loop). To see how this causes issues, consider the following sequence of events that could take place with an overdamped velocity controller:

1. Robot starts up and awaits new commands

2. Steering gets a new desired state and outputs a set of commands to the robot base

3. Overdamped velocity controllers get the wheel velocities part of the way to the desired velocities

4. Steering recomputes errors and, because the base did not reach the desired velocity (and therefore desired position), $Lfollow$ shows that the robot has fallen behind the state. Steering outputs larger velocity commands to make up for falling behind.

5. Overdamped velocity controllers again do not get the wheel velocities to the desired velocities

6. Steering recomputes errors again and, once again, finds that it's fallen behind, causing $Lfollow$ to again increase velocities

7. Overdamped velocity controllers still don't quite make it to the desired velocities

8. Steering recomputes errors again and, this time, finds that it's ahead of the desired state, causing $Lfollow$ to decrease the desired velocities

9. Overdamped controllers attempt to reduce velocity, but don't get quite far enough

10. Steering recomputes errors and is still ahead, further reducing desired velocities

11. Overdamped controllers get close to reduced velocities

12. Steering recomputes errors and finds that it has fallen behind the desired state, causing $Lfollow$ to increase the desired velocities

13. Repeat from 3

This situation easily illustrates the conflict that can arise when the steering system uses $Lfollow$ with a overdamped low-level velocity controllers that have a long time constant. In this case, the $Lfollow$ term causes the velocity commands to oscillate and never converge to the velocity that is part of the desired state. This situation was easily resolved by tuning the low-level velocity controllers to be critically damped as opposed to overdamped. This caused the velocity controllers to get the actual wheel velocities very close to the desired velocities promptly, so the $Lfollow$ term did not adjust the desired velocity any appreciable amount. This avoided the oscillations caused by $Lfollow$ when used with the overdamped controller and proved to be a better setup for precision navigation.

# 6 Trajectory Generation

Another major component of the path execution subsystem developed in this thesis is trajectory generation. This component sits between the steering and path planning components, taking global paths from path planning and outputting desired states that the steering component attempts to achieve. The trajectory generator fills the same role as the base_local_planner described in Section 3.2. The input to the trajectory generator is an ordered sequence of path segments (called a *path*). The path segments are described in Section 6.1. The output is a desired state for the current timestep, as described in Section 5.1.

## 6.1 Path Segment Description

The path segment used by the trajectory generator developed for this thesis is described in Table 2. There are three different types of path segments used in this thesis – straight lines, smooth constant curvature arcs and spin-in-places. One important assumption about these path segments is that, for a given sequence of segments, they will already be blended together to remove any discontinuities in the geometric parameterization between the end of a segment and the beginning of the following segment.

Of the fields listed in Table 2, the ones requiring the most explanation are the *reference_point* and the *initial_tangent_angle*. The *reference_point* can have the following meanings, depending on *segment_type*. For a straight line segment, the reference point is the start point of the line segment. For a constant curvature arc, the reference point is the center of the circle that the arc belongs to (the radius of that circle is $1/curvature$). For a spin-in-place segment, the reference point is the point about which to spin. The *initial_tangent_angle* can have the following meanings, depending on *segment_type*. For a straight line segment, the initial tangent angle is the direction

| Name | Description |
|---|---|
| header | This is a standard ROS header type that contains information such as the reference frame the rest of the fields are in and the timestamp for when the path segment was generated |
| segment_type | An integer enum representing the type this segment, such as a straight line segment, constant curvature arc segment or spin-in-place segment. |
| segment_number | The ID number of the segment that generated this state. |
| segment_length | The length of the segment. Whether it is in meters or radians depends on the *segment_type* |
| reference_point | The reference point for this path segment. Interpretation depends on the *segment_type* |
| initial_tangent_angle | The initial tangent angle for this segment. Interpretation depends on the *segment_type* |
| curvature | The curvature of this segment. Exact interpretation depends on the *segment_type*. For straight lines, curvature should be 0.0 |
| max_speed | A pair of the maximum translational speed and maximum rotational speed to be used for this segment |
| min_speed | A pair of the minimum translational speed and minimum rotational speed to be used for this segment |
| acceleration_limit | The acceleration limit for this segment. Whether it is in $m/s^2$ or $rads/s^2$ depends on the *segment_type* |
| deceleration_limit | The deceleration limit for this segment. Whether it is in $m/s^2$ or $rads/s^2$ depends on the *segment_type* |

Table 2: Path Segment Field Description

of the line segment. For a constant curvature arc, the initial tangent angle defines the actual point along the circle where the the arc segment begins. For a spin-in-place segment, the initial tangent angle is the angle that the spin should start from.

As discussed in Section 5.1, the spin-in-place segment type was added to the original path segment types to avoid numerical instability issues when using arcs with very small radii.

## 6.2 Trajectory Generation Algorithms

The trajectory generation algorithm used in this thesis is detailed in this section. On HARLIE, this loop ran at 20 Hz to regularly send commands to the steering algorithms detailed in Section 5.2. Note that the trajectory generator and steering algorithm, while both run at 20 Hz, are not explicitly synchronized. It should also be noted that, while many of these concepts are not limited to SE(2) space (position and rotation on the XY plane), these algorithms are written assuming SE(2) space and not a more generalized six degree of freedom system (SE(3) space).

Algorithm 3 describes the trajectory generation algorithm that runs at the loop rate of 20 Hz. Its inputs are the last desired state ($state_{t-1}$), the current path from the path planner ($path$), the last segment number being executed ($i_{t-1}$) and the distance traveled along that segment already ($l_{t-1}$). The outputs (which are fed into the algorithm on the next loop as well as being sent to the steering algorithm) are the current desired state ($state_t$), the current segment number being executed ($i_t$) and the distance traveled along that segment after this loop ends ($l_t$). Note that the segment number being executed, $i_t$, is an index into the sequence of path segments contained in the $path$. Also note that all of the algorithms described here assume that positions and orientations have all been placed in the same reference frame; on HARLIE, this was done with the ROS "tf" library [30].

There are a number of algorithms used in Algorithm 3. Two of these are fairly

**Algorithm 3:** Trajectory Generation Algorithm

**Input**: $state_{t-1}$, $path$, $i_{t-1}$, $l_{t-1}$
**Output**: $state_t$, $i_t$, $l_t$

```
1  robot_state_t = GetCurrentRobotState()
2  if path ≠ ∅ then
       // Compute new desired state
3      i_t, l_t = UpdateSegNumAndDist(robot_state_t, path, i_{t-1}, l_{t-1}, state_{t-1})
4      v_t = ComputeVelocityCommand(path, i_t, l_t)
5      pose_t = ComputeDesiredPose(path, i_t, l_t)
6      state_t.segment_type = path[i_t].segment_type
7      state_t.pose = pose_t
8      state_t.rho = path[i_t].curvature
9      state_t.velocity = v_t
10
       // Check new desired state for collisions
11     if state_t will be in collision then
12         state_t = MakeHaltState(robot_state_t)
13 else
14     i_t = 0
15     l_t = 0.0
16     state_t = MakeHaltState(robot_state_t)
```

simple: `GetCurrentRobotState` and `MakeHaltState`. `GetCurrentRobotState` gets the current robot position, orientation and velocities in a robot specific manner. For HARLIE, this information came from the output of the localization subsystem described in Section 4. `MakeHaltState` takes the given state (usually the current robot state) and generates a state that would cause the steering algorithm to stop and maintain that state. In this case, it simply generates a spin-in-place state at the robot's current position and orientation with zero speed and curvature; effectively, this state causes steering to stop both translational and rotational motion until given a new commanded state.

The other three algorithms used in Algorithm 3 are more complicated and require more detailed explanation. In all of these algorithms, `Length` is a simple function that returns the number of path segments in the *path* it is given. These algorithms are based on zero-indexed sequences (such as arrays in C) as opposed to one-indexed containers such as those found in Matlab. Also, in all of these algorithms, $dt$ is the time since the last loop ended (on HARLIE, $dt = \frac{1}{20Hz}$).

Algorithm 4 is the `UpdateSegNumAndDist` function used in Algorithm 3. Its inputs are the current robot state ($robot\_state_t$), the current path ($path$), the last segment number being executed ($i_{t-1}$), the distance traveled along that segment already ($l_{t-1}$) and the last desired state ($state_{t-1}$). Algorithm 4 updates the distance traveled along the current segment according to the velocity in $state_{t-1}$ while moving to the next segment when appropriate and not running past the end of the $path$. It then outputs these new values for the segment number being executed ($i_t$) and distance already traveled along that segment ($l_t$) for consumption by the remainder of Algorithm 3.

Two important details of this computation are on line 8 and line 16 of Algorithm 4. Line 8 projects the robot's motion onto the the tangent vector of the last state. This prevents situations where the robot is moving nearly orthogonal to the desired state's tangent vector from advancing the distance traveled along the path segment too far for

**Algorithm 4:** Update Segment Number and Distance Algorithm

**Input**: $robot\_state_t$, $path$, $i_{t-1}$, $l_{t-1}$, $state_{t-1}$
**Output**: $i_t$, $l_t$

**if** $i_{t-1} \geq$ `Length(`$path$`)` **then**
    $i_t =$ `Length(`$path$`)` $- 1$
    $end =$ true
$\delta l = state_{t-1}.velocity \cdot dt$
**if** $path\,[i_t]$ *is a spin-in-place segment* **then**
    $l_t = l_{t-1} + \delta l$
**else**
    $l_t = l_{t-1} + \delta l \cdot \cos\left(state_{t-1}.orientation - robot\_state_t.orientation\right)$
$\hat{l} = path\,[i_t]\,.segment\_length$
**if** $l_t > \hat{l}$ **then**
    $l_t = 0.0$
    $i_t = i_t + 1$
**if** $i_t \geq$ `Length(`$path$`)` **then**
    $i_t =$ `Length(`$path$`)` $- 1$
    $end =$ true
$\hat{l} = path\,[i_t]\,.segment\_length$
**if** *end is true* **then**
    $l_t = \hat{l}$

the actual distance along the path segment traveled. These situations can occur while using *phase space steering* if there is a large lateral offset from the path – *phase space steering* is designed to move nearly orthogonal to the desired state's tangent vector in that situation. Line 16 is required to ensure that the correct current segment and segment length are selected, even after all of the adjustments to the current segment number ($i_t$) have been made.

**Algorithm 5:** Compute Velocity Command Algorithm

**Input**: $path$, $i_t$, $l_t$
**Output**: $v_t$

```
1  if path[i_t] is a spin-in-place segment then
2      v_{t+1} = 0.0
3      v_t = path[i_t].max_speed.omega
4  else
5      v_t = path[i_t].max_speed.linear
6      if i_t < Length(path) − 1 then
7          v_{t+1} = path[i_t + 1].max_speed.linear
8      else
9          v_{t+1} = 0.0
10 dec = path[i_t].deceleration_limit
11 dec_dist = ((v_t + v_{t+1}) / 2.0) · ((v_t − v_{t+1}) / dec)
12 rem = path[i_t].segment_length − l_t
13 if rem < 0.0 then
14     rem = 0.0
15 else if rem < dec_dist then
16     v_t = sqrt(2.0 · rem · dec + v_{t+1}^2)
17 else
18     v_t = v_t + path[i_t].acceleration_limit · dt
19 if path[i_t] is a spin-in-place segment then
20     clamp v_t to interval [path[i_t].min_speed.omega, path[i_t].max_speed.omega]
21 else
22     clamp v_t to interval [path[i_t].min_speed.linear, path[i_t].max_speed.linear]
```

Algorithm 5 is the `ComputeVelocityCommand` function used in Algorithm 3. Its inputs are the current path ($path$), the segment number being executed ($i_t$) and distance already traveled along that segment ($l_t$), as output by Algorithm 4. Its

only output is the desired velocity command ($v_t$). Algorithm 5 has three steps. First, the algorithm initializes to the max speed for the current segment ($v_t$) and determines the max speed of the next segment ($v_{t+1}$). Next, it determines whether it can continue accelerating or needs to start braking in order to slow down to $v_{t+1}$ before the beginning of the next segment. Finally, the calculated velocity command $v_t$ is clamped to fall within the interval defined by the current segment's minimum and maximum speeds. After clamping, $v_t$ calculation is complete.

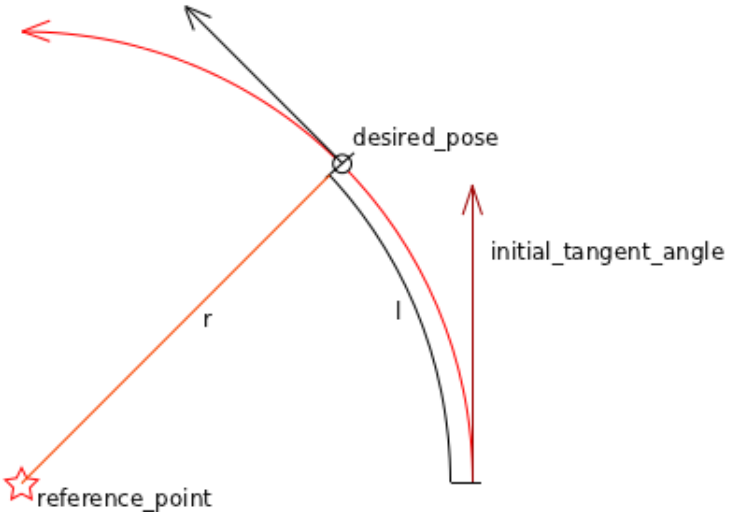
Algorithm 6 is the `ComputeDesiredPose` function used in Algorithm 3. Its inputs are the current path ($path$), the segment number being executed ($i_t$) and distance already traveled along that segment ($l_t$), as output by Algorithm 4. Its only output is the desired pose ($pose_t$). Algorithm 6 is made up of three cases, depending on what type of path segment the pose is being computed from. For a straight line segment, Algorithm 6 moves the reference point in the direction of the tangent vector (as defined by the path segment's initial tangent angle and sets the orientation to face along that tangent vector. For a constant curvature arc segment, it must find the correct point along the circle $\frac{1}{|\rho|}$ away from the segment's reference point as well as compute the correct tangent angle for that point on the circle. For a spin-in-place segment, it only needs to set the desired pose's orientation according to how far along the segment the trajectory generator is; because it is a spin-in-place, the desired position is the same as the segment's reference point.

See Figure 12 for the geometric relationships between the fields listed in Table 2 for each segment type; these geometric relationships are what Algorithm 6 is based on. For all of these figures, red denotes features that are described by the path segment while black features indicate features that are based on the current trajectory generator state. For example, in Figure 12a, the red line, labeled **r**, between the reference point and the desired pose represents the radius of the arc (or $1/curvature$) while the shorter black arc, labeled **l**, represents the trajectory generator's current

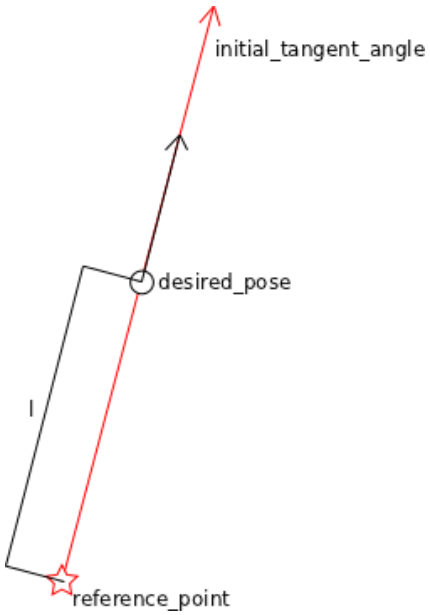
**Algorithm 6:** Compute Desired Pose Algorithm

**Input**: $path$, $i_t$, $l_t$
**Output**: $pose_t$

$\Psi = path\,[i_t]\,.initial\_tangent\_angle$
**switch** $path\,[i_t]\,.segment_t ype$ **do**
  **case** *straight line segment*
    $pose_t.position = path\,[i_t]\,.reference\_point + l_t \cdot \begin{bmatrix}\cos\Psi \\ \sin\Psi\end{bmatrix}$
    $pose_t.orientation = \Psi$
  **case** *constant curvature arc segment*
    $\rho = path\,[i_t]\,.curvature$
    $r = \frac{1}{|\rho|}$
    **if** $\rho \geq 0.0$ **then**
      $\hat{\Psi} = \Psi - \frac{\pi}{2}$
    **else**
      $\hat{\Psi} = \Psi + \frac{\pi}{2}$
    $\delta\Psi = l_t \cdot \rho$
    $pose_t.position = path\,[i_t]\,.reference\_point + r \cdot \begin{bmatrix}\cos\left(\hat{\Psi} + \delta\Psi\right) \\ \sin\left(\hat{\Psi} + \delta\Psi\right)\end{bmatrix}$
    $pose_t.orientation = \Psi + \delta\Psi$
  **case** *spin-in-place segment*
    $\rho = path\,[i_t]\,.curvature$
    **if** $\rho \geq 0.0$ **then**
      $\hat{\Psi} = \Psi - \frac{\pi}{2}$
    **else**
      $\hat{\Psi} = \Psi + \frac{\pi}{2}$
    $\delta\Psi = l_t \cdot \rho$
    $pose_t.position = path\,[i_t]\,.reference\_point$
    $pose_t.orientation = \Psi + \delta\Psi$

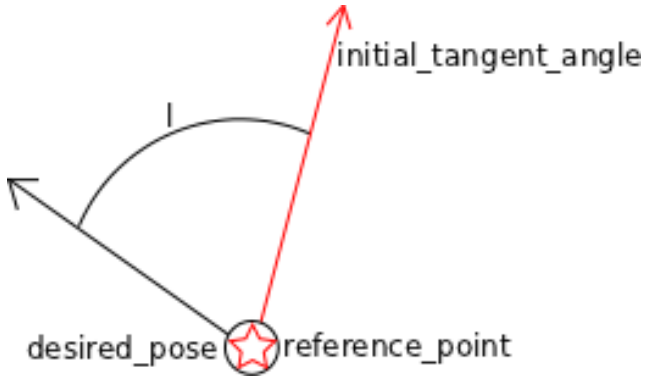



(a) Constant Curvature Arc




(b) Straight Line

Figure 12: Labeled Path Segment Geometries Figures (a) and (b)



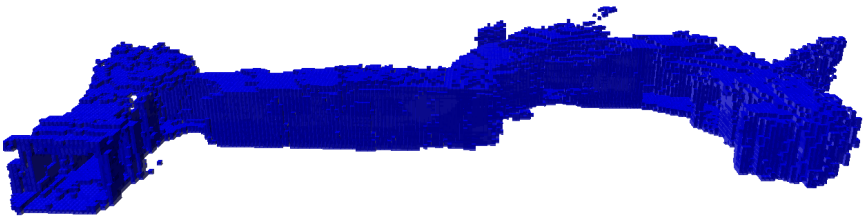

(c) Spin-in-Place

Figure 12: Labeled Path Segment Geometries Figure (c).

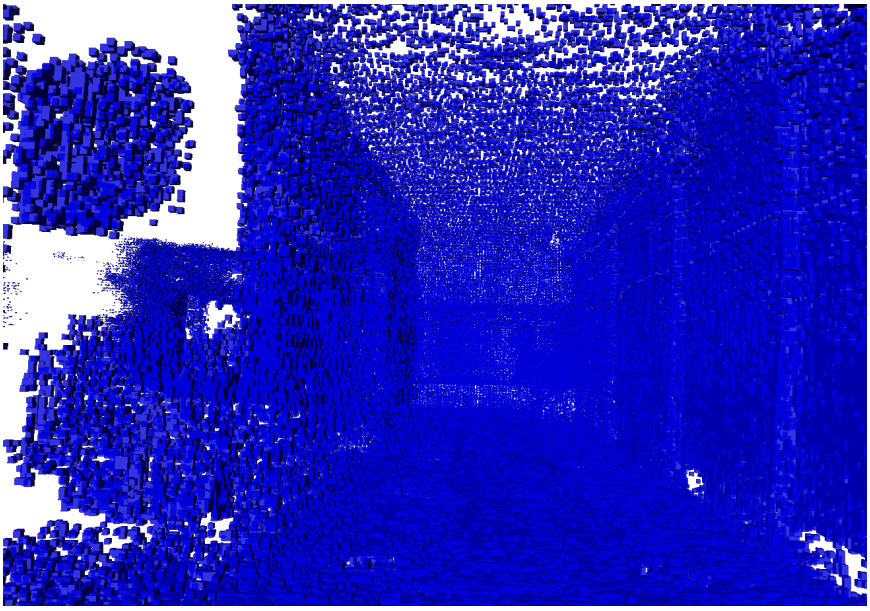
distance along the path segment ($l_t$) and is what the labeled **desired_pose** ($pose_t$) is calculated from.

## 6.3 Feedback for Planning

In the original design of the trajectory generator, the outputs of Algorithm 3 were not exposed outside of the trajectory generation subsystem. Things like the current path ($path$), the segment number being executed ($i_t$) and distance already traveled along that segment ($l_t$) were only fed back into Algorithm 3. This lack of information became a problem for the path planning system when it needed to replan around a dynamic (or otherwise unexpected) obstacle that invalidated the original path plan. In order to replan efficiently, the path planner needed to be aware of where along the previous path plan the trajectory generator was when it encountered the obstacle. Without that information, the path planner was unable to generate a new path plan that avoided the obstacle and was still smoothly blended with the segments of the original plan that were unaffected by the obstacle's presence. To provide the path planner with the information required, the trajectory generator output the $path$ it was currently using, the current segment being executed $i_t$ and the distance traveled along that segment $l_t$. With this information, the path planner could easily extrapolate exactly where the trajectory generator had commanded the robot before detecting



(a) Side View, 20 cm Resolution



(b) Inside View, 2.5 cm Resolution

Figure 13: costmap3d Examples. These are from an OctoMap built from 3D sensor data of the Glennan 3rd Floor while using costmap3d.

the collision and efficiently modify the path to avoid that collision. On HARLIE, this feedback mechanism was implemented using the ROS "actionlib" library [1].

## 6.4 Collision Detection

In order to do precise navigation in complex, dynamic environments it is not simply enough to precisely follow a planned path – the system must also avoid collisions with objects in the environment. The first step to avoiding a collision is detecting that it is going to occur. There were two different methods for collision detection used in this trajectory generation subsystem: one based on existing ROS obstacle mapping and the other based on an octree representation of the environment. The

original collision detection system used functionality based on the ROS costmap_2d described in Section 3.1. This method suffered from many of the drawbacks described in Section 3.1, specifically the difficulty of maintaining a high resolution obstacle map in an efficient manner. The collision detection system itself was also difficult to use for checking the collision of a single pose efficiently; since the functionality was originally part of the ROS base_local_planner (see Section 3.2), it was designed for checking a set of controls from the current robot pose and not a given state that was part of a path segment. It also did not have support for forward simulating, while checking for collisions, a desired state along one of the path segments used in this thesis, but could only forward simulate a set of constant velocity commands (translational and rotational).

To address these shortcomings in the existing ROS obstacle mapping, a new obstacle mapping component was developed for use by trajectory generation. This component, called costmap3d uses the OctoMap library [33] to allow efficient, hi-resolution 3D collision checking in large environments (see Figure 13 for an example of an OctoMap created by costmap3d). costmap3d improves three of the issues with the ROS obstacle mapping and collision checking mentioned above and in Section 3.1: simplistic integration of sensor measurements over time, efficiently representing large environments at small resolutions and a simple method for efficiently checking a single position in the map for collisions.

The first two issues, simplistic integration of sensor measurements over time and efficiently representing large environments at small resolutions, are addressed via the use of the OctoMap library. OctoMap uses a probabilistic representation of sensor measurements (derived from the occupancy grid mapping techniques introduced in [22]) to compute the probability of a given map cell being either free space or occupied. This means that the occupancy state of a map cell is based on all of the previous measurements that intersected with that cell and not simply the last measurement

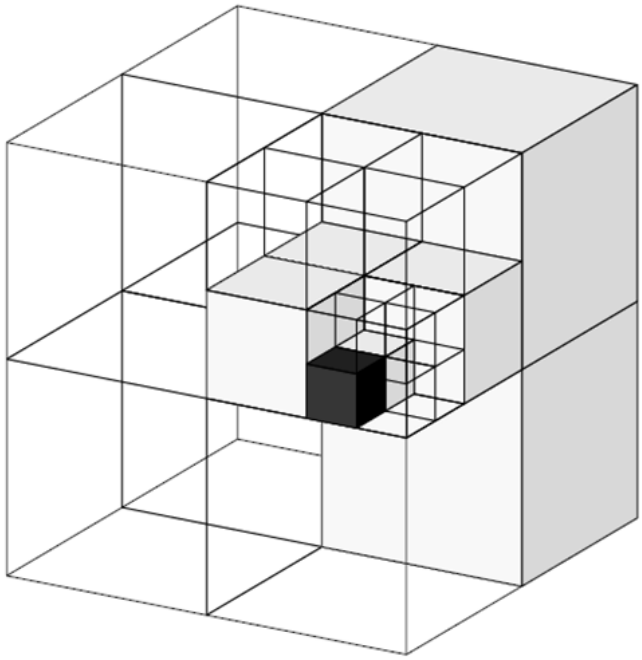


Figure 14: Example Octree Structure [33]

that did so (as is the case in costmap_2d).

OctoMap also solves the issue of efficiently representing large environments at small resolutions. It does so by using an octree representation for the map as opposed to a fixed size grid (where the cells can either be 2D as in the case of costmap_2d or 3D as in the case of a voxel grid). Octrees, very similar to quadtrees used for 2D environments, are a nested, recursive data structure where each voxel in the octree can be further subdivided into eight more voxels, thereby forming an octree inside of that voxel (see Figure 14). Because groups of smaller voxels with the same label can be represented by keeping the containing larger voxel, octrees can represent very large environments at high resolutions with little memory usage (compared to a traditional fixed size occupancy grid). For example, an empty hallway with a door at the end, a common environment for this precision navigation system, can easily be represented by an octree by maintaining the large, parent voxels in the empty hallway while using small voxels to represent the area near the doorway at a small enough resolution such

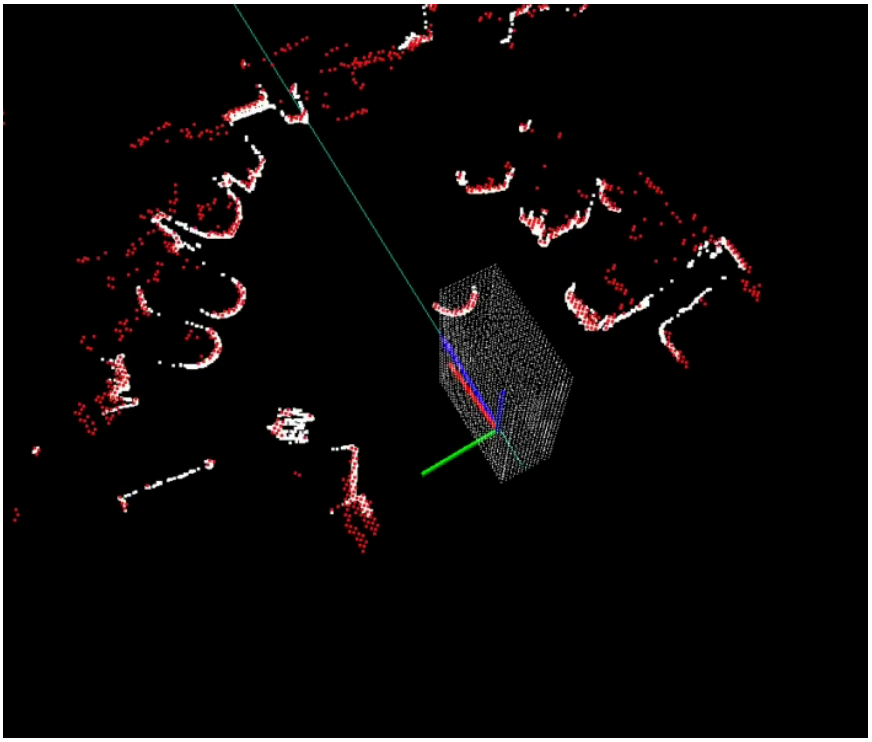


Figure 15: HARLIE costmap3d Collision Box Example

that the doorway does not occlude free space simply because the grid size is too large. To do this, OctoMap regularly "prunes" the underlying octree occupancy map to merge groups of voxels that have the same occupancy label (e.g. free or obstacle) into a single, larger containing parent voxel with that same label. After pruning, there is no need to keep the children in memory as they are losslessly represented by the parent voxel; this greatly reduces the memory requirements for representing large environments at high resolution, especially if those environments can be grouped into clumps of free and occupied space.

Though OctoMap solves the obstacle mapping issues of costmap_2d described above, the OctoMap library itself does not have any facilities for efficiently checking a given robot pose for collisions. OctoMap does provide functionality for getting the occupancy state of a given point, but no facilities for checking something like the bounding box of the robot at a given pose. That is the exact functionality provided by the costmap3d component developed for this thesis. The costmap3d contains an

OctoMap of the environment (constantly being updated with new sensor data) and provides a simple function for checking for collisions at a given robot pose, given the robot's height, width and length (these define the robot's bounding box). To do this, it approximates the faces of the bounding box with a dense set of points and checks each point (see Figure 15 for an example of HARLIE's set of points) – as soon as a single point is in collision, costmap3d can stop checking and return that that pose would be in collision. For HARLIE, the points were generated at a resolution of five centimeters and the costmap3d checked up to ten forward simulated states in the trajectory generation loop that was running at 20 Hz. The costmap3d library was also used successfully by Chad Rockey for 3D collision checking on a smart wheelchair [26].

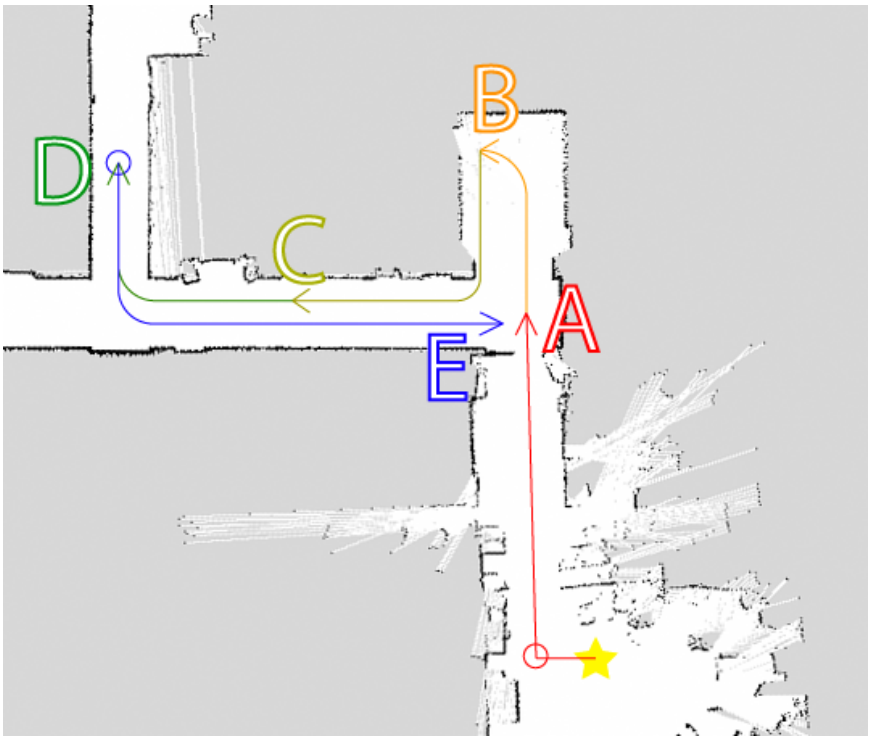



Figure 16: Annotated Goals with Pre-planned Paths Example. This is the second floor of the CWRU Glennan building. Each separate path segment is in a different color. Goals are annotated with a letter in the same color as the path leading to that goal. Circles are spin-in-place segment locations. Arrows indicate the direction of the paths. The star is the intended initial robot pose, which is in the Glennan 210 Robotics Lab

# 7 Path Planning

The third major component of the path execution subsystem is a path planner. This component sits above the trajectory generation component and takes goals from other robot subsystems and outputs paths that the precision navigation system should follow. The path planner fills a similar role to that of the navfn described in Section 3.3. The output is an ordered sequence of path segments (see Section 6.1 for the detailed description of a path segment) that must be blended smoothly such that there are no discontinuities in position or orientation between the end of one segment and the beginning of the next segment.

While a complete path planning system able to generate blended paths to arbi-

trary goals in arbitrary environments was beyond the scope of this thesis, a minimal path planner for using the precision navigation system on *a priori* paths to pre-defined goals with limited capability for replanning around dynamic obstacles was developed. This minimal path planner was used to validate the performance of the other precision navigation components in an autonomous system, where the only human action required is the selection of a destination location. See Figure 16 for an example of a set of pre-planned paths with corresponding annotated goals (simply A, B, C, etc. in this example but they could easily be "elevator" or "bathroom"). Possible extensions to this minimal path planner are discussed in Section 9.

The path planner developed in this thesis has a number of assumptions about the environment it is operating in. It should be noted that none of these assumptions are necessarily required for precision navigation with a sufficiently advanced path planning component and are specific to the limited path planner developed in this thesis. The first of these assumptions is that, in addition to an *a priori* obstacle map of static obstacles, this map will be annotated with goal locations and that there is an ordered sequence of path segments describing how to get from one goal to another. These goal locations and pre-planned paths are the only paths that the planner can send to the rest of the precision navigation system. For this thesis, the goals were selected to be "interesting" destination locations for a smart wheelchair, such as "elevator" or "bathroom". The paths between these goal locations were generated by driving the robot along the desired path while recording control points along the path. These control points were then split into different path segments and the geometric parameters were fit to each segment's control points using Matlab. For example, for a constant curvature arc segment, four control points were required in order to solve for the reference point, curvature and segment length (the initial tangent angle is fixed by the orientation of the robot at the first control point) **To do** (1) . Parameters such as maximum speed and acceleration limits were inserted by hand based on desired

behavior and HARLIE's safety limits.

The next set of assumptions is related to the initial conditions of the robot when a given pre-planned path is selected. Firstly, the path planner assumes that, when a given path is commanded, the robot is at the origin of the first path segment, as there is no automatic segment generation to connect the current robot pose to the origin of the commanded path. If the robot is not near the origin of the path when the path is commanded, whether the robot will be able to complete the path depends on the steering algorithm's attraction region, as the desired states output by the trajectory planner will always lie on a path segment. If the robot's pose is within the steering algorithm's attraction region, the robot will eventually converge onto the commanded path and precisely follow it; if the robot's pose is outside the attraction region, it is unlikely that the robot will converge on the commanded path. Even if the robot is within the steering algorithm's attraction region, there is a second condition that must be satisfied if the robot is to converge onto the path: the space between the robot's current pose and the origin of the path must be free of obstacles. Because the path planner doesn't plan between the robot's current pose to the origin of the path, it cannot replan around obstacles between the robot and the origin of the path. In a normal use case, this condition is not difficult to satisfy if the robot navigated to the origin of the path using a different path; for example, if the robot first went to the "elevator" and then took the path between the "elevator" and "bathroom" in order to get to the bathroom, then it should have ended up very close to the origin of the "elevator" to "bathroom" pre-planned path and it is unlikely that there would be any obstacles within a short distance (ten centimeters or so) of a goal point. Another use case, where a human has to position the robot near to the origin of a desired path before sending the planner a goal has a similar assumption – it is unlikely that a human, manually positioning the robot, would position it such that there is an obstacle between the robot and the origin of the path.

Even with these limitations, the path planner developed for this thesis was sufficient for precisely navigating in static indoor environments. In order to validate that dynamic replanning around obstacles was possible within the framework of the precision navigation system, the path planner was extended with the ability to "splice" in a sequence of path segments to go around the dynamic obstacle encountered. The splicing, which takes place whenever an imminent collision is detected by the trajectory generation, is made up of the following steps:

1. Wait for $n$ seconds, aborting splicing if there will no longer be a collision (e.g. if a person was simply walking across the robot's path)

2. Confirm the desired state (with forward simulation along the path segment) will still cause a collision

3. Shorten the current path segment to end at the current desired state

4. Add the following path segments to the path between the shortened current path segment and the next segment of the original path

   (a) Spin-in-place $\frac{\pi}{2}$ radians to be orthogonal to the original path segment

   (b) Semicircle with a one meter radius

   (c) Spin-in-place $\frac{\pi}{2}$ radians to the orientation of the current path segment

   (d) Add a straight line segment from the last spin-in-place to the next segment

5. Send path with spliced segments and the remainder of the original path to the trajectory generator

While these splicing steps are somewhat limited in that they assume that the obstacle will either move out of the way (the reason for waiting for $n$ seconds, fifteen in this thesis, before splicing) or that a semicircle with a one meter radius will be sufficient to go around the obstacle. Though the splicing behavior is a rather primitive

method of replanning to avoid dynamic obstacles, it was sufficient to function as a proof-of-concept that the precision navigation system can avoid dynamic obstacles.

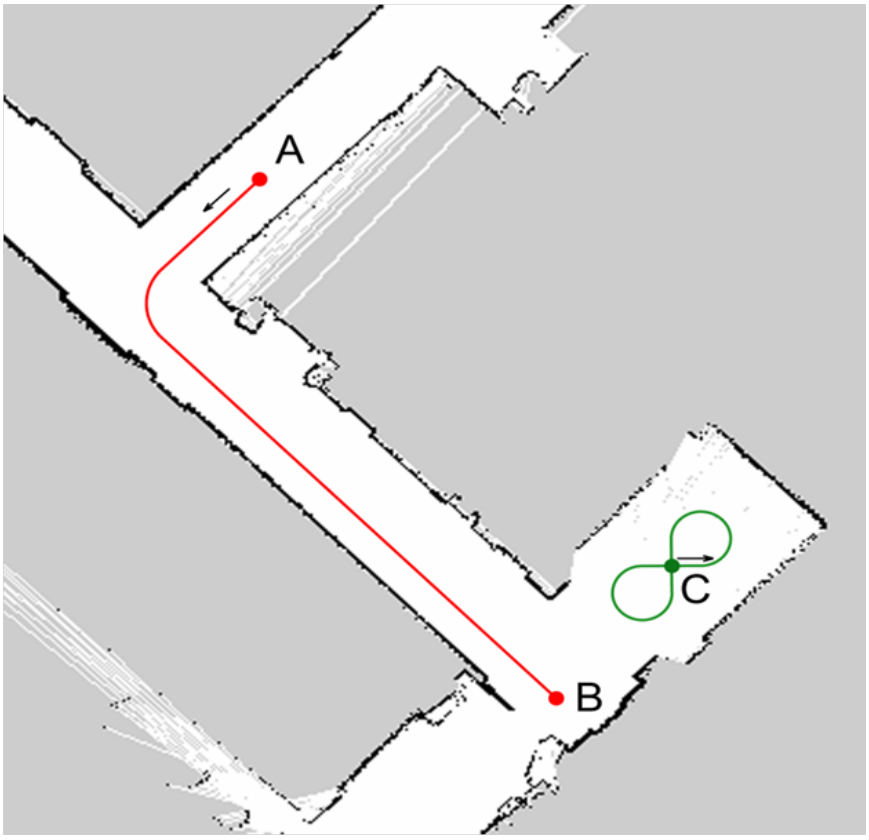



Figure 17: Paths for Navigation Precision Tests

# 8 Results

In order to evaluate the precision navigation system developed in this thesis, a number of different experiments, both on HARLIE and in simulation, were conducted. There were three major sets of experiments conducted: repeatability tests of the entire precision navigation system, tests to evaluate the acceptable initial conditions for phase space steering and tests to validate the splicing method used.

## 8.1 Path Following Precision

The first set of tests run to evaluate the performance of the precision navigation system described in this thesis were designed to measure the actual precision with which it executed a path. To do this, two distinct sample paths were created on the

second floor of the Case Western Reserve University Glennan building; these paths are shown overlaid to scale on the *a priori* map in Figure 17. The first path, called the "L" path, is in red, which proceeds down a hallway from point A to point B. The second path, called the "figure-8" path, is in green. In the figure-8, the robot starts at point C facing in the direction indicated by the arrow and then proceeds around each arc before stopping again at point C. Throughout this section, a "run" on the L path refers to one complete trip starting at point A and ending at point B. For the figure-8 path, a "run" is defined as starting at point C, going through both the top and bottom loops and stopping again at point C. For the precision navigation system, both of these paths were described geometrically as a sequence of arcs and line segments.

To generate the path following performance measurements, five independent runs of the experiment were gathered while logging the pose of the robot with respect to the fixed origin of the *a priori* map as estimated by the localization subsystem described in Section 4. After all five runs were completed, the logged data was post-processed to compute lateral offsets relative to the desired path at each of the logged poses. The root-mean-square (RMS) of those lateral offsets was calculated for each individual run. This RMS value was averaged over all five runs of a given experiment and the standard deviation of those RMS values was calculated to generate a measure of repeatability.

Performance was measured in terms of RMS lateral offsets from the defined path. Figure 18 shows the performance of autonomous runs of the precision navigation system on the L path. As shown, the path following errors were small and highly repeatable. Lateral errors were just over three centimeters RMS, and this result was repeatable with a standard deviation of less than four millimeters.

The figure-8 test path had relatively tight turn radii (2 feet), making it a more challenging path. Again, performance was measured in terms of RMS lateral offsets

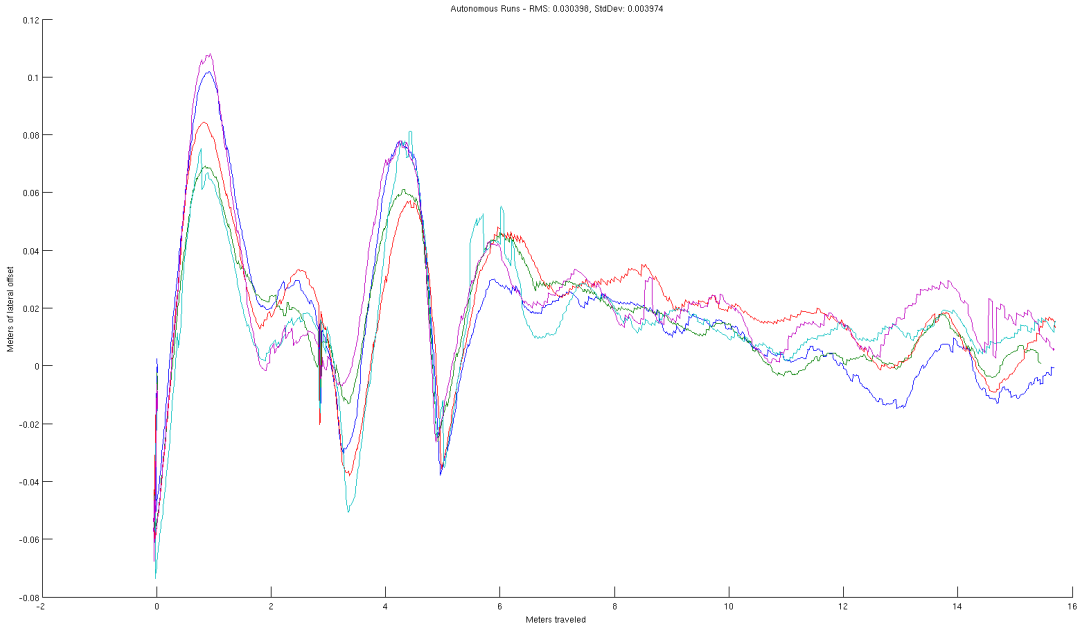



Figure 18: Precision Navigation Error on the L Path. The X-axis is meters traveled and the Y-axis is meters of lateral offset

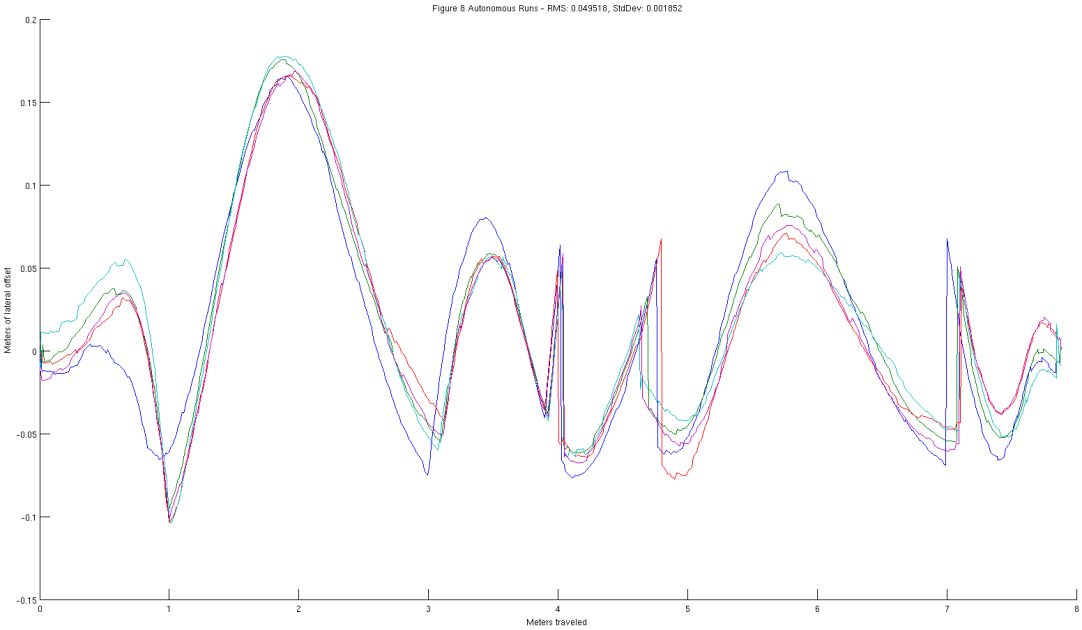



Figure 19: Precision Navigation Error on the Figure 8 Path. The X-axis is meters traveled and the Y-axis is meters of lateral offset

from the defined path. Figure 19 shows the performance of autonomous runs of the precision navigation system on the figure-8 path. As shown, the path following errors were small and highly repeatable. Lateral errors were less than five centimeters RMS, and this result was repeatable with a standard deviation of less than two centimeters.

The results for both of these tests show that the precision navigation system described in this thesis is sufficiently precise for the use cases previously described, such as navigating a robotic wheelchair through an ADA-compliant doorway that would have under seven centimeters of clearance on either side of the robot. The majority of the lateral offset error shown in Figure 18 and Figure 19 occurs at changes in the path curvatures. For example, in the L path error graph, the large variations in lateral offset at approximately one and four meters of distance traveled coincide with entering and leaving the arc between hallways. After leaving the arc (after approximately six meters traveled), the lateral offset error becomes much less variable as the robot reaches the long, straight section of the L path.

## 8.2 Phase Space Steering Initial Condition Tests

Another set of tests was conducted to gather information about the attraction region of the phase space steering algorithm (see Section 5.2.2) as well as gather quantitative data on the acceptable initial conditions for the path planner (see Section 7). Two different tests were conducted: the "door" test (see Section 8.2.1) and the "tangent" test (see Section 8.2.2). Note that all of the results of these tests depend on the gains and other parameters, as well as steering algorithm, used. For these tests, phase space steering was selected as the steering algorithm, with the following gains and parameters (as described in Section 5.2.2): $k_v = 0.1$, $k_\Psi = 1.0$ and the slope of the phase space mapping function $s = -1.0$.

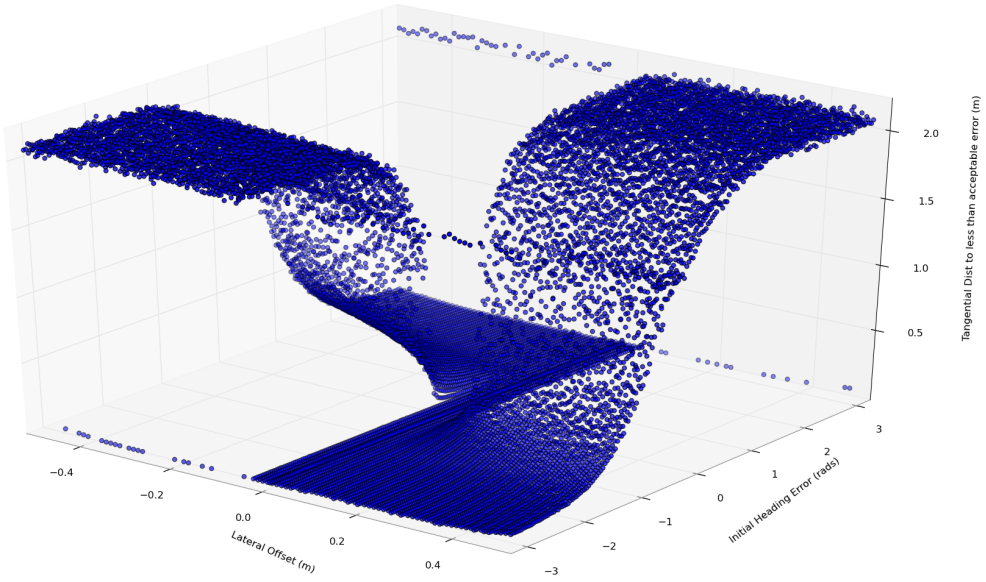



Figure 20: "Door" Test Simulation Results. The X-axis is the Initial Heading Error (rads), the Y-axis is the Initial Lateral Offset Error (m) and the Z-axis is the Tangential Distance to Path Convergence (m)

### 8.2.1 "Door" Test

The "door" test is designed to ask: Given a set of initial conditions (lateral offset and heading error), will phase space steering converge onto the path before reaching the door? This test involved commanding a path composed of a single straight line segment, positioned and oriented such that the line passes through the center of a doorway, while positioning the robot such that it had different lateral offsets and heading offsets to the path. These tests were conducted both in the Gazebo simulation environment described in Section 2.2 and a subset of those results were tested on HARLIE to verify that the simulation results transfered to the physical robot. Simulation results are shown in Figure 20.

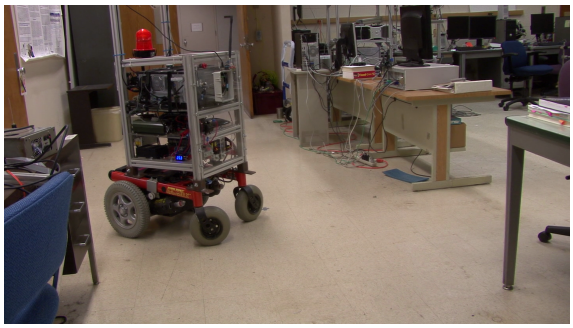
There are a number of useful results shown in Figure 20. The first is that the performance of the phase space steering algorithm is rotationally symmetric about the Z-axis. This makes sense for HARLIE, as a negative heading error and positive lateral offset will have the robot already facing towards the path and a positive

heading error and negative lateral offset will also have the robot facing towards the path. Any situation where the robot's initial conditions have it facing the path is much easier for the steering algorithm to converge onto the path in less distance along the path than a situation where the robot's initial conditions have it facing away from the path.

A second result from the simulation experiments is the actual shape of the experimental data. This shape implies that the way to converge to the path with the least distance traveled along the path is to face the path as quickly as possible before beginning to move forwards. Logically, this makes sense – moving forwards directly towards the path will converge more quickly than moving forwards at an angle towards the path. Essentially, the way to converge to the path with the least tangential distance traveled is to follow the gradient of this surface at whichever lateral offset and heading error the algorithm has – which is exactly what the phase space steering algorithm is designed to do.

This experiment also yielded information as to the minimum tangential distance required to converge to the path, given a set of initial conditions. This data can be used to determine whether, given a set of initial conditions and a doorway that must be traversed, a path planner must replan from the initial conditions to the path or if no replanning is required because there is enough distance before the door to converge to the path before reaching the door. For example, the data collected in simulation implies that, given a lateral offset of twenty-five centimeters and an initial heading error of forty-five degrees, the robot must be at least thirty-one centimeters away perpendicularly from the doorway in order to safely pass through the doorway. A number of runs were conducted with HARLIE to validate these results; these tests with the physical system confirmed the simulation results. See Figure 21 for some examples of HARLIE while conducting these tests. There are two different initial conditions shown (see Figure 21a and Figure 21b), as well as HARLIE clearing the

(a) Initial Condition A



(b) Initial Condition B

(c) Clearing Doorway

Figure 21: "Door" Test HARLIE Results. These are images captured from videos taken during the tests. The desired path runs through the middle of the door, perpendicular to the doorway.

doorway (see Figure 21c).

### 8.2.2 "Tangent" Test

The "tangent" test is designed to ask: Given a set of initial conditions (lateral offset and heading error), will phase space steering converge onto the path and reach the desired end pose without overshooting? Overshooting, in this case, is defined as the robot going more than four centimeters to the other side of the desired path (this approximates a task where the robot has been commanded to pull up very closely to a wall without hitting the wall). This test involved commanding a path composed of a

single straight line segment, positioned and oriented such that the line is parallel and near to a wall, while positioning the robot such that it had different lateral offsets and heading offsets to the path. These tests were conducted both in the Gazebo simulation environment described in Section 2.2 and a subset of those results were tested on HARLIE to verify that the simulation results transfered to the physical robot. Simulation results are shown in Figure 22, where a tangential distance of more than five meters means that no overshoot occurred before the end of the desired path.

Figure 22 shows a number of useful results. The first is that, just as with the "door" test results, the performance is rotationally symmetric about the Z-axis. It also indicates that the likelihood of overshooting is greatly influenced by the the lateral offset from the path – the less lateral offset the robot starts with, the more likely it is to overshoot the desired path while converging.

This experiment also yielded information on the set of initial conditions that are acceptable for how far the robot can go without overshooting a given path. An example of such a path is a smart wheelchair pulling up next to a handicap door assist button. Similarly to the "door" test results, these results can be used to determine whether a path planner must replan a path from the robot's current position to the origin of the desired path or if the desired path can be commanded directly with no replanning. For example, the data collected in simulation implies that, given a lateral offset of one meter and an initial heading error of ninety degrees, overshooting will occur; in that situation, it would not be safe to directly command the desired path if the desired path required no overshooting. A number of runs were conducted with HARLIE to validate these results; these tests with the physical system confirmed the simulation results. See Figure 23 for some examples of HARLIE while conducting these tests. One particular initial condition (see Figure 23a) as well as HARLIE successfully stopping parallel to a card reader that would unlock the door (see Figure 23b) are shown.

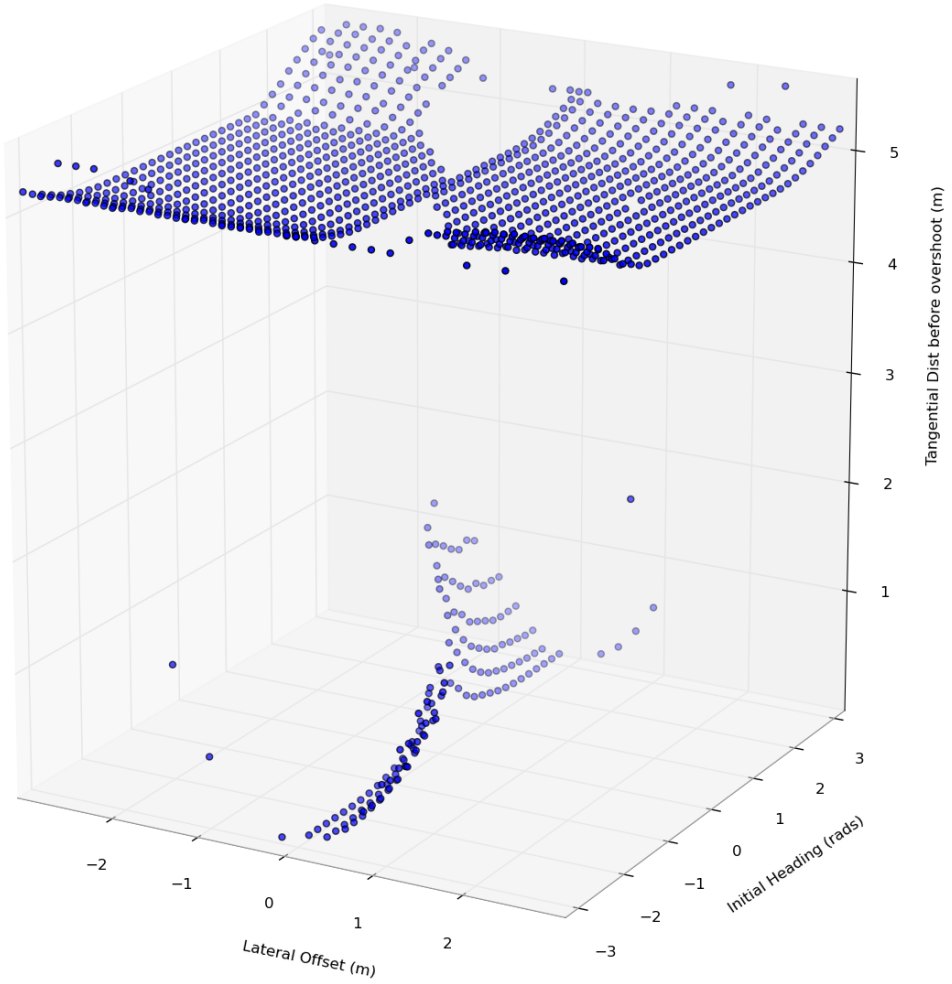
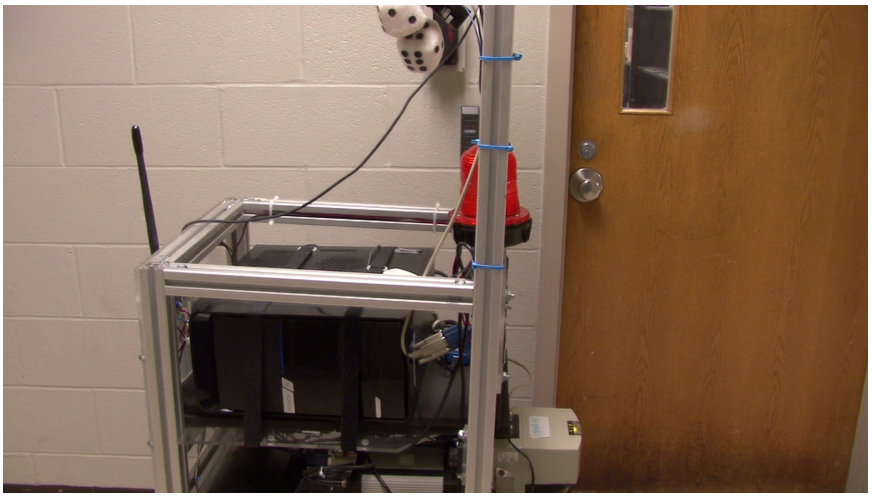



Figure 22: "Tangent" Test Simulation Results. The X-axis is the Initial Heading Error (rads), the Y-axis is the Initial Lateral Offset Error (m) and the Z-axis is the Tangential Distance before Overshoot (m).

(a) Initial Condition



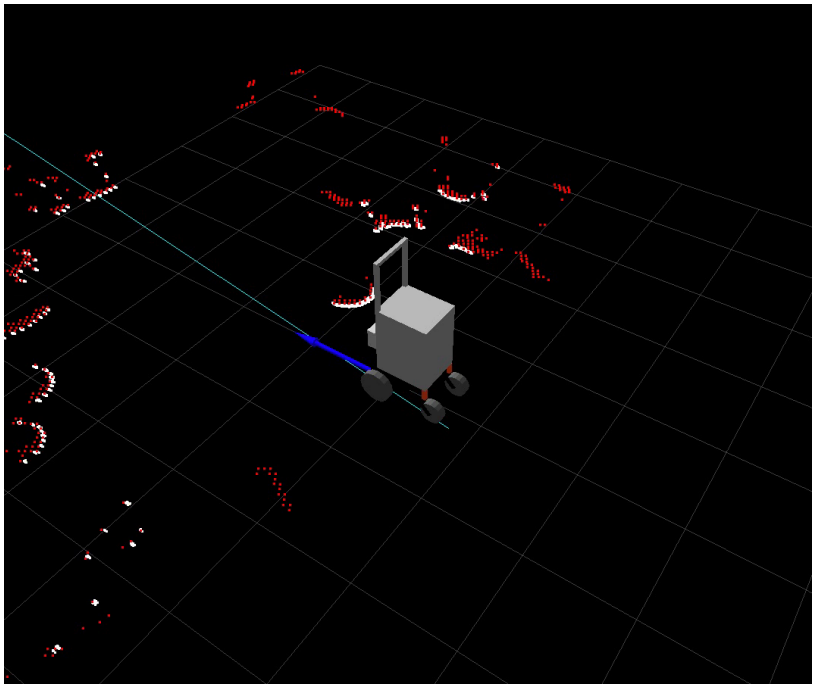
(b) Stopping Parallel to Card Reader

Figure 23: "Tangent" Test HARLIE Results. These are images captured from videos taken during the tests. The desired path runs parallel to the wall.

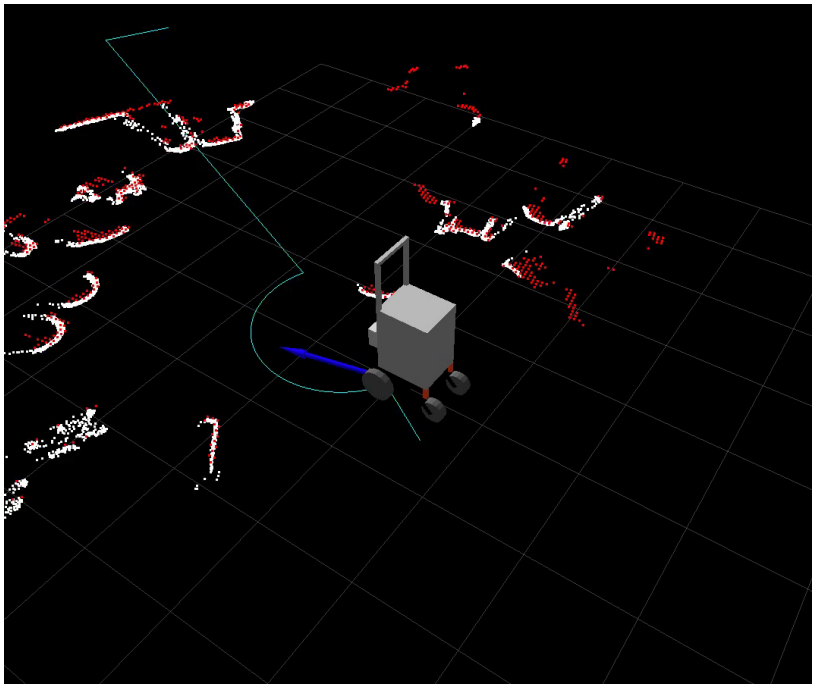
## 8.3 Splicing Tests

Experiments were also conducted to validate the splicing method described in Section 7. In this test, HARLIE was given a path where the first segment was a long straight line. This segment was then blocked with an orange construction barrel in order to force the planner to splice in a path around it. An example from one of these tests is in Figure 24. Figure 24a shows HARLIE while waiting for the planner to splice a path around the barrel; the barrel is the semicircle in the laser scan and obstacle map immediately in front and to the right of HARLIE. Figure 24b shows HARLIE after fifteen seconds of waiting for the barrel to move; the planner spliced in a sequence of segments as described in Section 7 and HARLIE began executing the newly spliced path segments. Figure 24c shows HARLIE after finishing the spliced segments and resuming the original path segment after successfully avoiding the barrel.

While these experiments validated that the splicing method described in Section 7 worked in some cases, they also confirmed some of the limitations of that splicing method. For example, in the first experiment the one meter radius of the spliced in semicircle was insufficient. After HARLIE started executing the spliced in path, the path was still going to result in a collision with the barrel. In order for HARLIE to complete the path in that experiment, the barrel had to be moved away from the path. This result led to increasing the radius of the spliced in semicircle to two meters for the results shown in Figure 24.

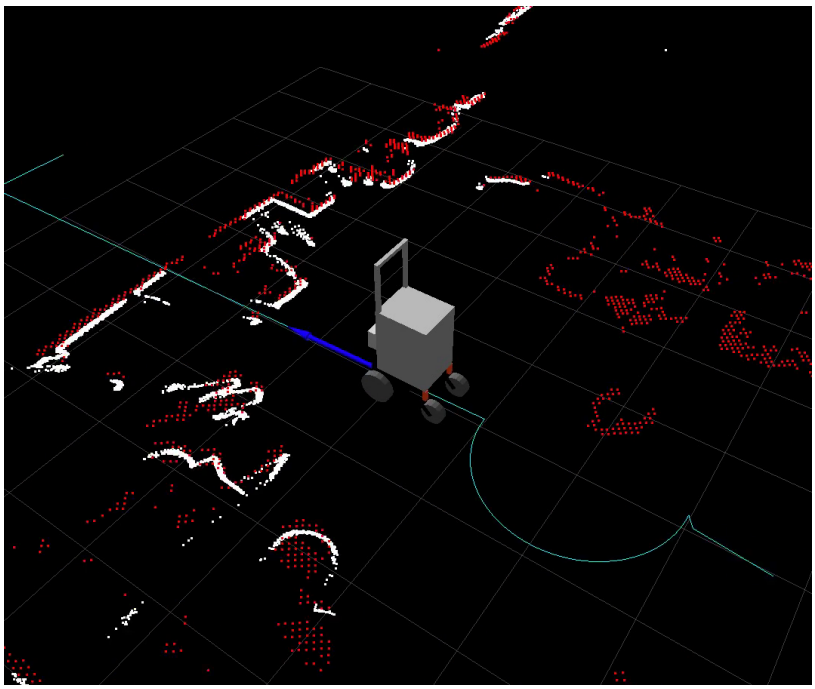


(a) HARLIE Stopped, Waiting for Obstacle to Move



(b) HARLIE Post-Splicing, On Spliced Arc

Figure 24: Splicing Test Results Figures (a) and (b)



(c) HARLIE Post-Splicing, Resuming Original Line Segment

Figure 24: Splicing Test Results Figure (c). These are images captured from videos of the GUI taken during the tests. Teal line is the desired path, blue arrow is the current desired state, red is the obstacle data from costmap3d, white is the current laser scan and the set of polygons is a model of HARLIE.

# 9 Future Work

There are a number of different avenues for building upon the precision navigation system described in this thesis. Some are described in more detail in this section.

One area for future work is in extensions to the precision localization subsystem described in Section 4. While the precision localization system was highly effective indoors with the very accurate laser scanner, *a priori* map and accurate relative localization system, there are cases where some of these components are unavailable. For example, on a robot that needs to navigate precisely in new buildings, an *a priori* map would not be readily available. SLAM algorithms could provide localization within unknown environments, but whether those localization estimates would be precise enough for the precision navigation system is unknown and would be an avenue for future work. Similarly, some robots, such as Otto, the Smart Wheelchair developed by Chad Rockey in [26], do not possess highly accurate laser scanners – analysis of how to utilize different, less accurate sensors for precision localization would be another area for future work. Currently, the precision localization system is limited to indoor environments only; more work would be required to extend the precision localization system to work in an outdoors environment.

Another area for future work would be more work characterizing the performance of the steering algorithms as well as trying new steering algorithms. A detailed analysis of the steering algorithms would extend the results presented in Section 8.2. Such an analysis could include things such as derivation and testing of equations that would allow a user to predict tangential distance to path convergence or tangential distance before overshoot, given the robot's initial conditions and gains. With the ability to predict those results for different gain values, it would become much easier to tune the steering algorithms to the desired behavior for a particular environment.

A different extension to the steering and trajectory generation that could be useful would be changing the path segment description from the simple geometric parame-

terization currently in use to a spline-based representation. Splines would allow each path segment to be more complex than a simple straight line, constant curvature arc or spin-in-place; splines could also be easier to fit to a sequence of points, such as those output by common planning algorithms or points learned by driving a desired path to "teach" the robot, than the current path segment representation.

Future work could also be devoted towards improving the simplistic path planner presented in Section 7. The planner presented there was rather limited, especially for use in a new environment where there are no *a priori* paths. Such a path planner could be built on some of the different dynamic planning algorithms discussed as alternatives to navfn in Section 3.3. Such an extension was explored by Bill Kulp in [15], utilizing the SBPL library [18] to generate paths that were converted to the format described in Section 6.1.

As part of this thesis, numerous avenues for integration with a "Smart" Building were discovered. A Smart Building is one that would be able to send information such as maps, goal locations with annotations and paths to the precision navigation system. One area where Smart Building integration would be important is in the specification of complex paths that would be very difficult for any autonomous path planner to determine. One example path that would be a good candidate for including in a Smart Building is the sequence of steps required for a power wheelchair user to enter the Case Western Reserve University Glennan building after hours or on weekends. In order to get into the building, a power wheelchair user must first drive up a handicap access ramp, then drive up to a card reader that unlocks the door with an ID card, then back up to the handicap power assist button to open the door, and then wait until the door opens before attempting to proceed through. These steps are further complicated by the fact that, after getting to the top of the handicap access ramp and driving to the card reader or door, there is a set of stairs that the wheelchair user could fall down if they steer slightly off to one side. Getting this sequence correct would be

tricky for a path planner with no *a priori* knowledge of the specific steps required for the Glennan building; the sequence is trivial if pre-planned and the building could download the pre-planned paths to an approaching robotic wheelchair.

# 10 Conclusion

This thesis has presented a precision navigation system for use on an indoor mobile robot. Using a planar laser scanner, wheel encoders, gyroscope and *a priori* map, a precision localization system was developed in order to generate quick, precise pose estimates for the mobile robot. Using these precise pose estimates, a precision path execution system made up of a steering algorithm, a trajectory generator and a simplistic path planner was developed. This system allowed HARLIE to precisely navigate indoors, including passing through doorways with less than seven centimeters of clearance on either side or pulling up parallel to a door reader. While the path planner was simplistic, it included the ability to splice in a fixed sequence of path segments to avoid unexpected obstacles.

The precision navigation system presented in this thesis also has a number of opportunities for further work. Some possibilities include allowing the precision localization system to operate without an *a priori* map or in an outdoor environment, a more rigorous analysis of the properties of the steering algorithms, extending the path segment description to include splines or improving the simplistic path planner. Some areas of possible integration with a "Smart" Building were also discussed, such as providing maps or complex pre-planned paths to a mobile robot entering the building for the first time.

# References


[1] *actionlib ROS Package Documentation*. URL: http://www.ros.org/wiki/actionlib (visited on 07/23/2012).

[2] SICK AG. *LMS291-S05 Datasheet*. July 2011. URL: https://www.mysick.com/PDF/Create.aspx?ProductID=33771&Culture=en-US.

[3] *AMCL ROS Package Documentation*. URL: http://www.ros.org/wiki/amcl (visited on 07/23/2012).

[4] *base_local_planner ROS Package Documentation*. URL: http://www.ros.org/wiki/base_local_planner (visited on 07/23/2012).

[5] *costmap_2d ROS Package Documentation*. URL: http://www.ros.org/wiki/costmap_2d (visited on 07/23/2012).

[6] Frank Dellaert et al. "Monte Carlo Localization for Mobile Robots". In: *IEEE International Conference on Robotics and Automation (ICRA99)*. May 1999.

[7] Analog Devices. *ADXRS150*. Mar. 2004. URL: http://www.analog.com/static/imported-files/data_sheets/ADXRS150.pdf.

[8] David Ferguson and Anthony (Tony) Stentz. "Field D*: An Interpolation-based Path Planner and Replanner". In: *Proceedings of the International Symposium on Robotics Research (ISRR)*. Oct. 2005.

[9] Jesse Fish. "Robotic Tour Guide Platform". MS Thesis. Case Western Reserve University, 2012.

[10] Dieter Fox. "Adapting the Sample Size in Particle Filters Through KLD-Sampling". In: *The International Journal of Robotics Research* 22.12 (2003), pp. 985–1003.

[11] Brian P. Gerkey and Kurt Konolige. "Planning and Control in Unstructured Terrain". In: *ICRA Workshop on Path Planning on Costmaps*. 2008.

[12] Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard. "Improved techniques for grid mapping with rao-blackwellized particle filters". In: *IEEE Transactions on Robotics* 23 (2007), p. 2007.

[13] Jason W. Harper. "Fast Template Matching For Vision-Based Localization". MS Thesis. Case Western Reserve University, 2009. URL: `http://rave.ohiolink.edu/etdc/view?acc_num=case1238689057`.

[14] Nathan Koenig and Andrew Howard. "Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator". In: *In IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2004, pp. 2149–2154.

[15] William R. Kulp. "Robotic Person-Following in Cluttered Environments". MA thesis. Case Western Reserve University, 2012.

[16] T. D. Larsen et al. "Design of Kalman filters for mobile robots; evaluation of the kinematic and odometric approach". In: *Proc. IEEE Int Control Applications Conf.* Vol. 2. 1999, pp. 1021–1026.

[17] S. M. LaValle. *Planning Algorithms*. Available at http://planning.cs.uiuc.edu/. Cambridge, U.K.: Cambridge University Press, 2006.

[18] Maxim Likhachev. *Search-based Planning with Motion Primitives*. Carnegie Mellon University. Oct. 2010. URL: `http://www.cs.cmu.edu/~maxim/files/tutorials/robschooltutorial_oct10.pdf`.

[19] Maxim Likhachev et al. "Anytime Dynamic A*: An Anytime, Replanning Algorithm". In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. June 2005.

[20] Maxim Likhachev et al. "Anytime search in dynamic graphs". In: *Artif. Intell.* 172.14 (2008), pp. 1613–1643.

[21] Eitan Marder-Eppstein et al. "The Office Marathon: Robust Navigation in an Indoor Office Environment". In: *International Conference on Robotics and Automation*. 2010.

[22] Hans Moravec and A. E. Elfes. "High Resolution Maps from Wide Angle Sonar". In: *Proceedings of the 1985 IEEE International Conference on Roboticsand Automation*. Mar. 1985, pp. 116 –121.

[23] *navfn ROS Package Documentation*. URL: http://www.ros.org/wiki/navfn (visited on 07/23/2012).

[24] *NI cRIO-9074*. National Instruments. URL: http://sine.ni.com/nips/cds/view/p/lang/en/nid/203964 (visited on 07/23/2012).

[25] *PID Controller*. URL: http://en.wikipedia.org/wiki/PID_controller (visited on 07/23/2012).

[26] Chad A. Rockey. "Low-Cost Sensor Package for Smart Wheelchair Obstacle Avoidance". MS Thesis. Case Western Reserve University, 2012.

[27] R. Siegwart and I. R. Nourbakhsh. *Introduction to Autonomous Mobile Robots*. Intelligent Robotics and Autonomous Agents. Mit Press, 2004. ISBN: 9780262195027. URL: http://books.google.com/books?id=gUbQ9_weg88C.

[28] *Sigmoid Function*. URL: http://en.wikipedia.org/wiki/Sigmoid_function (visited on 07/23/2012).

[29] Michael Tandy. *LIDAR-scanned-SICK-LMS-animation.gif*. Mar. 2008. URL: http://en.wikipedia.org/wiki/File:LIDAR-scanned-SICK-LMS-animation.gif (visited on 07/23/2012).

[30] *tf ROS Package Documentation*. URL: http://www.ros.org/wiki/tf (visited on 07/23/2012).

[31] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics*. MIT Press, 2006.

[32] Greg Welch and Gary Bishop. *An Introduction to the Kalman Filter*. 1995.

[33] K. M. Wurm et al. "OctoMap: A Probabilistic, Flexible, and Compact 3D Map Representation for Robotic Systems". In: *Proc. of the ICRA 2010 Workshop on Best Practice in 3D Perception and Modeling for Mobile Manipulation*. Software available at `http://octomap.sf.net/`. Anchorage, AK, USA, May 2010. URL: `http://octomap.sf.net/`.

## To do...

- [ ] 1. (p. 57) Wyatt did I get the 4 actual unknowns right? It's been soooooo long...