ROBUST AND RELIABLE REAL-TIME ADAPTIVE MOTION PLANNING

by

Sterling McLeod

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Computing and Information Systems

Charlotte

2019

ABSTRACT

STERLING MCLEOD. Robust and Reliable Real-time Adaptive Motion Planning. (Under the direction of DR. JING XIAO)

A key goal in robotics is to enable autonomous motion for mobile robots in a real-world dynamic environment with unknown obstacles. This problem requires bringing together state-of-the-art algorithms in path planning, robot control, and perception. An additional challenge is that systems that implement these algorithms must be carefully implemented with sophisticated software techniques.

This dissertation addresses the problem by expanding the Real-time Adaptive Motion Planning (RAMP) framework to enable real-time generation and execution of non-holonomic robot motion based on real-time perception of unknown dynamic obstacles in the presence of sensing and robot motion uncertainties. This dissertation further addresses how to systematically and rigorously test the implemented, integrated RAMP system to ensure robustness and reliability using state-of-the-art techniques in software testing.

ACKNOWLEDGMENTS

First and foremost, I need to thank my advisor, Dr. Jing Xiao. Dr. Xiao took me in as an undergraduate student that knew nothing about robotics, and through many years of guidance, support, and patience she helped me become someone that can fill a dissertation with state-of-the-art solutions to robotics problems.

I would like to thank Dr. Akella, Dr. Cukic, Dr. Shin, and Dr. Conrad for serving on my committee and providing helpful feedback along the way.

I am grateful to Graduate Assistance in Areas of National Need (GAANN) fellowship for funding my research for several years of my PhD, and for providing valuable pedagogical training during that time. I am also grateful for the support from the NSF Industry University Cooperative Research Center (I/UCRC) Robots and Sensors for the Human Well-Being (ROSEHUB) and from Shanghai mRobot via ROSEHUB.

I want to thank Mahmoud Abdelgawad and Dr. Anneliese Andrews for our collaborations on software testing for RAMP.

Thank you to all of my labmates, current and former, for all of the encouragement, conversations, advice, and fun over all the years spent in the lab.

I'm thankful to my family for their support throughout the PhD program, and for their efforts to adapt to the changes in my life that doing a PhD brought on.

Finally, I have to thank my fiancée, Samantha, for unwavering emotional support since day 1 of my PhD.

TABLE OF CONTENTS

LIST OF TABLES viii

LIST OF FIGURES x

CHAPTER 1: INTRODUCTION 1

1.1. Collision-free Motion Planning 2

1.2. Motion Planning Algorithms 4

1.3. Real-time Motion Planning 6

1.4. Non-holonomic Motion Planning 11

1.5. Localization, Mapping, and Simultaneous Localization and Mapping 13

1.6. Real-time Detection and Tracking of a Moving Object 14

1.7. Software Testing 16

1.8. Software Testing for Autonomous Robotics 18

1.9. Model-based Testing 19

CHAPTER 2: MOTIVATION AND OBJECTIVES 21

CHAPTER 3: REAL-TIME ADAPTIVE NON-HOLONOMIC MOTION PLANNING IN DYNAMIC ENVIRONMENTS WITH UNKNOWN OBSTACLES 24

3.1. Real-time Adaptive Non-holonomic Motion Planning in Unforeseen Dynamic Environments 24

3.1.1. Hybrid Trajectory Representation 24

3.1.2. Overview 28

3.1.3. Smooth Trajectory Transition and Adaptive Control Cycles 31

3.1.4. Trajectory Evaluation 34

3.1.5. Improvement of Trajectories Based on Sensing 35

3.2. Real-time Sensing and Perception of Unknown Obstacles 37

3.2.1. Overview 39

3.2.2. Algorithm for Real-Time Perception 40

3.3. Implementation, Results, and Discussion 42

3.3.1. Implementation and Experimental Setup 42

3.3.2. Visualization of Experiments 43

3.3.3. Records of Cycle Periods 44

3.3.4. Description of Test Cases and Results 46

3.3.5. Discussion of Performance 51

CHAPTER 4: LEVERAGING PAST EXPERIENCE IN RAMP 55

4.1. Review of Hilbert maps 56

4.2. RAMP Using Learned Information 57

4.2.1. Combining Real Sensor Data with Learned Data 58

4.2.2. Trajectory Evaluation Using Real and Learned Information 59

4.2.3. Initialization Using Learned Information 60

4.3. Experimental Results 60

4.3.1. Improving Initial Population 60

4.3.2. Real-time Execution with Learned Information 65

CHAPTER 5: SYSTEM-LEVEL TESTING OF A REAL-TIME ADAPTIVE MOTION PLANNING SYSTEM 73

5.1. Model-based Testing Approach 73

5.2. System-level Testing for RAMP 75

5.2.1. Interacton with Obstacles 75

5.2.2. Phase 1: Obstacle Test Model 76

5.2.3. Phase 2: Obstacle Internal Test Paths 78

5.2.4. Phase 3: Obstacle Interaction Test Paths 78

5.2.5. Phase 4: Obstacle Input-Space Partitioning 81

5.3. Test Execution and Evaluation 82

5.4. Extended System Level Behavioral Model 85

5.4.1. Obstacle Internal Test Paths 87

5.4.2. Obstacle Interaction Test Paths 87

5.4.3. Test Execution 88

5.4.4. Test Execution Results 89

CHAPTER 6: CONCLUSIONS AND FUTURE WORK 91

6.1. Contributions 92

6.2. Future Work 94

6.2.1. Learning 94

6.2.2. Testing 95

6.2.3. Implementation 96

6.3. Applications 97

REFERENCES 98

LIST OF TABLES

TABLE 1: Performance data for five runs of case 1 50

TABLE 2: Performance data for five runs of case 2 51

TABLE 3: Performance data for five runs of case 3 51

TABLE 4: Performance data for five runs of case 4 52

TABLE 5: Performance data for real-robot tests with two dynamic obstacles (Section 4.3.1.3). Two sets of tests were run: one that did not use any learned information, and one that improved trajectory initialization using learned information (Section 4.2.3). For each set, 5 tests were run. 67

TABLE 6: Performance data for tests that combine real and probabilistic data during runtime in the environment shown in Fig. 26(a). Two sets were run: one that did not use any learned information, and one that used probabilistic data to improve trajectory initialization and real-time execution. For each set, 5 tests were run. 69

TABLE 7: Performance data for tests that combine real and probabilistic data during runtime in the environment shown in Fig. 26(c). Two sets were run: one that did not use any learned information, and one that used probabilistic data to improve the initialization and real-time execution. For each set, 5 tests were run. 69

TABLE 8: Performance data for real tests that combine real and probabilistic data during runtime in the environment shown in Fig. 26(e). Two sets were run: one that did not use any learned information, and one that used probabilistic data to improve the initialization and real-time execution. For each set, 5 tests were run. 70

TABLE 9: Performance data for real tests that combine real and probabilistic data during runtime in the environment shown in Fig. 27(a). Two sets were run: one that did not use any learned information, and one that used probabilistic data to improve the initialization and real-time execution. For each set, 5 tests were run. 71

TABLE 10: Input domains and blocks of values of a mobile obstacle 81

TABLE 11: For each ABTC, 100 test cases were executed. Each column shows the results for the specific ABTC. 83

TABLE 12: Input domains and blocks of values of a dynamic obstacle 86

TABLE 13: Results for the first 8 ABTCs. For each ABTC, 100 test cases were executed. Each column shows the results for the specific ABTC. 90

TABLE 14: Results for the second 8 ABTCs. For each ABTC, 100 test cases were executed. Each column shows the results for the specific ABTC. 90

LIST OF FIGURES

FIGURE 1: Three common robot models 3

FIGURE 2: Roadmap examples 4

FIGURE 3: An example of a probabilistic roadmap build in a 2D C-space that can be used for path planning queries 5

FIGURE 4: An example of RRT exploring a C-space with circle obstacles. The red line highlights the path to the goal. 5

FIGURE 5: A visualization of an artificial potential function in a 2D space. The direction and size of the arrows show how the force will move the robot at each location. 7

FIGURE 6: A visualization of a population in RAMP. Many trajectories give the robot options on how to move to the goal. If the current trajectory becomes blocked, the robot can easily switch to a different trajectory. 9

FIGURE 7: A non-holonomic vehicle is shown. Its velocity is constrained to be in the direction of the back wheel angle θ. 11

FIGURE 8: An occupancy grid, where the black shapes are obstacles, grey pixels are occupied, and white pixels are free space. 15

FIGURE 9: V-Model for software development 17

FIGURE 10: Overview of the Model-based Testing approach 19

FIGURE 11: Hybrid trajectory examples 25

FIGURE 12: A quadratic Bezier curve connecting two straight line segments 26

FIGURE 13: Switching trajectories 33

FIGURE 14: Example motion from start to finish 37

FIGURE 15: The sensing module receives an occupancy grid based on depth data and outputs a list of circles representing position and size, and predicted linear velocities for each obstacle. 39

FIGURE 16: Case 1: One dynamic obstacle (human) and two static obstacles. The Turtlebot 2 platform runs RAMP-H. 45

FIGURE 17: Case 2: Two dynamic obstacles come into view sequentially and two static obstacles. The Turtlebot 2 platform runs RAMP-H. 46

FIGURE 18: Case 3: Two dynamic obstacles moving in different directions and two static obstacles. The Turtlebot 2 platform runs RAMP-H. 47

FIGURE 19: Case 4: Two dynamic obstacles moving in the same direction and two static obstacles. The Turtlebot 2 platform runs RAMP-H. 48

FIGURE 20: Visualization of cycle frequencies 49

FIGURE 21: Applying the Hilbert map approach to an environment of size 5m x 3m with one static obstacle (lower right) and one dynamic obstacle that repeatedly moves through the environment. The value at each location is the probability of that location being occupied. 57

FIGURE 22: A Turtlebot 2 is positioned in front of a large static obstacle that occludes most of the environment. The visible locations have values corresponding to the real sensing data (0 or 255), and the pixels that the robot cannot sense are replaced with a value representing the probability of the location being occupied. Lower values correspond to higher probability. The pixel values change to correspond to real data when the robot can sense them (Figs. 22(b) and 22(c)). 59

FIGURE 23: Hilbert map of an environment with one dynamic obstacle moving with straight line motion, and the result of one instance of running 500 pre-planning cycles using learned information. 62

FIGURE 24: Hilbert map of an environment with one static obstacle and one dynamic obstacle moving in a 1m line, and the result of one instance of running 500 pre-planning cycles using learned information. 63

FIGURE 25: Hilbert map of an environment with two dynamic obstacles, and the result of one instance of running 500 pre-planning cycles using learned information. 65

FIGURE 26: Three 5m square environments for testing the effect of incorporating probabilistic information, and their associated occupancy grids generated by the Hilbert maps model. The static obstacles are removed when recording training data. 66

FIGURE 27: Real environment (3.5m square) for experiments combining real and probabilistic sensing data with one static obstacle and one moving obstacle. The robot's initial position is $(1.5, 0)$ and the goal is $(1.0, 3.5)$. 71

FIGURE 28: Model-based test generation process 74

FIGURE 29: Structural model for a dynamic obstacle 77

FIGURE 30: Robot (square) and obstacles (circles) in an example test area, and the obstacle behavioral test model on the right. 77

FIGURE 31: Illustration of ABTCs for system-level testing involving 3 obstacles. Obstacles are shown transitioning from a stopped state (S) to a moving state (M). The delays correspond to the number of state changes of the prior moving obstacles. The bottom obstacle is always the obstacle that begins first, i.e. its delay is 0. The 2nd and 3rd obstacles begin at their respective delays after the first obstacle. 80

FIGURE 32: Overall performance of a RAMP system dealing with three mobile obstacles 84

FIGURE 33: Obstacle behavioral model that accounts for speed changes 86

FIGURE 34: Overall performance of a RAMP system dealing with three dynamic obstacles 89

CHAPTER 1: INTRODUCTION

Enabling autonomous robots to work seamlessly in real-world environments has long been a goal in the research of robot motion planning. Significant progress can benefit many applications and industries, such as material-handling, janitorial services, self-driving cars, and package delivery. Despite decades of research, however, there are still challenges to achieving an autonomous robot in many real-world environments. Three key reasons are:

- Many real-world environments are unpredictable.
- Robots are subject to uncertainty from sensors and motors.
- Robots require sophisticated software to accomplish tasks autonomously.

Many real-world environments are unpredictable, e.g. air-ports, hospitals, and schools. These environments are mostly populated by humans and objects (including those that can be moved by humans), and humans can change their motion in whatever ways they want. Because of this, a robot must be able to detect unforeseen changes in an environment and adapt to those changes at real-time in order to prevent collision and accomplish tasks.

Additionally, a robot must account for uncertainty while planning and moving. Both sensors and motors introduce uncertainty into a robotic system. If uncertainty is ignored, it is possible that a robot can collide with obstacles in the environment because the perception about the environment is inaccurate.

A method to validate and verify robotic systems is necessary to ensure that robots acting in real-world environments will be safe and effective. However, real-time robotic software systems pose several challenges that make them hard to test: they are concurrent systems, their environments contain many unknowns, and the algorithms are often non-deterministic to tackle the unpredictable environments. These challenges must be addressed to effectively test robotic software systems.

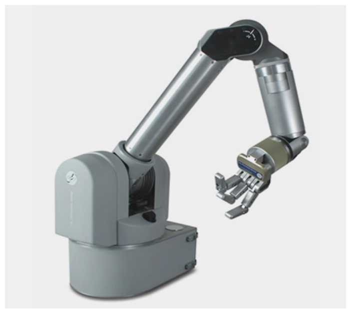
Addressing all of the above issues laid out above to enable autonomous robots requires building on a large amount of existing research. In the remainder of this chapter, I will survey the existing literature relevant to these research problems, including collision-free motion planning, motion planning algorithms, real-time motion planning, non-holonomic planning, localization and mapping, obstacle detection and tracking, software testing for autonomous robotics, and model-based testing.

1.1 Collision-free Motion Planning

A fundamental task in robotics is enabling a robot to move from an initial state to a goal state via a collision-free trajectory. This is achieved by planning a path or trajectory in the *configuration space*, or C-space, of a robot. Given the geometric information of a robot and its environment, the C-space of a robot is the set of all possible configurations that a robot may achieve.

A common type of mobile robot is a differential-drive robot (Figure 1(a)) These robots are driven by two wheels that have independent motors, and this type of motion

(a) Differential drive mobile robot


(b) Manipulator with seven degrees of freedom

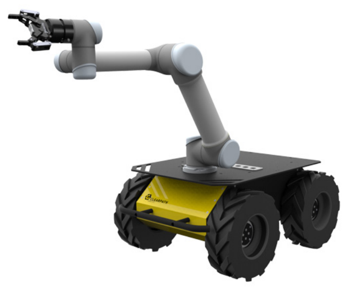

(c) Mobile manipulator that combines a mobile robot and a manipulator

Figure 1: Three common robot models

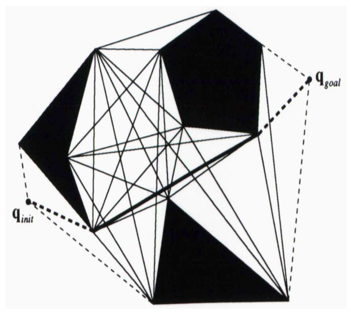
allows them to translate on a plane and change orientation. The C-space of such a robot is three-dimensional. Robot manipulators have more complicated geometry and typically have high-dimensional configuration spaces. The manipulator in Figure 1(b), for example, has seven joints, seven degrees of freedom, and therefore its C-space has seven dimensions. Mobile robots and manipulators can be combined as shown in Figure 1(c), and the C-space of such robots considers the dimensionality of both the mobile base and the manipulator.

Finding a path in a robot's C-space is the first step of motion planning. The next step is to consider the physical constraints of a robot, such as maximum acceleration, turning radius, etc. These constraints are considered when planning a trajectory to move the robot along a given path. The trajectory will generate full motion states (position, velocity, and acceleration) for the robot. A separate module for control is responsible for moving the robot along a trajectory.

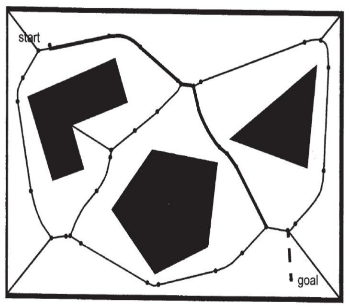
1.2 Motion Planning Algorithms

Motion planning algorithms attempt to find a collision-free path in the robot's C-space. There are many variations of motion planning algorithms, and many differences in the approaches. A very general categorization of motion planning algorithms is deterministic and non-deterministic algorithms.

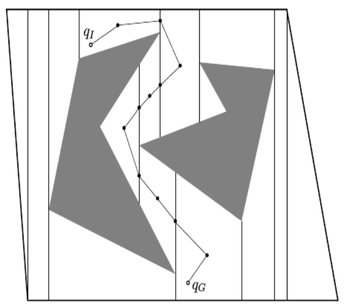
Deterministic methods refer to approaches that build a structure of the environment that does not rely on random sampling, such as a roadmap structure, and subsequently use graph-search algorithms to find a path [52]. These methods are also considered *complete* algorithms.



(a) Visibility graph connecting q_{init} and q_{goal}



(b) Voronoi graph connecting start and goal



(c) Cell decomposition connecting q_I and q_G

Figure 2: Roadmap examples

Roadmap methods include building visibility graphs [27], voronoi graphs [9], and cell decomposition [19]. The goal of these methods is roughly the same: build a roadmap of the environment, and then apply graph-search techniques to find the final path. They differ in the method used to build a roadmap, see Figure 2. The graph-search techniques can be Dijkstra's algorithm [29], A* algorithm, etc. These techniques can work well for low-dimensional and static environments, but they do

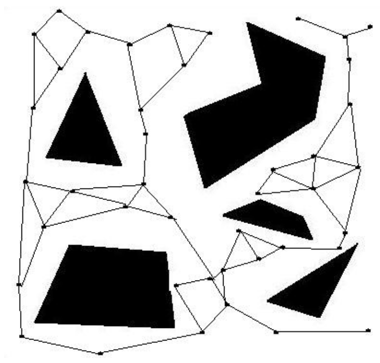


Figure 3: An example of a probabilistic roadmap build in a 2D C-space that can be used for path planning queries

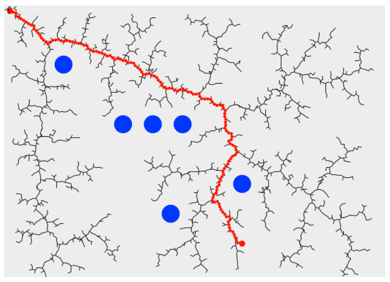


Figure 4: An example of RRT exploring a C-space with circle obstacles. The red line highlights the path to the goal.

not scale to higher-dimensional problems or working in dynamic environments.

The problem with deterministic methods is that they need to explicitly compute the robot's C-space. When a robot's C-space becomes high-dimensional, it is far too expensive to build a deterministic representation of the space. Computing the C-space requires the algorithm to compute the obstacle regions in the C-space, which scales exponentially with the number of dimensions.

Non-deterministic methods revolve around random sampling of a robot's C-space. These algorithms are effective at solving motion planning problems for robots that have a high-dimensional C-space, such as a 6-DOF manipulator. The Probabilistic Roadmap (PRM) algorithm [40] samples a C-space and builds an undirected graph of the sampled points. After building a graph, the initial and goal states can be added to the graph, and a graph-search algorithm can be used to find a path. The Rapidly-exploring Random Trees (RRT) algorithm [53] samples a robot's C-space and iteratively builds an n-ary tree until the goal state is connected to the tree. The initial state is the root of the tree, and the path from the tree's root to the goal state is used as the planned path. Figures 3 and 4 show examples of the structures used to represent the C-space using PRM and RRT.

Non-deterministic methods face issues with narrow passages in environments, and most variations on those approaches try to address this problem by improving the sampling method [3][92][12][96] or improving the edge connections [72][11][39]. PRM has been extended to dynamic environments by applying the algorithm to kinodynamic motion planning [36].

1.3 Real-time Motion Planning

Real-time Motion Planning refers to algorithms that operate in environments that can change. These algorithms need to adjust the path while the robot is moving in order to react to changes in an environment.

Non-deterministic algorithms have been extended to work in dynamic environments. Extended RRT (ERRT) [16] is an iteration on RRT aimed at dynamic environments. ERRT maintains a *waypoint cache* that stores the nodes of the path found by RRT. As the environment changes, RRT must re-plan a new path and starts by sampling from the waypoint cache to find nodes that are likely to lead to the goal. Although the waypoint cache speeds up path-planning queries, re-planning paths entirely can be too expensive to achieve robust real-time planning.

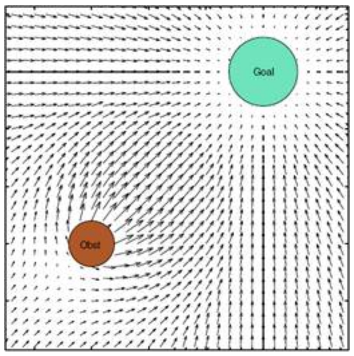


Figure 5: A visualization of an artificial potential function in a 2D space. The direction and size of the arrows show how the force will move the robot at each location.

Artificial potential functions (APF) [43] generate motion by defining attractive and repulsive functions in a robot's task space (for basic motion planning, this would be Euclidean space) to guide the robot to the goal. The robot is pulled towards the goal, and if the robot approaches obstacles, then the APF repels the robot away. Figure 5 shows a visualization of this. The APF approach is suitable for real-time motion planning because the functions are easy to update when the environment changes. However, potential functions are subject to local minima.

The Elastic Strip algorithm [15] is an approach to planning that combines local artificial potential functions and decomposition path planning [14] to move a high-dimensional robot in an unstructured dynamic environment while satisfying task constraints. The decomposition planning approach generates a path for the robot to follow, and as the environment changes, the robot's C-space path is modified by local potential functions to avoid obstacles while maintaining the path's homotopy. This framework was extended to Elastic Roadmaps [95] to utilize global planning that can find new homotopic paths as the robot moves. This framework works well for task-constrained motion, but maintaining many types of constraints can decrease the performance of obstacle avoidance. The experiments shown have limited obstacle avoidance needs. Additionally, experiments with this framework only used a holonomic mobile base, so it has not been shown to work with non-holonomic robots.

The D* algorithm [81][82] is a graph-search algorithm that handles changes in an environment by updating edge costs on-the-fly. As a robot moves around an environment, new information is obtained and changes to the edge costs are propagated throughout the graph. The D* algorithm, unlike many other real-time algorithms, is complete and optimal. D*-lite [45] has all the same properties as D* (complete, optimal, handles dynamic environments), but is a simpler algorithm. D* and D*-lite work well for low-dimensional spaces, such as 2 or 3 dimensions. However, they do not scale to higher dimensions well, and the performance is heavily affected by the resolution of the grid that represents the environment.

Real-time Adaptive Motion Planning (RAMP) [89] is a motion planning framework that plans and moves the robot simultaneously, and is designed to operate in environments that have unforeseen moving obstacles. It utilizes evolutionary computation for real-time trajectory modification [94] and evaluation to efficiently adapt to changes. RAMP maintains a set of trajectories (called a *population*) that are possible trajectories for the robot to move on. As the robot moves, the population is modified with simple operators that allow both small improvements and drastic alterations of paths. The RAMP framework has been shown to allow autonomous mobile manipulators to move in the presence of moving obstacles with unforeseen motion. However, it has only been shown to work in simulation, the mobile bases are holonomic [89], the

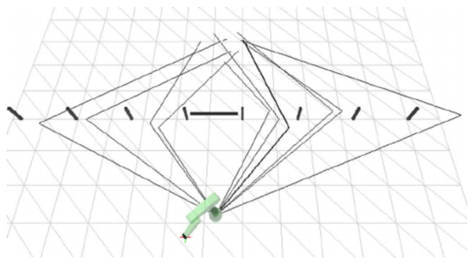


Figure 6: A visualization of a population in RAMP. Many trajectories give the robot options on how to move to the goal. If the current trajectory becomes blocked, the robot can easily switch to a different trajectory.

geometry of obstacles is known, and obstacle locations are broadcast to the planner. Thus, the existing work does not consider the uncertainty inherent in real robots, it does not consider mobile bases that are subject to non-holonomic constraints, and it does not consider using real-time sensing and perception to handle any unknown obstacles in the environment.

Human-aware motion planning models pedestrian behavior to perform well in human-centered environments. This work uses probabilistic models to more accurately predict pedestrian trajectories and improve planning decisions. Leveraging this knowledge helped to remove a robot from being ”frozen” when in dense environments [87] by modeling joint collision avoidance among multiple pedestrians. The joint collision prediction model was used to predict behavior between a robot and a human, and help a robot move in densely crowded pedestrian environments. Optimization approaches have also been explored for human-aware motion planning [42].

This type of planning can enable mobile robots to conform to social norms created by pedestrians. However, they do not generalize to dealing with arbitrary obstacles. Further, the work explored in [74] shows that behavior from human-human interactions cannot be transferred to robot-human interactions until robots are a normal presence in pedestrian environments.

Recent work in motion planning has been based around learning various aspects of the motion planning problems. Typically, deep learning and reinforcement learning techniques are utilized to learn obstacle behavior, collision avoidance, or an end-to-end model. Diffusion maps, a dimensionality-reduction machine learning technique, has been used in motion planning to reduce the space complexity of storing the shortest-path between N pairs of points from $O(|N|^2)$ to $O(N)$ by developing a parametrization of a 2D map's pairwise potentials [21]. This allows shortest-path information to be obtained and used for very large maps, and can be extended to work in real-time. Deep reinforcement learning [20] was used to learn a cost function to evaluate trajectories that respect social norms among pedestrians. Inverse reinforcement learning has been employed to learn collision avoidance behavior for mobile robots [46] based on demonstrated behavior with the goal of generating socially acceptable behavior from robots in such environments. End-to-end learning approaches use raw sensor data (such as images, laser scans, or point clouds) for training and learn control commands for mobile robots [73][83].

These methods have been shown to work well for mobile robots, but have not been considered for use in high-dimensional problems, such as manipulator or mobile-manipulator planning. Further, learning-based methods require large amounts of offline training (both in the time it takes to train and the number of data points needed to train), and after training the obstacle avoidance is tuned only for human obstacles and it is unclear how the deep learning results can be generalized to environments with different kinds of moving obstacles. These methods also suffer from a lack of

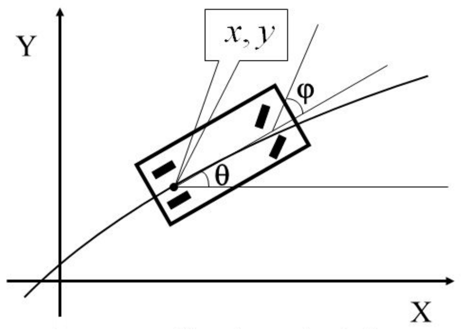


Figure 7: A non-holonomic vehicle is shown. Its velocity is constrained to be in the direction of the back wheel angle θ.

interpretability.

1.4 Non-holonomic Motion Planning

One of the most common constraints on a robot's motion are non-holonomic constraints. These are differential constraints that restrict the possible velocities a robot can obtain. Common non-holonomic robots are car-like robots and differential drive robots. A car-like robot model is shown in Figure 7, and its velocity is constrained with the following equation:

$$\dot{y}cos(\theta) - \dot{x}sin(\theta) = 0 \tag{1}$$

These constraints typically arise when the number of controllable inputs for a robot system is less than the number of degrees of freedom a robot has. For instance, a car-like robot has three degrees of freedom (x, y, θ), but only two controllable inputs (speed and turning angle).

Early work on this type of motion planning focused on control systems for the constraints and kinematic models of non-holonomic robots [49][50][10][71]. The algorithm proposed in [50] begins by planning a collision-free holonomic path. Then, it subdivides the path until each endpoint can be connected with a non-holonomic curve. Because this work is focused on defining the controllibility of non-holonomic systems, they do not consider dynamic obstacles or real-time planning. Later work approached the control problem with genetic algorithms [22][31][33], but they also leave out dynamic obstacles as a consideration.

When planning non-holonomic paths for a robot, it is common to utilize a specific type of curve to move the robot on. Reeds and Shepp defined optimal paths for a non-holonomic vehicle that can drive forwards and backwards [77]. These curves, commonly called Reeds and Shepp curves, assume that the robot's velocity is constant throughout the curve. Using polynomial spirals for non-holonomic planning was proposed in [41] by defining a parametric control system. The advantages of these curves are that 1) variable curvature throughout the curve can be obtained by using higher-order polynomials and 2) they can be planned quickly (usually within 1 millisecond). Bézier curves are another type of curve to satisfy non-holonomic constraints [55][23]. They are preferred in many cases because they can be analytically computed by a set of control points, which makes them even faster to compute than polynomial spirals.

The work in [55] updates non-holonomic motion at real-time based on new sensing information about the environment, but the environment does not contain dynamic obstacles. Using B-spline curves for optimized motion in aerial robots was shown in [37], but this work assumes that a higher-level planner provides a path, and it does not consider dynamic obstacles with unpredictable motion. A deformation approach to non-holonomic motion in dynamic environments was shown in [48], but this method requires an initial collision-free path and global information about an environment to perform deformation. Bézier curves have been used to generate motion around dynamic obstacles in [23], but the motion of the dynamic obstacles was known.

Nearly all real-world mobile robots are subject to non-holonomic constraints. Thus, generating motion that can satisfy such constraints is a critical piece of enabling real-world autonomous robots.

1.5 Localization, Mapping, and Simultaneous Localization and Mapping

Three important and related areas of mobile robot navigation are localization, mapping, and simultaneous localization and mapping (SLAM). Localization is the problem of estimating a robot's pose in an environment based on sensing information and typically a known map of the environment. Localization typically requires a map of the environment because dead reckoning (estimating position without using environment landmarks) becomes less and less accurate as the robot drives further and accumulates motion error. Mapping is the problem of establishing the representation of a robot's environment based on sensing and typically localization information. Mapping typically requires good localization because landmark positions can only be estimated from the robot's pose so error in the robot's pose will result in error in the map. Simultaneous Localization and Mapping (SLAM) is the problem of both estimating a robot's pose and building a map of a robot's environment simultaneously to produce a map of an unknown environment. Since neither a map or the robot's location is known, SLAM must perform each task without the aid of the other task, i.e., mapping without the aid of localization and localization without the aid of a map.

Each of these problems are essentially state estimation problems. For localization, typically the robot's pose is the state being estimated. For mapping, typically a set of landmark positions is the state, and the SLAM problems includes both a set of landmark positions and the robot's pose in the state.

Two common approaches to this are Kalman filters [38][78][85] [66] and particle filters [67][70][85] [34]. Kalman filters model the uncertainty by maintaining a Gaussian distribution to represent the *belief distribution*, which encodes the most likely state of the object and the uncertainty about the belief. Particle filters model uncertainty by maintaining a set of estimates that represents a belief distribution approximation, and updating that set as time goes on based on the uncertainty.

1.6 Real-time Detection and Tracking of a Moving Object

Detection and tracking of moving objects is a necessary task for autonomous robots. Avoiding unknown obstacles depends on a robot's ability to sense the environment, detect the obstacles, and track them to predict their future motion. Detection and Tracking of a Moving Object (DATMO) involves several aspects: 1) sensor selection, 2) object representation, and 3) accounting for sensing uncertainty.

Common sensors for obstacle avoidance are range-finding sensors, such as laser-based sensors or RGB-D cameras. Two-dimensional laser sensors perform a "scan" on a single plane in a cone shape, and return the depth that the laser reached for each direction that is scanned. LIDAR sensors are 3D laser-based sensors that return points clouds instead of only a single scan. RGB-D cameras emit infra-red patterns

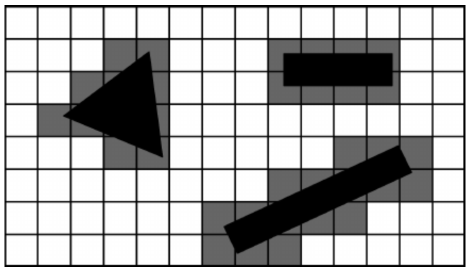


Figure 8: An occupancy grid, where the black shapes are obstacles, grey pixels are occupied, and white pixels are free space.

to detect the distance of objects in view [97]. The maximum distance for laser-based sensors is in the magnitude of hundreds of meters, whereas the maximum distance for RGB-D cameras is roughly 5 meters.

Self-driving cars are typically equipped with a suite of laser-based sensors [86][88][69] that together allow for real-time obstacle detection and tracking. These sensors scan the road ahead and return point clouds for perception algorithms to act on. In off-road environments, the sensors are used to identify obstacle regions, which are regions that exceed a critical vertical height, and classify terrain. In urban environments, depth information is used to identify terrain, stop lights, static obstacles, and dynamic obstacles (pedestrians and other cars) [54]. Unfortunately, performing so many complicated tasks requires very expensive sensors. The authors in [54] claim that hundreds of thousands of US dollars were used to equip their car with sensors. Low-cost robots, such as the Turtlebot platform [25], typically use either 2D laser scanners or RGB-D cameras to navigate indoor environments.

Common features extracted from 2D depth data are lines, corners, and circles. Fast leg detection is done in [93] by extracting lines and circular arcs from laser data. Lines and L-segments were used in [66] to detect moving obstacles. An alternative to extracting features from range data is to build an occupancy grid, which represents obstacles as occupied cells in a grid. Detecting occupancy was proposed in [30] by building a grid where each cell represents the probability of a cell being occupied. This approach has been simplified for modern applications by setting each cell to be one of three discrete values to represent if a cell is occupied, unoccupied, or unknown [62] (Figure 8). An occupancy grid approach [58] is used as the default local mapping method for the ubiquitous ROS navigation stack [75][61].

Object tracking requires a method to deal with the inherent uncertainty present in sensors. This task requires a system to estimate the state of the object given sensor information. Since this problem is essentially a state estimation, approaches described in Section 1.5 can be used for object tracking.

Some approaches are specific to tracking common objects in robotics environments, such as humans or vehicles. Autonomous vehicle applications utilize LIDAR sensors for tracking obstacles when the sensor itself is moving [60][66]. Human-aware applications learn general human motion [90] or the cooperative behavior of humans [46]. One recent tracking approach focused on fusing LIDAR and cameras to detect and classify obstacles [68] so that only the pedestrians are tracked, and used clustering techniques for point cloud data to do the tracking [18].

1.7 Software Testing

Software testing is an integral part of the software development process. The well-known V-model of software development can be seen in Fig. 9. The right side of the model outlines the levels of testing needed to thoroughly test a System Under Test

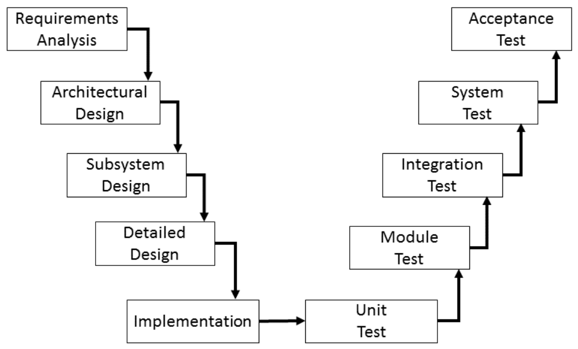


Figure 9: V-Model for software development

(SUT). Effective testing of a SUT involves creating a set of test cases (test suite) to run on the SUT for each testing level, identifying the expected output of these cases, running the system against the generated tests, and measuring the results of the test suite. Generating a test suite depends on several factors, such as the testing level, domain-specific knowledge, and maintaining a reasonable amount of test cases.

One of the biggest problems facing software test generation of any system is that the possible input to any computer program is nearly infinite. Consider that a 64-bit CPU can accept up to $2^{64} - 1$ different possible values, and that both the sequential and temporal order of values can be significant. However, most input is irrelevant for the system based on domain knowledge of the SUT. Test engineers must use domain knowledge to identify meaningful blocks of input and input behavior (sequential and/or temporal order) to a system in order to create an effective test suite.

A common method to reduce the number of tests in a test suite is by using test coverage criteria. test coverage criteria is a collection of rules that define requirements for a test set in order to adequately test a System Under Test (SUT) [4]. In other words, test coverage criteria define what input to use when testing software. The most effective coverage criteria for a system will depend on the model used to structure the software (graphs, data flow, logical expressions, etc.), the type of coverage desired (statement coverage, branch coverage, etc.), the various qualities of software (concurrent, synchronized, etc.), and the level of testing one wishes to perform on a SUT (unit, system-level, etc.).

1.8 Software Testing for Autonomous Robotics

The literature of testing autonomous robots and evaluating their performance mostly covers field trials/test arenas and computer-based simulation.

Field trials were performed to test autonomous robots [59][84][17]. While it is possible to generate many states during field trials, the lack of a systematic approach can lead to two limitations: 1) fewer scenarios being encountered and the testers hand-picking scenarios, and 2) not generating many significantly different types of scenarios, i.e., the test cases may not provide sufficient coverage of the possible scenarios.

Dynamics simulators [26][44] can produce high fidelity simulations for robotics. These simulators reproduce three-dimensional dynamic environments to re-create various scenarios that a robot may encounter. However, creating many different scenarios is still something that must be manually done by a user. Dynamic simulators themselves do not provide a practical method to enumerate many different test cases.

Laval et al. [51] present guidelines to robot system-level testing, and they recognize that tests should be repeatable, reusable, and automated as much as possible. However, the tests involved human intervention, which do not scale to systems with a large input space to cover with specific coverage-criteria.

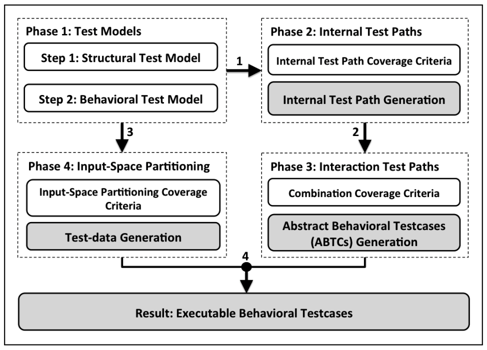

Phase 1: Test Models
Step 1: Structural Test Model
Step 2: Behavioral Test Model
1
Phase 2: Internal Test Paths
Internal Test Path Coverage Criteria
Internal Test Path Generation
3
2
Phase 4: Input-Space Partitioning
Input-Space Partitioning Coverage Criteria
Test-data Generation
Phase 3: Interaction Test Paths
Combination Coverage Criteria
Abstract Behavioral Testcases (ABTCs) Generation
4
Result: Executable Behavioral Testcases

Figure 10: Overview of the Model-based Testing approach

1.9 Model-based Testing

Model-Based Testing (MBT) uses various models to systematically generate tests [28], and has been well accepted as the state-of-the-art for system-level testing due to its potential for systematic and automatic test generation. Figure 10 shows an overview of how executable test cases are generated with an MBT approach. MBT includes three key elements: models that describe software behaviors (i.e., test models), coverage criteria that guide the test generation algorithms, and tools that generate supporting infrastructure for the test cases.

Guan and Offutt introduced a MBT technique for testing component integration in real-time embedded systems (RTES) [35]. However, they do not address concurrency, and they assume a static world.

MBT for dynamic environments was first proposed in [5] by using the Communicating Extended Finite State Machine (CEFSM) model [13] to model a set of discrete interactions (such as converse, listen, assist, greet, and guide) between a robot and a human, and to generate world states for system-level testing. Petri Nets were also used for this same application [7]. Coloured Petri Nets (CPNs) were used to build a test model for testing factory robots that carry a load from one place to another in [56]. While all of this work utilizes MBT for robotics applications in dynamic environments, none of them address the interactions between an autonomous mobile robot and unknown moving obstacles. Further, none of them provide test execution and validation of the technique.

CHAPTER 2: MOTIVATION AND OBJECTIVES

The previous chapter surveyed existing work for autonomous robots in dynamic environments, and highlighted that each approach has significant shortcomings, does not consider non-holonomic bases, and/or only exists in simulation without considering real-world uncertainties. Furthermore, nearly all methods are validated only by a handful of experiments, rather than being rigorously validated via systematic testing techniques.

This dissertation focuses on extending the Real-time Adaptive Motion Planning (RAMP) framework to address these issues by developing a RAMP framework for real non-holonomic robots that is robust to unknown obstacles, and verify its reliability by rigorous and automatic software testing.

This dissertation specifically addresses the following problem. Given a 2D planar environment, a rigid body mobile robot subject to non-holonomic constraints, an initial position, and a goal position, a real-time motion planner must generate motion that will move the robot from its initial position to the goal position in the environment while avoiding unknown obstacles that move in arbitrary ways.

The obstacles in the environment are not known beforehand to the robot. The size, shape, and velocity of the obstacles must be inferred by real-time perception based on real sensing data. The motion of the obstacles is not limited by speed or direction, and an obstacle's velocity may change at any time. The obstacles may enter or leave the environment at any time.

The environment considered in experiments is a small 2D environment. The size of the environment is small enough to allow dead reckoning for localization, rather than more sophisticated localization and mapping techniques.

The robot is modeled as a disc moving in the environment with configuration (x, y, θ), where θ is the yaw rotation of the robot. The robot's velocity is subject to the non-holonomic constraint:

$$\dot{x} sin(\theta) - \dot{y} cos(\theta) = 0 \tag{2}$$

The robot's motion is further limited by dynamic constraints which enforce maximum acceleration and velocity values on the robot.

The dissertation introduces approaches to address the following aspects of this problem:

- RAMP extended to planning non-holonomic motion in the presence of obstacles that move in unforeseen ways [63].
- RAMP extended to planning in the presence of unknown obstacles by incorporating real-time perception using real sensors [64].
- RAMP extended to leverage past experience when navigating in the presence of unknown obstacles [65].
- A software test generation framework to validate the effectiveness of an implemented RAMP system [1][2].

The following chapters detail these approaches extending RAMP and the validation of the approaches on real robot systems as well as the software testing framework applied to testing RAMP.

CHAPTER 3: REAL-TIME ADAPTIVE NON-HOLONOMIC MOTION PLANNING IN DYNAMIC ENVIRONMENTS WITH UNKNOWN OBSTACLES

This chapter introduces an approach to achieve real-time non-holonomic motion planning in the presence of unknown obstacles.

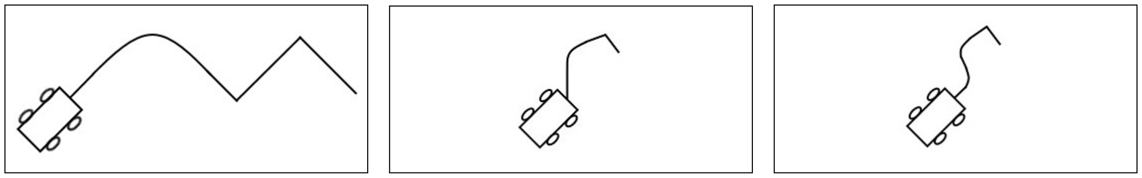
3.1 Real-time Adaptive Non-holonomic Motion Planning in Unforeseen Dynamic Environments

We introduce an approach for real-time adaptive non-holonomic motion planning in unforeseen dynamic environments that progressively converts holonomic segments to non-holonomic segments on the fly for execution by the robot. This real-time hybrid planner, called RAMP-H, enables smooth and adaptive non-holonomic motion of the robot towards its goal while avoiding dynamic obstacles of unforeseen motion in the environment.

3.1.1 Hybrid Trajectory Representation

A mobile robot can be either holonomic or non-holonomic. Its configuration can be expressed as (x, y, θ). Its state can be expressed as a point (x, y, θ, t) in its configuration-time (CT) space. A trajectory is represented as a set of states in the robot's CT space. A non-holonomic trajectory satisfies the non-holonomic constraint:

$$-\dot{x}\sin\theta + \dot{y}\cos\theta = 0 \tag{3}$$



(a) Full hybrid trajectory (b) Robot cannot execute the trajectory (c) A new curve is formed so that the robot can execute the trajectory.

Figure 11: Hybrid trajectory examples

which is necessary for a non-holonomic robot and is desirable for a holonomic robot (if it is not omnidirectional) to conduct smooth motion. A *non-holonomic segment* is any set of continuous points on a trajectory that satisfy equation (1). Such a segment may consist of curves and straight lines, as long as all points satisfy the non-holonomic constraint. Any segments that require the robot to stop and rotate in place are not non-holonomic segments.

Given a holonomic path consisting of straight-line segments (generated by the RAMP-H planner, detailed in Section III), the algorithm converts the first two straight-line segments to a non-holonomic segment by creating a Bezier curve to connect the two straight-line segments, as shown in Fig. 11. Velocity and acceleration values that satisfy the robot's motion constraints are generated for each point on the curve, as detailed below.

Each Bezier curve for connecting two straight-line segments is defined by three control points $(X_0, Y_0), (X_1, Y_1)$, and (X_2, Y_2), as shown in Fig. 12, and can be expressed in terms of a parameter u [55]:

$$\begin{aligned} x &= (1-u)^2 X_0 + 2u(1-u)X_1 + u^2 X_2 \\ y &= (1-u)^2 Y_0 + 2u(1-u)Y_1 + u^2 Y_2 \end{aligned} \tag{4}$$

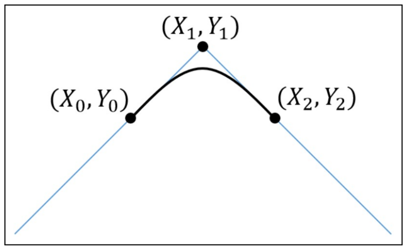


Figure 12: A quadratic Bezier curve connecting two straight line segments

The velocity and acceleration along the curve can be further derived as functions of $\dot{u}$ and $\ddot{u}$:

$$\begin{aligned}\dot{x} &= (Au + C)\dot{u} \\ \dot{y} &= (Bu + D)\dot{u}\end{aligned} \tag{5}$$

$$\begin{aligned}\ddot{x} &= (Au + C)\ddot{u} + A\dot{u}^2 \\ \ddot{y} &= (Bu + D)\ddot{u} + B\dot{u}^2\end{aligned} \tag{6}$$

where $A = 2(X_0 - 2X_1 + X_2)$, $B = 2(Y_0 - 2Y_1 + Y_2)$, $C = 2(X_1 - X_0)$, and $D = 2(Y_1 - Y_0)$. (X_0, Y_0) is chosen on the first straight-line segment. (X_1, Y_1) is the knot point connecting the two straight-line segment, and (X_2, Y_2) is chosen on the second straight-line segment, such that A, B, C, and D are not zero.

The Reflexxes Type II library [47] is used to generate smooth motion for a hybrid path. For each straight-line segment, we apply Reflexxes to generate the trajectory for x and then compute the corresponding trajectory for y as the linear function of x. For the Bezier curve, Reflexxes is used to generate u, $\dot{u}$, and $\ddot{u}$ values for a trajectory of u and those values are substituted into the above equations to obtain the corresponding non-holonomic trajectory.

There are two $\dot{u}$ values that are necessary to specify to generate a trajectory using

Reflexxes - the initial and maximum $\dot{u}$. The initial $\dot{u}$ can be found when u is zero:

$$\dot{u}_0 = \frac{\dot{x}_0}{C} = \frac{\dot{y}_0}{D} \tag{7}$$

where $\dot{x}_0$ and $\dot{y}_0$ are the velocities at the start of the curve (i.e., at the curve's first control point).

The maximum velocity $\dot{u}_{max}$ can be decided by setting $\ddot{u} = 0$. If a maximum of $\dot{u}$ exists, it can be obtained by:

$$\dot{u}_{max} = min(\sqrt{\frac{\ddot{x}_{max}}{A}}, \sqrt{\frac{\ddot{y}_{max}}{B}}) \tag{8}$$

where $\ddot{x}_{max}$ and $\ddot{y}_{max}$ are determined by the robot's velocity limits. Otherwise, the value of $\dot{u}_0$ is used for $\dot{u}_{max}$.

Note that if a Bezier curve makes too sharp of a turn, i.e., one with large curvature values, the robot may not be able to follow it. The maximum angular velocity on a curve occurs at the point u_r, which is the point of minimum radius R_{min} along the curve, as derived in [55]:

$$u_r = -\frac{AC + BD}{A^2 + B^2} \tag{9}$$

$$R_{min} = \sqrt{\frac{[(A^2 + B^2)u_r^2 + 2(AC + BD)u_r + (C^2 + D^2)]^3}{(BC - AD)^2}}. \tag{10}$$

For a given Bezier curve, we can use the values of u_r and $\dot{u}_{max}$ and equation (5) to make a conservative (i.e., worst-case) estimation of the robot's linear and angular velocities, v_r and ω_r respectively, required at u_r. If the robot's maximum angular velocity is not large enough, i.e. $\omega_{max} < \omega_r$, then the robot cannot follow the Bezier curve.

If a robot's starting orientation is not aligned with the beginning straight-line portion of the non-holonomic trajectory segment (converted from the first two straight-line segments of a holonomic path, as described above), as shown in Fig. 11(b), the robot cannot readily execute the non-holonomic trajectory segment. That often happens when trajectory switching is needed. In such a case, a straight-line segment aligned with the robot's starting orientation and a Bezier curve are added to form a non-holonomic *transition trajectory*, as shown in Fig. 11(c), to enable the robot smoothly get on the original non-holonomic segment.

The result is a hybrid trajectory, consisting of up to two Bezier curves followed by straight-line holonomic segments, allowing smooth transition of the robot from one straight-line segment to the next.

3.1.2 Overview

RAMP-H conducts planning and execution of motion simultaneously via planning, control, and sensing cycles. Planning cycles modify a set of trajectories, called a population, to find a better trajectory for the robot to move on. Unlike the other cycles, planning cycles are not set to occur at specified intervals. When one planning cycle finishes, a new one begins immediately. At each control cycle, the robot will switch to the best trajectory in the population. As the robot moves, changes in the environment are captured and updated in each sensing cycle and conveyed to the planner to facilitate adaptation.

Algorithm 1 outlines our method. Our planner starts from generating an initial population of holonomic trajectories that all start from the same initial state of the robot, and each trajectory consists of straight-line segments and self-rotations, where the knot configurations connecting the straight-line segments can be generated randomly. The procedure **convert** is then called to add a non-holonomic segment at the beginning of each trajectory to enable a smooth transition of the robot to the first straight-line segment of that trajectory and also convert the first two straight-line segments of the trajectory into a non-holonomic segment (as described in Section II). The result is a population of hybrid trajectories starting from two non-holonomic segments followed by holonomic segments. Section III-B provides more details of **convert**. Next, The procedure **evaluate** is called to evaluate and rank the fitness of the hybrid trajectories, as detailed in Section III.C. The best trajectory is obtained for the robot to move on.

During the first control cycle, the robot moves along the best trajectory, τ_1, as long as its portion within the first control cycle, $\tau_{1,c}$, is collision-free. Note that the duration of a control cycle Δt_c is set to ensure that $\tau_{1,c}$ is non-holonomic. While the robot moves along $\tau_{1,c}$, the planner is simultaneously improving the population of trajectories so that at the next control cycle, the robot can switch to a better trajectory if available. For that purpose, at the beginning of the current control cycle, the planner updates the starting states of all other trajectories (except for the currently being executed one) to be the state at the beginning of the next control cycle and converts the updated trajectories to be hybrid ones. Note that the conversion can change the starting time of the next control cycle because different hybrid trajectories require different starting times (Section III-B).

Algorithm 1 RAMP-H overview

```
1: Set control and sensing cycle time $\Delta t_c$ and $\Delta t_s$ respectively;
2: $i \leftarrow 1$; //index of control cycle
3: initialize a population $P_1$ of holonomic trajectories;
4: **convert** $P_1$ to hybrid trajectories $P_1^H$;
5: **evaluate** $P_1^H$ and obtain the best trajectory $\tau_1$
6: and the portion $\tau_{1,c}$ within the first $\Delta t_c$ of $\tau_1$;
7: $t_1 \leftarrow 0$; //beginning time of 1st control cycle
8: $t_2 \leftarrow \Delta t_c$; //beginning time of 2nd control cycle
9: $\tau_{best} \leftarrow \tau_1$;
10: **while** goal is not reached **do**
11:     simultaneously do Move, Plan, and Sense:
12:     **Move**:
13:     **if** $\tau_{i,c}$ is feasible **or** collision on $\tau_{i,c}$ is not expected
14:     within time period $t_{thresh}$ **then**
15:         move along $\tau_{best}$;
16:     **else**
17:         pause motion;
18:     **Plan**:
19:     **if** beginning of $i$th control cycle **then**
20:         $t_{i+1} \leftarrow t_i + \Delta t_c$;
21:         obtain $P_{i+1}$ by updating the starting state of $P_i$;
22:         **convert** $P_{i+1}$ to $P_{i+1}^H$ and find the earliest
23:         starting time $t_{i+1,e}$ in $P_{i+1}^h$;
24:         $t_{i+1} \leftarrow t_{i+1,e}$
25:     **adjust** $P_{i+1}$ and $P_{i+1}^H$;
26:     **evaluate** $P_{i+1}^H$;
27:     **modify** $P_{i+1}$;
28:     **if** end of $i$th control cycle **then** //reaching time $t_{i+1}$
29:         $i \leftarrow i + 1$;
30:         $\tau_{best} \leftarrow \tau_i$ //update best trajectory
31:     **Sense**:
32:     **if** new sensing cycle **then**
33:         **evaluate** $P_{i+1}^H$;
34:         collision-check $\tau_{i,c}$;
```

The planner improves the trajectories by calling **adjust**, **evaluate**, and **modify** repeatedly in multiple *planning cycles* (i.e., the **while** loop), detailed in Section III-C. During robot motion and planning, sensory updates of the environment are conducted every *sensing cycle*, and the trajectories are re-evaluated and re-ranked to reflect the changes in the environment. Re-evaluating is necessary because the previous best trajectory may no longer be the best after changes occur in the environment. If the best non-holonomic trajectory segment $\tau_{i,c}$ is not collision-free and the collision is expected to occur within time period, t_{thresh}, then the robot's motion will be paused to wait for the planner to find a collision-free trajectory through more planning cycles. The value of t_{thresh} is chosen such that the robot stops early enough to allow for a non-holonomic motion around a stopped obstacle. If the obstacle moves away from $\tau_{i,c}$ before t_{i+1}, then the robot will resume its motion along τ_{best}.

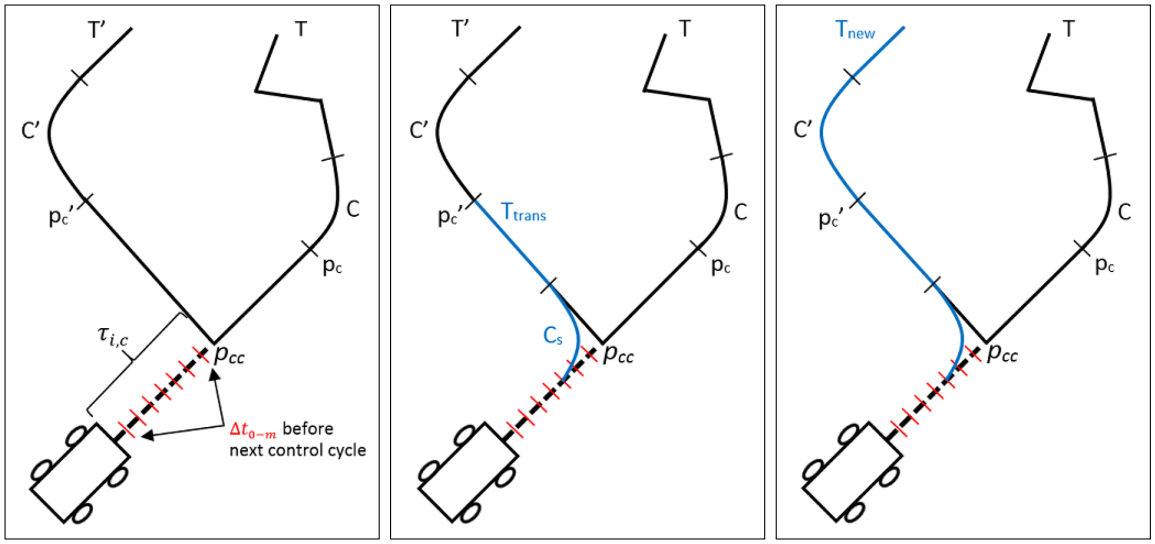
3.1.3 Smooth Trajectory Transition and Adaptive Control Cycles

Given a population P of holonomic trajectories, where each is in terms of knot configurations connecting straight-line segments, the procedure **convert** obtains a population of hybrid trajectories P^H from P by adding a non-holonomic transition trajectory segment, if necessary, followed by converting the first two straight-line segments to a non-holonomic trajectory segment, as described in Section II, for each trajectory in P. At each control cycle, after the starting states of trajectories are updated to be the states at the next control cycle, **convert** is called on the new population to prepare non-holonomic segments.

Note that a transition trajectory segment has to occur *before* the next control cycle in order for the robot to smoothly switch to the intended trajectory by the time of the next control cycle. Thus, a transition trajectory consists of a Bézier curve to connect the straight-line segment of the current trajectory before transition and the first straight-line segment of the new trajectory. The segment points (X_0, Y_0), (X_1, Y_1), and (X_2, Y_2) for a *transition curve* are the robot's current position, the future position at the next control cycle, and the first control point of the target trajectory's Bézier curve, respectively.

The control points of the transition curve for a trajectory, however, depend on the robot's kinematic constraints and the sharpness of the curve needed for transition. We divide the control cycle period Δt_c into m steps. Starting at the time of the mth step, our system assigns the (future) position of the robot at that time as the first control point and attempts to plan a kinematically feasible transition trajectory. If such a trajectory cannot be found, the position at the m-1th step is used. Our system continues this process until either it finds a kinematically feasible transition trajectory or the possible positions at all steps are exhausted so that no switching will be possible to that trajectory. Different trajectories are likely to have different first control points.

Fig. 13 shows an example. In Figure 13(a), the robot is moving on the best trajectory, T, at the current time and wants to compute a transition trajectory to switch to trajectory T'. The next control cycle is expected to occur in Δt_c seconds and is at point p_{cc}. The trajectories' curves between their first two segments are labelled as C and C'. The motion states at p_c and p'_c are the initial states of their respective curves.



(a) Robot wants to switch from T to T' (b) Transition trajectory shown in blue (c) Full switch shown in blue

Figure 13: Switching trajectories

The transition trajectory can be seen in 13(b) as the solid blue line. It consists of a Bézier curve, C_s, that begins at one of the future time steps and a straight-line segment that ends at the first control point of C'. Once the transition trajectory, T_{trans}, has been created, the curve from the target trajectory, C', and the remaining points on T' are concatenated onto T_{trans} to form T_{new}. In Figure 13(c), T_{new} is shown in blue.

It is unlikely that all trajectories in the transition population will begin at the same time. Therefore, the starting time of the next control cycle is set to be the earliest starting time among all transition trajectories in Algorithm 1, i.e., the frequency of control cycles is determined both by the need of non-holonomic conversion of trajectories for the robot to follow and the need for the robot to switch to a better trajectory. Thus, the interval of an actual control cycle is adaptive, unlike in the previous RAMP design [89].

3.1.4 Trajectory Evaluation

Procedure **evaluate** calls an evaluation function to measure the fitness of each hybrid trajectory so that trajectories can be ranked based on their fitness. The evaluation utilizes a cost function to determine fitness and the function depends on a trajectory's feasibility.

3.1.4.1 Feasible trajectories

Feasible trajectories are defined as those with no predicted collision in their non-holonomic portion and are kinematically feasible for the robot to move on. The cost evaluation function for a feasible trajectory is:

$$f = w_1 \frac{T}{\eta_1} + w_2 \frac{\Delta\theta}{\eta_2} + w_3 \left(\frac{D}{\eta_3}\right)^{-1} \tag{11}$$

where T is the execution time of a trajectory, $\Delta\theta$ is the orientation change needed to move on the trajectory, and D is the minimum distance to any obstacles along the trajectory, w_i is a weight, and η_i is a normalization value. Note that if there is no obstacle, the third term is zero.

3.1.4.2 Infeasible trajectories

Infeasible trajectories are defined as those with collision in the non-holonomic portion or are not kinematically feasible for the robot to move on. The cost equation is also a weighted sum:

$$g = \rho_T + \rho_\theta \tag{12}$$

$$\rho_T = \frac{\delta_T}{T_{coll}} \qquad \text{and} \qquad \rho_\theta = \delta_\theta \frac{\Delta\theta}{\eta_\theta} \tag{13}$$

where δ_T and δ_θ are large constant values, T_{coll} is the duration before the first predicted collision in the trajectory, $\Delta\theta$ is the orientation change needed to move on the trajectory, and η_θ is a normalization term.

The penalty ρ_T ensures that trajectories with collisions expected earlier will be penalized more than trajectories with collision expected later. ρ_θ ensures that trajectories with larger orientation change are penalized more. δ_T and δ_θ are set such that the penalty for a large orientation change will not be larger than for collision. No infeasible trajectory will ever be fitter than a feasible one.

Procedure **evaluate** only checks for collision in the non-holonomic segments of a trajectory to take into account the fact that the environment may change in unpredictable ways, trajectories are constantly evolved, and thus non-holonomic conversion occurs later for later segments and will be checked for collision then. This saves a lot of time. Collision checking is conducted between the robot's non-holonomic trajectory segment and the predicted trajectory segment for the same time period of every obstacle.

3.1.5 Improvement of Trajectories Based on Sensing

The RAMP-H planner improves hybrid trajectories through the following operations.

3.1.5.1 Adjust

At each control cycle, the new population P_{i+1} is obtained by updating the starting state of P_i. The updated state is the robot's intended future state at the next control cycle based on its current trajectory. However, this update assumes perfect motion of the robot. Due to motion error, the robot may not reach this state and may encounter unpredicted collision as a consequence. The **adjust** procedure is called in each planning cycle (i.e., the **while** loop in Algoritm 1) to address this issue.

The procedure **adjust** polls the robot's internal sensors to obtain the robot's current motion state. The difference between the robot's current position and the expected position along the current trajectory at the time of the planning cycle is added to the position of the state s_{cc} at the beginning of the next control cycle as an offset. The positions in all the trajectories of the population starting at s_{cc} are offset accordingly. Although the act of offsetting is quite simple, the significance of this method is that it relies on the interaction of planning and control cycles to treat specific time cycles as clear, unambiguous landmarks in time to perform the adjustment.

If the robot is stopped indefinitely by imminent collision, then the **adjust** procedure allows the planner to plan alternative trajectories for the robot from where the robot is stopped.

3.1.5.2 Modify

In each planning cycle, the planner also calls the procedure **modify** to hopefully improve the population of trajectories. In procedure **modify**, a subset of trajectories are randomly selected, and next the knot configurations (i.e., the holonomic paths) are altered by common genetic operators as used in the RAMP planner [89]: *Insert*, *Delete*, or *Change* a knot configuration, *Swap* the order of two knot configurations, and *Crossover* two trajectories. New trajectories are converted again into hybrid

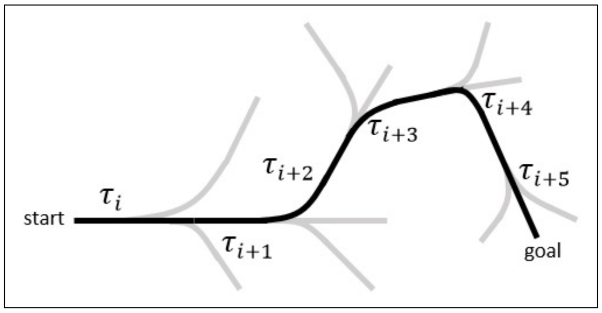


Figure 14: Example motion from start to finish

ones (if necessary) and evaluated. They are then used to replace some randomly selected trajectories except for the best trajectory in the previous population. However, an infeasible trajectory will not replace a feasible one and a trajectory will not be added if its fitness is less than the minimum fitness in the population. This approach mixes elitism and diversity in evolving a trajectory population over generations (i.e., planning cycles).

Note that our planner continuously improves trajectories through planning cycles while interacting with sensing cycles to adapt trajectories to changes in the environment. The planner also interacts with control cycles to allow the robot to switch to trajectories best adapt to the environment. Therefore, the actual motion that the robot executes is most likely a concatenation of non-holonomic segments from different hybrid trajectories along the way, as shown in Fig. 14.

3.2 Real-time Sensing and Perception of Unknown Obstacles

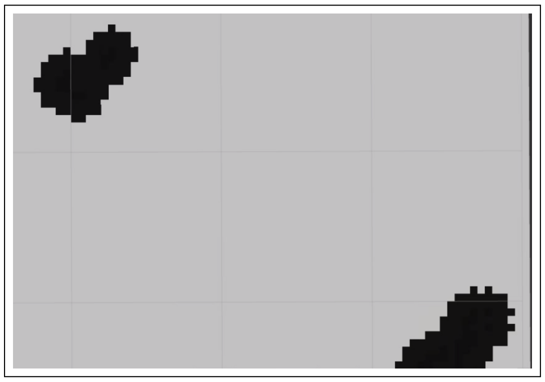
In order for a RAMP planner to effectively adapt to unknown or unpredictable obstacles in real environments, it must use real sensors to detect and track obstacles at real-time. In this section, we describe how to achieve real-time perception using depth data in 2D environments.

Algorithm 2 Real-time perception

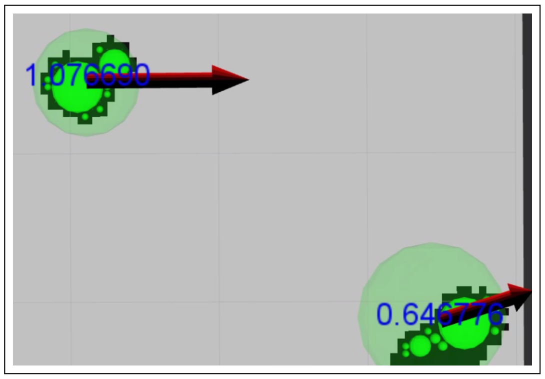
```
1: $H_0 \leftarrow$ most recent depth data
2: $I_0 \leftarrow$ binary occupancy grid based on $H_0$
3: $I_0 \leftarrow I_0|I_1|...|I_N$ // ameliorate noise
4: $N \leftarrow$ contour_points($I_0$) // vertices of a polygon for each connected component
5: $O_i \leftarrow \emptyset$ // initialize set of obstacles in most recent grid
6: for each polygon $n_i \in N$ do
7:     $c_i \leftarrow$ bounding circle for pixels in $n_i$
8:     $\mathbf{s}_i \leftarrow$ packed circles into $n_i$
9:     $o_i \leftarrow$ new obstacle created from $c_i$ and $\mathbf{s}_i$
10:    $O_i \leftarrow O_i \cup o_i$
11: Perform data association between $O_i$ and previous set $O$ of obstacles
12: Perform Kalman filter update on each obstacle's position and predict obstacle
    velocity
13: Publish obstacle list $O$
```

Algorithm 3 Obstacle circle packing

```
1: $G \leftarrow$polygon bounding an obstacle
2: $E \leftarrow$edges of $G$
3: $L \leftarrow$cells inside $G$
4: $S \leftarrow \emptyset$ // initialize list of circles
5: while $L.length > 0$ do
6:     $PQ \leftarrow$priority queue of cells
7:     for each cell $l \in L$ do
8:         $d_i \leftarrow \min_{\forall e \in E} dist(l, e)$
9:         $d_j \leftarrow \min_{\forall c \in C} dist(l, c)$
10:        $c_l \leftarrow$ circle centered on $l$ with radius $\min(d_i, d_j)$
11:        insert $c_l$ into $PQ$ with value $\min(d_i, d_j)$
12:    $M \leftarrow PQ.pop()$
13:    Move $M$ to $S$
14:    Remove all cells from $L$ whose center is in $M$
15: return $S$
```



(a) Example occupancy grid built by the costmap_2d package. Occupied pixels are black, and unoccupied pixels are grey. The grid's resolution is 5cm.



(b) Representation of an obstacle is a hierarchy of a bounding circle and set of packing circles (with a size threshold). The numeric values and arrows represent the predicted velocity.

Figure 15: The sensing module receives an occupancy grid based on depth data and outputs a list of circles representing position and size, and predicted linear velocities for each obstacle.

3.2.1 Overview

Arbitrary obstacles in a 2D environment are approximated by a collection of simpler 2D shapes. Ideally, the approximation should have two qualities: it enables fast collision detection and it is efficient to build. Computing circles to bound an obstacle and to fill an obstacle is efficient, and circles provide fast collision checking. Thus, the goal of the sensing module is to build obstacle approximations using circles, and subsequently predict obstacle speeds and headings.

The input to the sensing module is an occupancy grid (Fig. 15(a)) obtained by passing depth data to the costmap_2d ROS package [58], and the output is a list of obstacles that have position and size (based on circle center and radius), heading, and speed (Fig. 15(b)). After obstacles are found, they are sent to the planning module to use for collision detection.

The system represents each obstacle as a 2-level hierarchy of circles. The top-level circle bounds the obstacle region, and its center is on the obstacle's centroid position with a radius proportional to the size of the obstacle region. The bottom-level circles are inner circles packed into the obstacle. The top-level bounding circle is used to represent the obstacle's position, and the bottom-level inner circles are used to detect collision with the robot.

Kalman filters are used to model uncertainty in obstacle position. In addition to modeling uncertainty, noise is further reduced by accumulating occupancy grids and averaging speed values. Linear velocities are predicted based on position displacement and radius change between sensing scans.

3.2.2 Algorithm for Real-Time Perception

Algorithm 2 outlines the operations of the sensing and perception module. The robot is equipped with a sensor that returns depth data at fixed intervals as the robot moves. The depth data are projected onto a local occupancy grid through the costmap_2d ROS package [58] which is then used to update a global grid of the environment.

Once the global grid is updated, it is combined with recent grids (e.g., all grids in the last 0.25 seconds) by using the OR bitwise operation on corresponding pixels of the grids (line 3, Alg. 2). This ameliorates the effects of noise that cause boundary pixels of an obstacle on the grid disappear.

Since the world is represented as a grid, image processing techniques are leveraged for identifying obstacle regions. The contour points can be found by using the OpenCV library (line 4, Alg. 2) and the results are the vertices of arbitrary polygons that represent obstacle regions.

A bounding circle c_i computed for a polygon (line 7, Alg. 2) has center $\sigma = (x_i, y_i)$, which is the centroid point of the pixels in the polygon, and radius $R = max_i\{dist(z_i \in Z, \sigma)\}$, where Z is the set of all obstacle pixels in the polygon. Circles can be packed into the polygon using Algorithm 3. This algorithm builds a list of inner circles by iteratively computing the largest circle that will fit into the polygon and adding that circle to the list until all space within the polygon is filled.

Each obstacle is represented as a *circle group*, which is an n-*ary* tree with maximum depth 1 where the root node is a bounding cicle and its children are all the inner circles that can be packed into the polygon representing the obstacle region. Operations pertaining to the obstacle's identification, such as data association and velocity prediction, are performed on the bounding circle in the circle group, whereas the inner circles are used when performing collision detection with the robot.

Once a circle group has been computed for each obstacle, data association is performed to relate the latest set of obstacles O_i to the previous set of obstacles O (line 11, Alg. 2). Data association is done by marching bounding circles in O_i to bounding circles (called targets) in the previous set O. Each circle is matched with its closest target, unless 1) the distance between the circles is larger than a threshold, 2) the change in radius is larger than a threshold. If an obstacle in O_i cannot be matched to a target, then a new obstacle is created. If a target has no matched obstacle in O_i, then the corresponding obstacle in O is deleted.

A Kalman filter is maintained for each obstacle to estimate its position (the center of its bounding circle). The result of the Kalman filter update step gives the obstacle's final position. Obstacle speed is predicted based on an obstacle's position displacement and the radius change of the bounding circle (line 12, Alg. 2). These two values are summed to find the total distance covered by an obstacle since the last iteration. Once the speed has been predicted, it is averaged with previous speed values to help stabilize the speeds. An obstacle's direction is computed based on the change in position of the obstacle's bounding circle.

Once the circle groups and linear velocities of each obstacle have been computed, the obstacle information is sent to the planning module to initiate a sensing cycle. In physical experiments performed for this work, the sensing module maintains a publishing rate of 10Hz.

3.3 Implementation, Results, and Discussion

The RAMP-H system has been incorporated with real-world, real-time perception and tested in real experiments. Its effectiveness is verified by experiments where a robot running RAMP-H moves among a walking human and moving robot obstacles as well as static obstacles. The following subsections detail our implementation, experimental environments, and results.

3.3.1 Implementation and Experimental Setup

Our planner is implemented using C++ and utilizes the ROS framework [75] for communication among the different modules. Each module (i.e., RAMP-H planner, trajectory generation, path modification, trajectory evaluation, and trajectory following) runs on its own ROS node. All nodes (other than trajectory following and on-board control of the robot) were run on an Intel i7 8-core CPU at 3.4GHz. The robot platform used for exprimentation was a Turtlebot 2 [25]. Depth data is obtained from the robot's RGB-D camera.

In all experiments, the environment is a 3.5m square with a flat floor and carpet. The robot running RAMP-H begins at location $(0, 0)$ and is oriented towards the goal $(3.5, 3.5)$. Predicted obstacle information (position, size, and linear velocity) is published from the sensing node at 10Hz. The robot's pose is obtained from odometry information calculated by the Turtlebot 2's internal gyroscope and wheel encoders.

The maximum linear and angular accelerations of the RAMP-H robot (Turtlebot 2) are set to 1m/s^2 and $\frac{\pi}{2}$rad/s^2 respectively and the maximum linear speed is set to 33cm/s. The maximum angular speed for the RAMP-H robot is set to $\frac{\pi}{4}$rad/s. The radius of 21cm is used to represent the robot's size when performing collision detection and verifying dynamics constraints for non-holonomic curves.

In the following, we show results for four different scenarios with two unknown static obstacles and different unknown dynamic obstacles: Case 1: one moving obstacle, Case 2: a moving robot followed by a human obstacle, Case 3: both a human and moving robot moving in different directions, and Case 4: both a human and moving robot moving in similar directions.

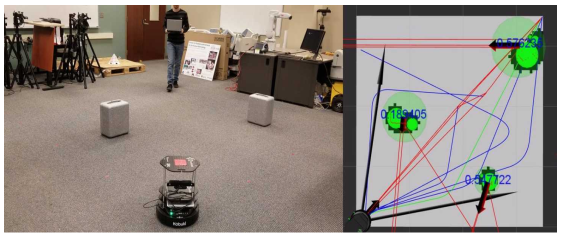
3.3.2 Visualization of Experiments

Figures 16, 17, 18, and 19 show snapshots of a test from each case. In these figures, the rviz window is attached to the real-life image of the test. This visualization shows the position of the robot and obstacles and various other important details. The positions and sizes of obstacles are shown as bounding circles (slightly transparent) in the visualization, and the estimated speeds (in m/s) are shown as the numeric values inside the bounding circles. The estimated direction of an obstacle is shown by an arrow emitting from its bounding circle (the size of the arrow scales with the obstacle's speed, but has a minimum length). The opaque circles inside the obstacle cells are the results of the circle-packing (Algorithm 3) for each obstacle. In addition to obstacle information, the population of trajectories is also displayed. The trajectories are color coded as: blue - feasible, red - infeasible, and green - the current best trajectory in the population (regardless of feasibility). If the obstacles have nonzero speed, then their predicted trajectories are displayed in red emitting from the center of the obstacle. The long black arrows emitting from the robot represent the field of view for the robot's RGB-D camera. This is shown to highlight the limits of the robot's ability to sense the environment. The black arrow emitting from the center of the robot shows its current orientation.

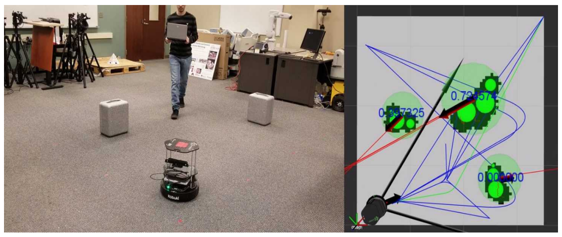
3.3.3 Records of Cycle Periods

Figure 20 helps visualize RAMP-H's cycles (planning, sensing, and control) that occur while the robot is moving in each unknown and dynamic environment. These charts record real data from the experiments. In each of the subfigures, the horizontal axis is the time in seconds and each tick mark shows when a cycle occurred.

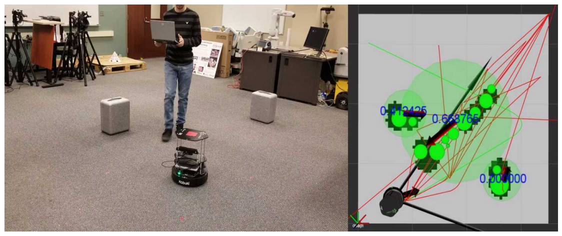
For each test, a fixed number of planning cycles occur before any control and sensing cycles. This is done to improve the optimality of the population. In all tests run for this paper, 50 planning cycles are run in this way. After those 50 planning cycles, control and sensing cycles begin running. The frequency charts reflect this



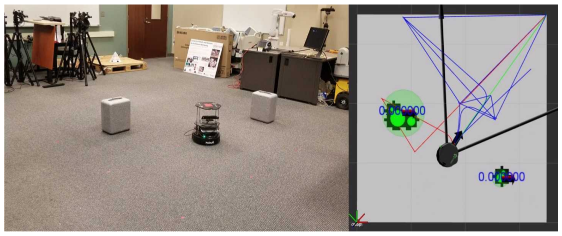
(a)



(b)



(c)



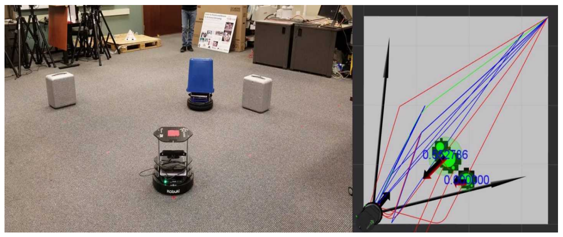
(d)

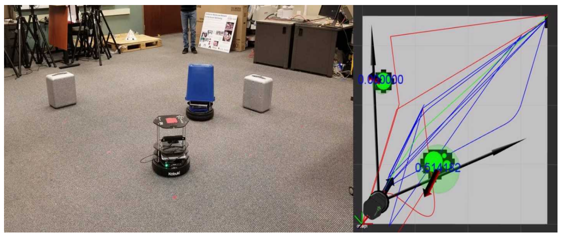
Figure 16: Case 1: One dynamic obstacle (human) and two static obstacles. The Turtlebot 2 platform runs RAMP-H.

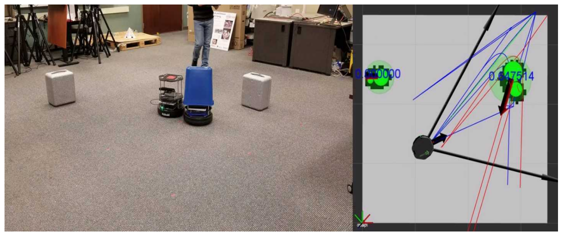
- one can see an extremely dense amount of planning cycles before the control and sensing cycles begin. Once the other cycles begin, the planning cycle periods slow down.

All of the cycles use shared resources and involve inter-process communication with other modules, e.g. sensing cycles involve communication between the planning and sensing modules. The synchronization between modules and resources is handled by ROS.

Planning cycles occur as often as possible, i.e. they do not occur at regular intervals. As such, they are the most frequently occurring cycles during RAMP's execution time. Sensing cycles are set to occur at fixed intervals by the planning module subscribing to obstacle information published by the sensing module at the specified rate. The frequency of control cycles changes (see Section 3.1.3) and is generally much longer


(a)


(b)


(c)

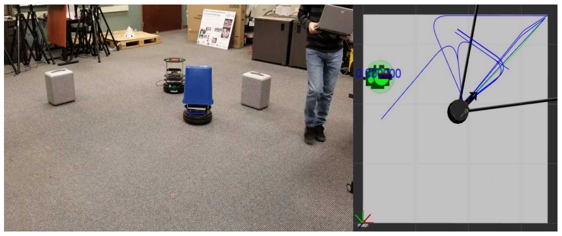

(d)

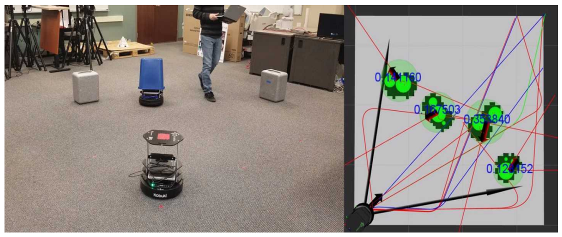
Figure 17: Case 2: Two dynamic obstacles come into view sequentially and two static obstacles. The Turtlebot 2 platform runs RAMP-H.

than planning and sensing cycles.

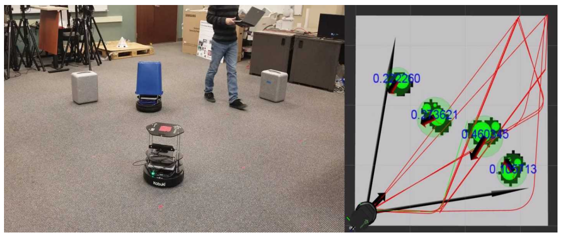
The length of a planning cycle depends on the computational load on the system and the availability of resources that the planning cycles need (e.g. the memory holding the population of trajectories). Sensing cycles are also affected by the computational load on the system. Some gaps in the planning and sensing cycles that are caused by waiting for resources throughout the run, but generally the planning cycles are very dense throughout a run and the sensing cycles are close to the desired rate. Many of these gaps occur right after a control cycle occurs. This is due to control cycles using a large amount of resources and having a longer execution time than the other cycles.

3.3.4 Description of Test Cases and Results

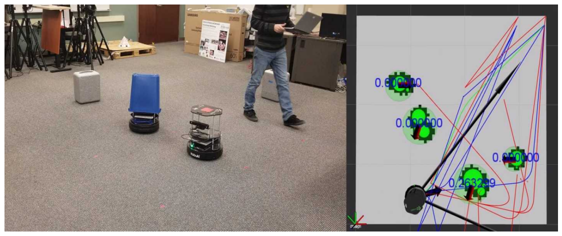
We now explain each case in more detail.



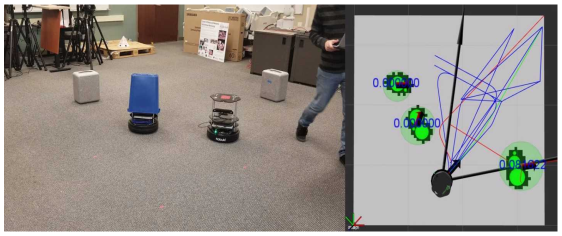
(a)



(b)



(c)



(d)

Figure 18: Case 3: Two dynamic obstacles moving in different directions and two static obstacles. The Turtlebot 2 platform runs RAMP-H.

3.3.4.1 Case 1: One Dynamic Obstacle and Two Static Obstacles

Figure 16(a) shows the environment as the human obstacle moves towards the robot and comes into view. The predicted direction of the obstacle is far from its actual direction (Fig. 16(a)). However, as the obstacle gets closer, its predicted velocity becomes more accurate and the robot switches to a new trajectory to avoid the obstacle (Fig. 16(b)). The robot then continues to move farther from the obstacle until past it, and switches trajectories to move to the goal.

3.3.4.2 Case 2: Two dynamic obstacles interacting sequentially and two static obstacles

This case is meant to show that the RAMP-H system can handle obstacles in a sequential manner, which is something that happens very often in real-world environ-

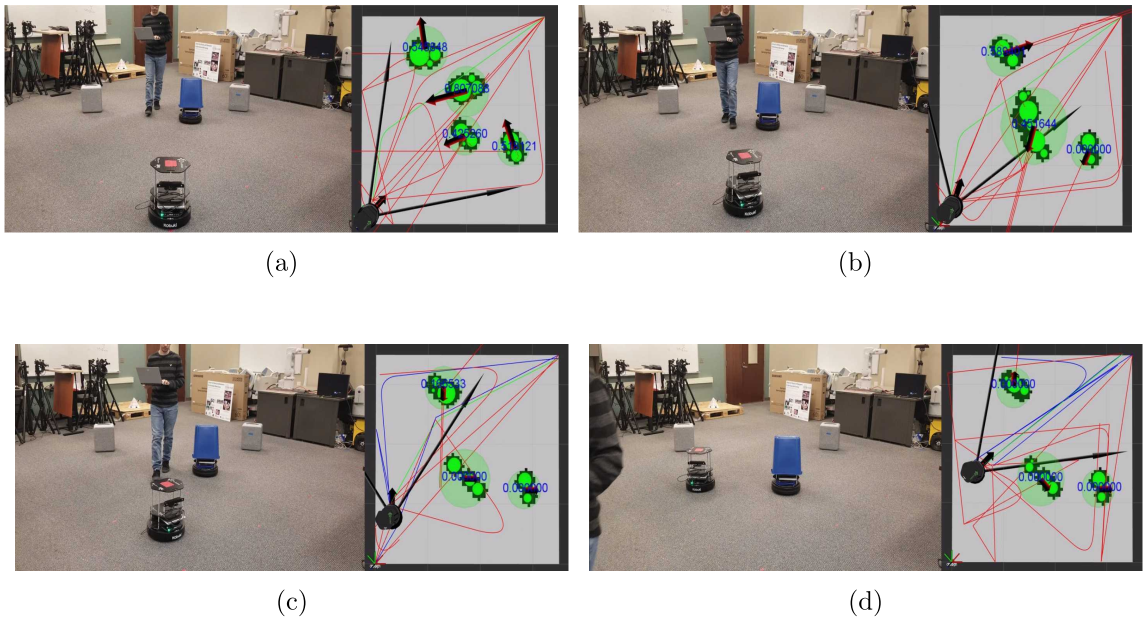
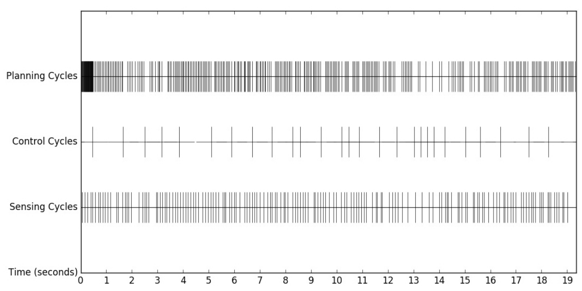


Figure 19: Case 4: Two dynamic obstacles moving in the same direction and two static obstacles. The Turtlebot 2 platform runs RAMP-H.

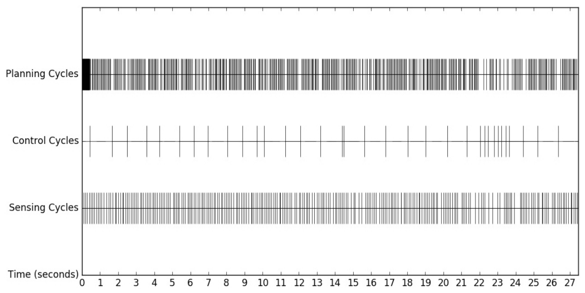
ments, such as moving down a hallway.

At the start of the tests, two static obstacles are placed in the environment (only one can be viewed by the robot due to being on the edge of the field of view), one dynamic obstacle (a robot that will move with constant velocity) is initially positioned several meters away from the robot with orientation directed at the robot, and the other dynamic obstacle (human) is positioned out of the robot's view.

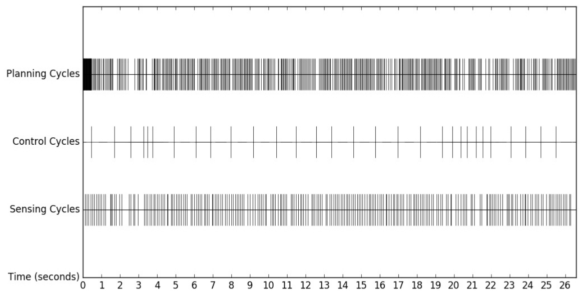
The first dynamic obstacle moves toward the robot in a straight-line and the RAMP-H planner is forced to adapt to this obstacle early in the run (Figs. 17(a), 17(b)). After this first obstacle is avoided, a human obstacle approaches the robot by walking in its path (Fig. 17(c)). The RAMP-H planner detects the new obstacle and switches trajectories to avoid the obstacle and move to the goal (Figs. 17(c), 17(d)).



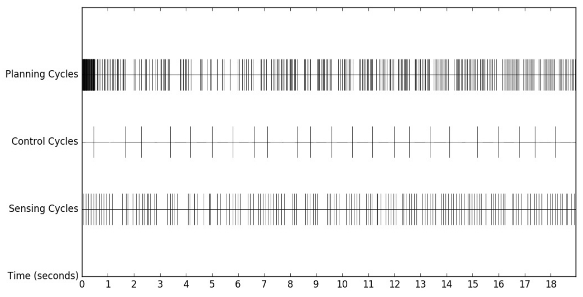
(a) Visualization of cycles for test 1 of case 1



(b) Visualization of cycles for test 1 of case 2



(c) Visualization of cycles for test 1 of case 3



(d) Visualization of cycles for test 1 of case 4

Figure 20: Visualization of cycle frequencies

3.3.4.3 Case 3: Two dynamic obstacles moving in different directions, and two static obstacles

Case 3 places two static obstacles in the environment and has two dynamic obstacles that will be in the robot's view concurrently (as opposed to sequentially in Case 2). In this case, the dynamic obstacles move in significantly different directions.

When the robot begins moving, the two dynamic obstacles start to move. One of those obstacles moves towards the robot, and the other dynamic obstacle (the human) moves away from the robot's initial path (Fig. 18(a)). The robot switches trajectories to avoid the dynamic obstacle moving towards it, but in doing so it turns towards the human dynamic obstacle. The robot switches trajectories again to move between the two dynamic obstacles and then move towards the goal (Figs. 18(b)-18(d)).

3.3.4.4 Case 4: Two dynamic obstacles moving in similar directions, and two static obstacles

Case 4 is similar to Case 3 in that there are two dynamic obstacles that the robot can sense concurrently. However, in this case, the dynamic obstacles move in a similar direction. The distinction between these two cases was made because it typically elicits different behavior from the robot. In Case 3, the robot moves between the two dynamic obstacles, but in Case 4 the robot must move around both dynamic obstacles as if they are one unit.

When the robot begins moving, the two dynamic obstacles start to move towards the robot (Fig. 19(a)). The robot switches trajectories to avoid both of the obstacles with one large curve (Fig. 19(b)). As the obtacles get closer, they become one connected component on the occupancy grid (Fig. 19(b)), and the robot switches to a wider curve to provide more clearance while avoiding the obstacles (Fig. 19(c)). After moving around the dynamic obstacles, the robot switches trajectories again to avoid one of the the static obstacles and move towards the goal (Fig. 19(d)).

Table 1: Performance data for five runs of case 1

Run Number	1	2	3	4	5	Average
Runtime (s)	19.03	25.15	18.01	21.31	18.90	20.48
Min. Dist. from Obstacles (m)	0.75	0.66	0.65	0.32	0.77	0.63
Number of Trajectory Switches	9	12	8	17	9	11
Distance Travelled (m)	8.06	7.31	7.02	6.51	7.03	7.19
Time in Imminent Collision (s)	0.65	0.00	0.00	1.00	0.00	0.33
Number of Planning Cycles	331	416	325	402	355	365.8
Number of Sensing Cycles	162	226	151	203	170	182.4
Number of Control Cycles	27	33	29	24	21	26.8

Table 2: Performance data for five runs of case 2

Run Number	1	2	3	4	5	Average
Runtime (s)	27.56	25.40	21.62	27.88	18.50	24.19
Min. Dist. from Obstacles (m)	0.35	0.38	0.50	0.43	0.72	0.48
Number of Trajectory Switches	13	15	13	18	13	14.4
Distance Travelled (m)	7.19	7.68	6.76	8.24	6.81	7.34
Time in Imminent Collision (s)	2.00	0.00	0.10	0.75	0.30	1.17
Number of Planning Cycles	506	461	405	486	333	438.2
Number of Sensing Cycles	253	239	195	246	159	218.4
Number of Control Cycles	34	33	24	33	23	29.4

Table 3: Performance data for five runs of case 3

Run Number	1	2	3	4	5	Average
Runtime (s)	26.32	23.28	27.42	19.26	19.25	23.11
Min. Dist. from Obstacles (m)	0.64	0.39	0.98	0.65	0.81	0.69
Number of Trajectory Switches	17	8	19	15	14	14.6
Distance Travelled (m)	6.18	7.49	7.05	6.21	6.24	6.63
Time in Imminent Collision (s)	0.00	2.15	0.00	0.00	0.00	0.43
Number of Planning Cycles	461	412	483	336	372	412.8
Number of Sensing Cycles	235	201	251	163	176	205.2
Number of Control Cycles	30	31	29	29	22	28.2

3.3.5 Discussion of Performance

As RAMP-H is a stochastic algorithm, the results from different test runs are different. The most significant difference between the tests is which direction the robot moves to avoid the obstacles. The other differences include the optimality of the trajectory and slight differences in the obstacle predicted speed and direction. After moving away from obstacles, the robot's motion is determined by how quickly RAMP can find a new and better trajectory to the goal.

Tables 1-4 show performance data from the five runs of each case. In Tables 1, the number of times the robot switches trajectories for Case 1 ranges from 9 to 17.

Table 4: Performance data for five runs of case 4

Run Number	1	2	3	4	5	Average
Runtime (s)	19.11	18.78	20.15	22.96	21.69	20.54
Min. Dist. from Obstacles (m)	0.86	0.78	0.48	0.46	0.61	0.64
Number of Trajectory Switches	10	10	9	18	18	13
Distance Travelled (m)	5.99	7.14	7.45	7.16	6.47	6.84
Time in Imminent Collision (s)	0.10	0.00	0.00	0.70	0.00	0.16
Number of Planning Cycles	312	324	368	417	366	357.4
Number of Sensing Cycles	155	153	181	202	191	176.4
Number of Control Cycles	23	29	29	33	32	29.2

Note that switching trajectories is influenced by the need to both avoid obstacles and optimize motion. It can also be influenced by the time spent in imminent collision because the best trajectory in the population can change with each new control cycle during or soon after an imminent collision state.

The number of trajectory switches for Case 2 increases from Case 1 and some runs have double the amount of trajectory switches. This is likely due to the robot twice avoiding dynamic obstacles. Avoiding dynamic obstacles twice can also be the cause of Case 2 having the largest average time spent in imminent collision, as shown in Table 2. Case 2 has the longest average runtime which is a result of both having the largest average time spent in imminent collision and the largest average distance travelled.

The performance data for Case 3 and 4 can be seen in Tables 3 and 4. These cases are similar because they have multiple dynamic obstacles in the robot's view concurrently. Table 3 shows that Case 3 has significantly longer runtime for two of the runs. This can be due to the stochasticity of RAMP - the robot motion could have been a very sub-optimal trajectory to avoid obstacles. Aside from the runtime, the performance results for Cases 3 and 4 are similar.

The number of planning, sensing, and control cycles depends largely on the runtime. As shown in Tables 1-4, the number of these cycles increases as the runtime increases. Small discrepancies in these numbers (such as planning cycles in runs 4 and 5 for Case 3) are due to the computational load throughout a run being larger than others. For instance, this can happen if more obstacles are in view throughout the run, which depends on the motion the robot takes.

Some of the tests have differences in what the robot can initially see. For example, in the test shown in Figure 16, the first snapshot shows that the robot can detect both static obstacles in the initial position. In case 2 shown in Figure 17, the robot can only see one of the static obstacles. This is due to the static obstacles being moved outside of the robot's field of view. In the tests where the robot cannot initially see one of the static obstacles, the static obstacles are detected as the robot moves around the environment.

Although measures are taken to reduce large and rapid changes in the predicted velocities of the obstacles, the sensing module can still generate large differences on new sensing cycles. This is largely due to noise in the depth data. The RGB-D sensor being used in our experiments has a limited range. The maximum recommended range is 3.5m [57]. When obstacles are beyond or near that limit, the noise can cause nonzero speeds and directions that may significantly differ from the real velocity. However, as obstacles get closer to the robot, the predicted velocities become reasonable and the robot is able to avoid the obstacles.

In all of the experiments, only the internal sensors of the Turtlebot 2 (wheel encoders and gyroscope) were used for the RAMP-H localization. This method is sufficient for our experimental environment. Other better localization methods can be used if needed. For instance, a large map of unknown static obstacles can be obtained through running a Simultaneous Localization and Mapping (SLAM) algorithm [85], and then a localization algorithm based on the map from SLAM can be run in parallel with RAMP to obtain the robot's pose as it moves through the environment. This may be necessary for large environments that contain different rooms, for instance.

More sophisticated feedback sensors or localization methods can decrease the amount of motion error to adjust for. However, some motion error will always exist and need to be accounted for. By addressing the issue of adapting to the motion error, our method puts planning and control in sync regardless of how much error has accumulated and if the sensing means are limited, e.g. if a camera cannot be used due to a dimly lit environment.

The experiments carried out cover a variety of cases that an autonomous robot may encounter and illustrate how a RAMP-H system effectively handles them. In addition to the high-level behavior, low-level performance data are provided from the experiments. It should be noted that our experiments were done on a low-cost robot with limited sensing rather than more sophisticated mobile robots that can utilize LIDAR sensing and/or higher speeds. With more accurate and faster sensors, the RAMP-H framework can be more effective in dealing with faster moving obstacles.

CHAPTER 4: LEVERAGING PAST EXPERIENCE IN RAMP

Many real-world applications require robots to navigate in unknown dynamic environments, such as robots serving people in crowded public space, autonomous rescue or security robots, and autonomous self-driving cars, and real-time robot motion planning is required in those environments. Real-time motion planning approaches include, for example, re-planning algorithms [45, 32], Elastic Roadmaps framework [95], velocity obstacle modelling [80], and the Real-time Adaptive Motion Planning (RAMP) framework [89]. While those approaches are based on different algorithms and have different assumptions about the environments, they share one common characteristic: they guide a robot navigating an environment as if it never navigated the environment before; no past experience is incorporated.

However, a person navigates an environment utilizing past experience even though the environment has new dynamic obstacles with unknown motion each time the person visits. For example, inside a food court, a shopper navigates among many unknown people with unknown motion (i.e., without knowing where people are going) but can still utilize past experience (for instance, chairs at a table are likely to be pulled away from the table when people sit down) to navigate more efficiently without collision in such an environment.

The work presented in this chapter is inspired by the above observation and combines learning from past experience and real-time motion planning.

Our approach is to use the Hilbert maps framework [76] to learn a probabilistic occupancy map of a dynamically unknown environment from the observation data collected in previous visits of the environment and incorporate the learned probabilistic map in the RAMP framework for guiding the robot navigation more effectively. Each new visit of the environment by the robot will also add new observation data to improve the Hilbert map of the environment.

4.1 Review of Hilbert maps

The Hilbert maps framework [76] is an approach to learning the occupancy of an environment from depth information to predict the occupancy of a location in the environment. The approach is based on projecting depth data into a high-dimensional Hilbert space defined by an approximate kernel function, and then learning a linear logistical regression model in that high-dimensional space. The result is a sigmoid likelihood discriminative model that can predict the probability of a point in Euclidean space being occupied. The probability that a point is not occupied is defined by the sigmoid function,

$$\mathbf{P}(y = -1|\mathbf{x}, \mathbf{w}) = \frac{1}{1 + exp(\mathbf{w}^T \Phi(\mathbf{x}))} \tag{14}$$

where $\mathbf{w}$ is a parametric vector to be learned and $\Phi(\mathbf{x})$ computes a feature vector for location $\mathbf{x}$. The algorithm described in [79] uses a Bayesian approach to learn the $\mathbf{w}$ vector.

A robot equipped with a depth sensor can build time-indexed occupancy grids of a dynamic environment based on sensing. These occupancy grids act as observations

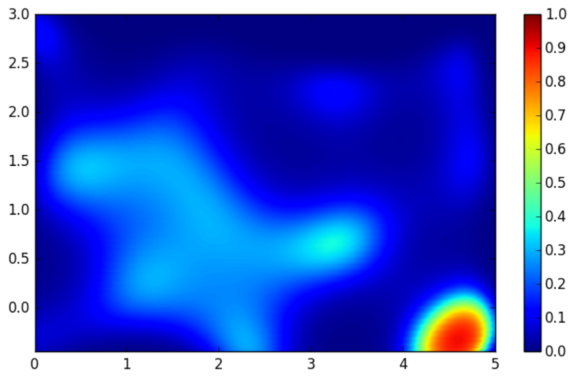


Figure 21: Applying the Hilbert map approach to an environment of size 5m x 3m with one static obstacle (lower right) and one dynamic obstacle that repeatedly moves through the environment. The value at each location is the probability of that location being occupied.

on the environment. The grids can be used as training data for the Hilbert maps algorithm, i.e., the model can be trained based on the real sensing observations from a robot. Once the model has been trained, it can be polled with a position to obtain the probability of the position being occupied.

4.2 RAMP Using Learned Information

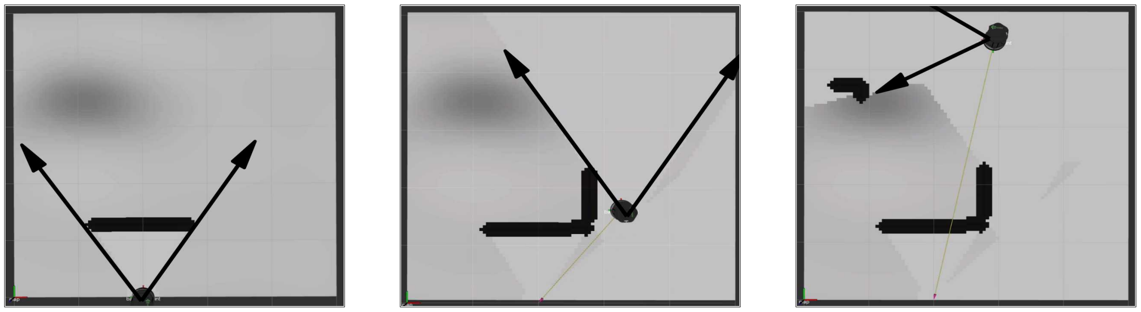
Given a dynamically unknown environment, RAMP normally initializes a population of robot trajectories randomly, and then it guides a robot navigating in the environment through sensing and real-time motion planning.

Now, as the robot moves in the environment guided by RAMP, it also collects observations of the environment. After a sufficient number of observations are collected over time, we use them to build a Hilbert map of the environment. After that, when the robot is going to navigate in the dynamic environment again, RAMP can incorporate the Hilbert map information to obtain a population of trajectories more suited for that environment.

4.2.1 Combining Real Sensor Data with Learned Data

As the robot starts moving in a dynamically unknown environment, only part of the environment can be viewed by its sensors. The occupancy information is unknown to the robot for the non-visible areas until they are within range of the robot's sensors. A trained Hilbert map model can provide useful information for those occluded areas of the environment. Our approach combines the learned Hilbert map information and the real sensed obstacle information for RAMP to use in guiding the robot's movement.

Specifically, an occupancy grid is created that replaces unknown pixel values with probabilistic information. Figure 22 shows an example of this. In this Figure, the robot has an RGB-D camera that can view an environment in a cone. In Fig. 22(a), a large obstacle occludes most of the environment. The cells that the robot can sense have a value of either 0 (occupied) or 255 (free). The value of each occluded cell is replaced by a value corresponding to the probability of occupancy for the location from the Hilbert map. Lower values correspond to higher probabilities of occupancy, i.e., the top-left region has obstacles moving in it more often than the other regions of the environment. As the robot moves, the newly sensed locations have their values on the occupancy grid replaced with real-data values (Figures 22(b) and 22(c)). The probability value of each cell in the unseen region that a robot trajectory passes is used in trajectory evaluation.



(a) Initial grid that combines real and probabilistic data. The static obstacle occludes most of the environment.

(b) As the robot moves in the environment, areas of the map that the robot can sense change to real data.

(c) A second obstacle is detected and more of the environment is revealed.

Figure 22: A Turtlebot 2 is positioned in front of a large static obstacle that occludes most of the environment. The visible locations have values corresponding to the real sensing data (0 or 255), and the pixels that the robot cannot sense are replaced with a value representing the probability of the location being occupied. Lower values correspond to higher probability. The pixel values change to correspond to real data when the robot can sense them (Figs. 22(b) and 22(c)).

4.2.2 Trajectory Evaluation Using Real and Learned Information

In evaluating a trajectory, our approach uses both sensed real obstacles and learned information in unseen regions based on the Hilbert map by adding an additional term to the evaluation functions f and g as follows:

$$Cost_{fe} = f + w_4 * p_{max} \tag{15}$$

$$Cost_{in} = g + p_{max} * Q_{coll} \tag{16}$$

where p_{max} is the maximum probability of occupancy over the cells passed by the trajectory that the robot has not sensed, Q_{coll} is a large constant, and f and g are the evaluation functions from Section 3.1.4.

4.2.3 Initialization Using Learned Information

The learned Hilbert map information of an environment can be used to initialize the population of trajectories before the RAMP robot starts moving and observing the environment. Instead of using a randomly generated population of trajectories directly to start the robot, the RAMP now runs pre-planning cycles to improve the initial population based on the Hilbert map, and uses the evaluation function Eq.(15) to rank trajectories; note that the third term in f is now zero because no sensing and detecting real obstacles happens yet.

4.3 Experimental Results

Two types of experiments were performed to examine the effect of improving the initial population and incorporating learned information into the real-time execution of RAMP. The following subsections detail these experiments and results.

4.3.1 Improving Initial Population

Improving the initial population of RAMP was tested on several different environments. For each environment, a robot equipped with an RGB-D camera is placed at the origin of a $3.5m$ square environment. The robot remains stationary while dynamic obstacles move around the environment. The obstacle motion is captured by the robot and an occupancy grid is created for each set of depth data. The occupancy grids are indexed by time and used as training data to train a Hilbert Map model of the environment occupancy. Once the model has been trained, 500 pre-planning cycles (Section 4.2.3) are run before sensing and control cycles begin so that the initial population can be improved. During these pre-planning cycles, all trajectories are treated as feasible since there is no real obstacle information. The expectation of these experiments is that after running the pre-planning cycles the initial population will have only a small number of trajectories in the regions with high probability of occupancy. This results in many options to avoid these regions while also preserving diversity in the population.

The remainder of this subsection will cover three cases used to experiment with Hilbert maps: one dynamic obstacle moving in a straight line, one static and one dynamic obstacle moving in a straight line, and two dynamic obstacles moving in a straight line. In all cases, the RAMP robot does not know how the dynamic obstacles will move.

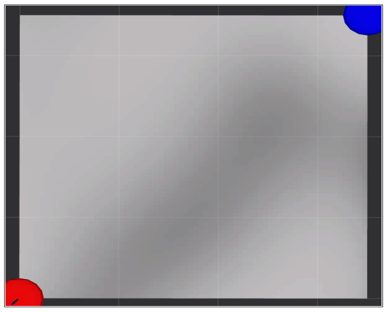
4.3.1.1 One Dynamic Obstacle Moving in a Straight Line

The first case examined is one with a single dynamic obstacle commanded to move in a straight line back and forth. The obstacle moves at roughly 0.33m/s and stops for 1 second before changing directions (forward or backward). It is initially positioned near a corner in the environment (Fig. 23(a)) and moves along a diagonal through the environment to roughly simulate walking along a common walking path that a human agent may take.

Time-indexed occupancy grids for this experiment (2570 in total) were collected and used to train a Hilbert map model. The occupancy grid generated from the model can be seen in Fig. 23(c). The results of one experiment is shown in Figure 23(d). The population shown is the population after running 500 pre-planning cycles. The

(a) Initial location of environment

(b) The moving robot obstacle moves on a straight line towards the robot and then reverses back to its initial position.


(c) The occupancy grid generated from the Hilbert map model

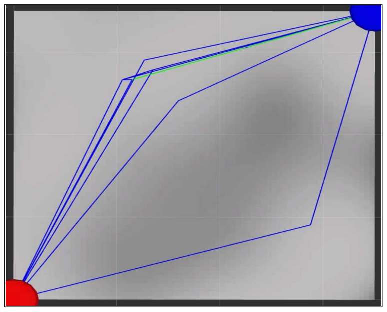

(d) Initial population of RAMP after incorporating learned information

Figure 23: Hilbert map of an environment with one dynamic obstacle moving with straight line motion, and the result of one instance of running 500 pre-planning cycles using learned information.

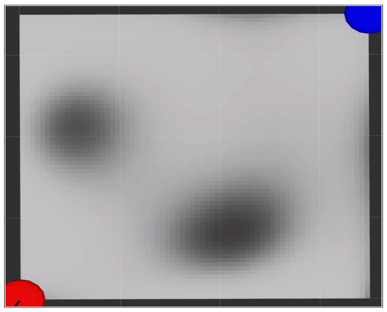
majority of trajectories avoid the area with high probability of occupancy while one trajectory travels through a high occupancy region.

4.3.1.2 One Static Obstacle and One Dynamic Obstacle Moving in a Line

Another dynamic environment was examined that contained one static obstacle and one dynamic obstacle moving in a small line (Fig. 24). There were 1,179 total time-indexed occupancy grids used to train the model for this case. The occupancy information generated from the Hilbert map model shows two areas with large probability of occupancy. Because the dynamic obstacle moves in a small space, the area

(a) Initial setup of environment

(b) The moving robot obstacle moves 1m towards origin. The other obstacle (grey box) is static.


(c) The occupancy grid generated from the Hilbert map model

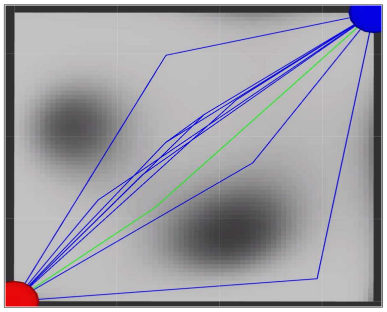

(d) The population after 500 pre-planning cycles

Figure 24: Hilbert map of an environment with one static obstacle and one dynamic obstacle moving in a 1m line, and the result of one instance of running 500 pre-planning cycles using learned information.

is nearly always occupied so it has similar probability to the static obstacle.

Figure 24 shows an example of a population after 500 pre-planning cycles using the Hilbert map trained on this environment. Most of the trajectories go between the two obstacle regions, but the population still maintain two trajectories that cover the obstacle regions.

4.3.1.3 Two Dynamic Obstacles Moving in Straight Lines

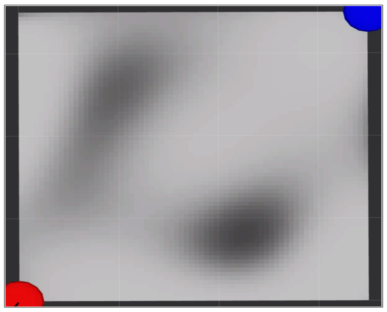
A case with two dynamic obstacles was designed with a human obstacle walking in a line with size roughly 1.5m and a moving robot driving in a line with size roughly 1m. In this environment, the Hilbert map shows a long streak where the human walked and a smaller area for the moving robot (similar to the moving robot in case 3). A total of 1,439 occupancy grids were used to train the Hilbert map model for this case. The population seen in 25(d) shows that most trajectories are concentrated in moving along the diagonal of the environment to avoid both regions with high probability of occupancy, while two trajectories travel within the high probability regions.

Experiments on real robots were performed for this environment to show that the performance of RAMP can be improved when the initial population is improved. The robot is initially positioned at $(0m, 0m)$ with orientation 0.785rad and its goal is $(3.5m, 3.5m)$. While the robot moves, the obstacles move in straight lines back and forth to simulate many obstacles moving in these regions. Ten total tests were carried out: five tests that used no prior information in RAMP, and five tests that performed 100 pre-planning cycles using the probabilistic obstacles generated from the Hilbert map data (Fig. 25(d)).

The performance data for these tests can be seen in Table 5. The tests utilizing the learned information perform better in all categories except the time spent in imminent collision and minimum distance to obstacles. However, for these two categories the results are similar. Note that for the minimum distance to obstacles metric, larger values are better. For all categories, the standard deviation is lower when improving the initialization step with learned information. The benefits of utilizing the Hilbert map model can be mostly seen in the runtime and number of trajectory switches. For those categories, the mean is significantly lower and the standard deviation is far lower. This implies that utilizing the Hilbert map model to generate trajectories that avoid areas of high probability allows the robot to move to the goal faster because it

(a) Initial setup of environment

(b) The moving robot obstacle moves 1m towards origin. The human obstacle moves roughly 1.5m towards the origin.


(c) The occupancy grid generated from the Hilbert map model

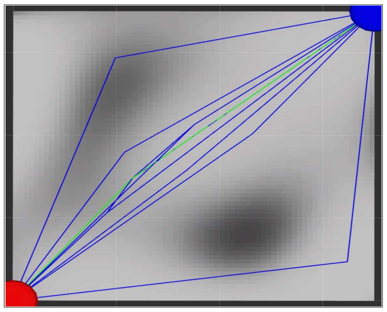

(d) The population after 500 pre-planning cycles

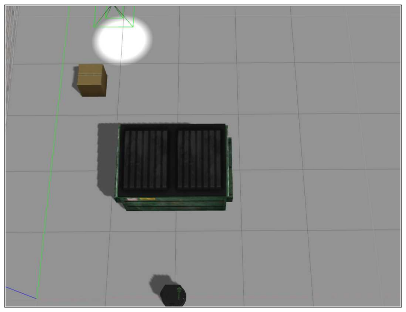
Figure 25: Hilbert map of an environment with two dynamic obstacles, and the result of one instance of running 500 pre-planning cycles using learned information.

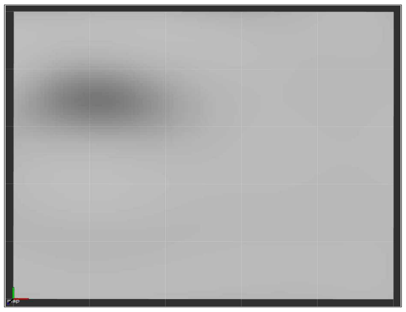
is more likely to avoid obstacles, and the robot will not need to switch trajectories as often to avoid obstacles (although it may still switch to a more optimal trajectory).

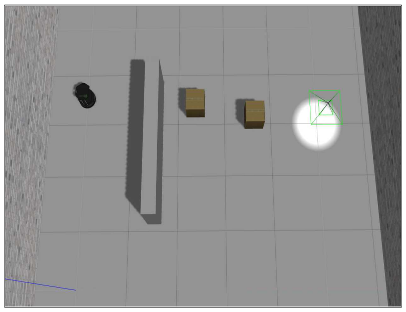
4.3.2 Real-time Execution with Learned Information

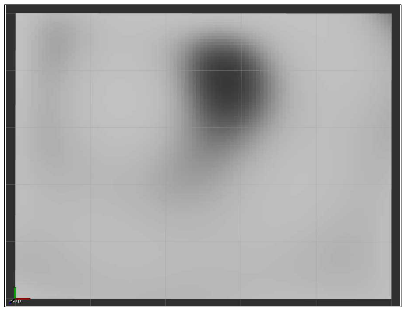
Experiments were performed incorporating the learned information both in the initialization and in the real-time simultaneous moving, sensing, and planning process of the RAMP robot.

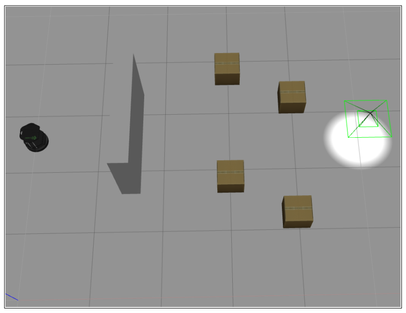
Three $5m$ square environments were created in Gazebo [44]. They can be seen in Figure 26. Each environment contains 1-4 dynamic obstacles. When collecting data to train the Hilbert map model (one for each environment), the robot is positioned


(a) Environment with one static obstacle (dumpster) and one moving obstacle (cardboard box). The goal is (1.0, 5.0).


(b) Occupancy grid generated from Hilbert Map model trained on 771 occupancy grids where the dynamic obstacle moves on a 1m diagonal path repeatedly.


(c) Environment with one static obstacle (rectangle box) and two moving obstacles (cardboard boxes). The goal is (5.0m, 3.0m).


(d) Occupancy grid generated from Hilbert Map model trained on 741 occupancy grids where the dynamic obstacles move on a 1m diagonal path repeatedly.


(e) Environment with one static obstacle (rectangle box) and four moving obstacles (cardboard boxes). The goal is (5.0m, 2.5m). The two bottom dynamic obstacles move at twice the speed of the top two dynamic obstacles.

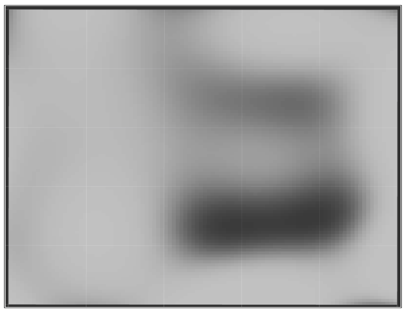

(f) Occupancy grid generated from Hilbert Map model trained on 1682 occupancy grids where the dynamic obstacles move on a 1m horizontal path repeatedly.

Figure 26: Three 5m square environments for testing the effect of incorporating probabilistic information, and their associated occupancy grids generated by the Hilbert maps model. The static obstacles are removed when recording training data.

so that it can view the dynamic obstacles' motion. The obstacle motion is captured by the robot's RGB-D camera sensor and an occupancy grid is created for each set of depth data. The occupancy grids are indexed by time and used as training data to train the Hilbert map models. Next, two sets of five robot navigation runs are performed on each environment: one set does not use any learning, and the other set uses learning to both improve the initial trajectory population and real-time RAMP execution.

The first environment can be viewed in Figure 26(a). The robot's starting position is $(2.0m, 0.0m)$, and its goal position is $(1.0m, 4.5m)$. The environment contains a moving obstacle (cardboard box) that repeatedly moves on a 1 meter straight line path. A large static obstacle is placed in front of the robot's initial position to occlude most of the environment. The first set of runs do not incorporate any probabilistic information and only considers the time to execute a trajectory, i.e., $w_1 = 1$ in Eq. 15 and all other weights are 0. This will cause the robot to move to the left side of the environment more often because it leads to a faster trajectory. However, this leads the robot to the moving obstacle, and it has to maneuver around in a tight space.

Table 5: Performance data for real-robot tests with two dynamic obstacles (Section 4.3.1.3). Two sets of tests were run: one that did not use any learned information, and one that improved trajectory initialization using learned information (Section 4.2.3). For each set, 5 tests were run.

	Mean without learning	Mean with learning
Runtime (seconds)	28.77 ± 7.41	21.12 ± 2.36
# of trajectory switches	14 ± 10.89	12.8 ± 3.83
Time in imminent collision (s)	0.44 ± 0.72	0.5 ± 0.58
Distance traveled (m)	6.05 ± 1.16	5.80 ± 0.48
Min. dist. from obstacles (m)	0.71 ± 0.28	0.66 ± 0.14

The second set of navigation runs incorporates the probabilistic information to improve the initial robot trajectory population and to consider the probability of occupancy for unseen locations during real-time execution. The grid generated from the Hilbert map model is shown in Figure 26(b). The expectation is that using the probabilistic information will cause the robot to move to the right of the static obstacle more often despite trajectories in this direction having a significantly longer distance to the goal. Essentially, the planner is choosing to avoid the region of the map with high probability of occupancy, despite the longer execution time, to more easily reach the goal.

Table 6 shows the results of these experiments. Utilizing the probabilistic information about the occupancy of the environment improves the performance of the robot in each metric. Surprisingly, it resulted in a lower execution time and only a slightly longer distance travelled. The reason is that, without using the learned information about unseen areas, the robot might need to stop, turn around, or move away from the goal at some point to navigate around the moving obstacle when it went into the region with high probability of occupancy. When avoiding the region with high probability of occupancy, however, the robot could go to the goal with very little need to navigate around the obstacle.

The second environment is shown in Figure 26(c). It contains two dynamic obstacles and a more elongated static obstacle. The robot's starting position is $(0.0m, 3.5m)$ and the goal position is $(5.0m, 3.0m)$. In this environment, the increased time of avoiding of the high probability of occupancy region is more severe because the static obstacle is larger. However, the region with high probability of occupancy now con-

Table 6: Performance data for tests that combine real and probabilistic data during runtime in the environment shown in Fig. 26(a). Two sets were run: one that did not use any learned information, and one that used probabilistic data to improve trajectory initialization and real-time execution. For each set, 5 tests were run.

	Mean without learning	Mean with learning
Runtime (seconds)	29.744 ± 4.76	26.29 ± 5.01
# of trajectory switches	20 ± 4.06	17.4 ± 5.13
Time in imminent collision (s)	0.77 ± 1.07	0.00 ± 0.00
Distance traveled (m)	10.94 ± 3.13	11.21 ± 1.56
Min. dist. from obstacles (m)	0.84 ± 0.24	1.03 ± 0.40
Traversed high occupancy region	4 of 5 runs	0 of 5 runs

Table 7: Performance data for tests that combine real and probabilistic data during runtime in the environment shown in Fig. 26(c). Two sets were run: one that did not use any learned information, and one that used probabilistic data to improve the initialization and real-time execution. For each set, 5 tests were run.

	Mean without learning	Mean with learning
Runtime (seconds)	32.69 ± 7.66	35.26 ± 7.32
# of trajectory switches	19 ± 4.92	19 ± 9.15
Time in imminent collision (s)	0.04 ± 0.09	0.00 ± 0.00
Distance traveled (m)	10.22 ± 2.68	12.25 ± 0.66
Min. dist. from obstacles (m)	0.61 ± 0.18	1.01 ± 0.22
Traversed high occupancy region	4 of 5 runs	0 of 5 runs

tains multiple dynamic obstacles so that navigating this area is more challenging and unsafe. The results of tests in this environment are shown in Table 7. In this environment, the most significant performance difference is in the minimum distance to obstacles. When using learned information, the robot maintains a significantly larger clearance to the obstacles. The tests without using prior information perform better in runtime and distance travelled, but that is expected since the robot needs to travel farther to go around the static obstacle. This trade-off is worthwhile because maintaining a larger distance from obstacles results in safer motion.

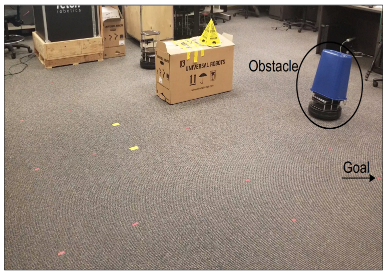
The third environment is shown in Figure 26(e). It contains four dynamic obstacles and one static obstacle. The robots starting position is $(0.0m, 2.5m)$ and the goal

Table 8: Performance data for real tests that combine real and probabilistic data during runtime in the environment shown in Fig. 26(e). Two sets were run: one that did not use any learned information, and one that used probabilistic data to improve the initialization and real-time execution. For each set, 5 tests were run.

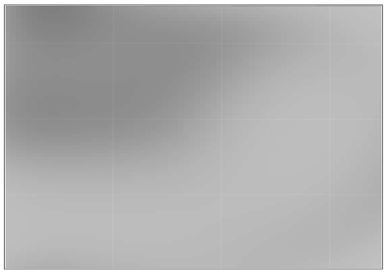
	Mean without learning	Mean with learning
Runtime (seconds)	33.07 ± 8.34	26.3 ± 2.66
# of trajectory switches	15.00 ± 6.48	12.6 ± 3.78
Time in imminent collision (s)	0.00 ± 0.00	0.00 ± 0.00
Distance traveled (m)	9.64 ± 1.53	9.26 ± 0.83
Min. dist. from obstacles (m)	2.05 ± 1.39	1.98 ± 0.92
Traversed high occupancy region	5 of 5 runs	0 of 5 runs

position is $(5.0m, 2.5m)$. The two dynamic obstacles at the bottom of the environment move at roughly twice the speed of the obstacles at the top of the environment. This is captured in the occupancy grid generated by a Hilbert map model in Figure 26(f). The bottom region has a much higher probability of occupancy due to the faster obstacles being at various positions along their path more often. This is interesting because incorporating the probabilistic data allows the robot to capture the effect of obstacle speeds in a region. The results of tests in this environment are also shown in Table 8. All performance metrics are improved with the incorporation of the Hilbert map information. The most significant improvement is the improved robot runtime to avoid the faster obstacles.

Experiments were performed on a real environment that utilized the probabilistic data during real-time execution. The environment can be seen in Figure 27(a). This environment is a $3.5m$ square containing one static obstacle and one dynamic obstacle. The robots starting position is $(1.5m, 0.0m)$ and the goal position is $(1.0m, 3.5m)$. The static obstacle occludes the dynamic obstacle in the environment when the robot is in its initial configuration. Data was collected to train a Hilbert maps model by



(a) Initial configuration of the environment. The obstacle (blue) moves on a $1m$ line repeatedly.



(b) Occupancy grid generated from Hilbert Map model trained on 1642 occupancy grids where a human obstacle moves randomly in roughly a $1.5m$ square.

Figure 27: Real environment ($3.5m$ square) for experiments combining real and probabilistic sensing data with one static obstacle and one moving obstacle. The robot's initial position is $(1.5, 0)$ and the goal is $(1.0, 3.5)$.

a human obstacle walking randomly in a small region in the top-left corner of the environment. During tests, the dynamic obstacle moves on a $1m$ line repeatedly. Table 9 shows the results of tests in this environment. Utilizing the probabilistic data during real-time execution and when improving the initial population led to better results in each category except distance travelled. When the robot avoids the region with the dynamic obstacle, it is able to reach the goal in less time despite the longer travel time because it does not have to spend time navigating around the dynamic

Table 9: Performance data for real tests that combine real and probabilistic data during runtime in the environment shown in Fig. 27(a). Two sets were run: one that did not use any learned information, and one that used probabilistic data to improve the initialization and real-time execution. For each set, 5 tests were run.

	Mean without learning	Mean with learning
Runtime (seconds)	31.78 ± 10.11	27.32 ± 4.28
# of trajectory switches	20 ± 9.25	13.6 ± 5.32
Time in imminent collision (s)	1.16 ± 1.29	0.16 ± 0.32
Distance traveled (m)	7.11 ± 1.55	6.34 ± 0.87
Min. dist. from obstacles (m)	1.39 ± 0.55	2.37 ± 0.84
Traversed high occupancy region	5 of 5 runs	0 of 5 runs

obstacle. It also leads to a significantly higher minimum distance to obstacles which makes the overall motion much safer. The number of trajectory switches is less which is likely due to the robot not needing to adapt to a nearby dynamic obstacle, so the trajectory switches are largely done to make the robot's motion more optimal.

CHAPTER 5: SYSTEM-LEVEL TESTING OF A REAL-TIME ADAPTIVE MOTION PLANNING SYSTEM

This chapter introduces an approach to generate executable test cases for a Real-time Adaptive Motion Planning (RAMP) system.

5.1 Model-based Testing Approach

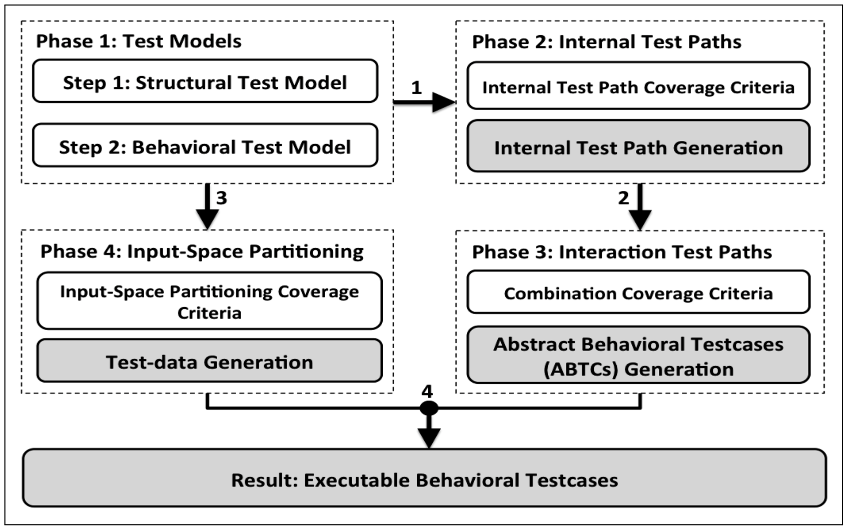
Our objective is to apply a systematic Model-based Testing (MBT) approach [5, 6, 7, 8] to testing RAMP by generating test cases for system-level testing. The MBT approach addresses the system under test (SUT) and the objects that SUT interacts with, called *actors*. In system-level testing, the SUT is the robot, and the obstacles that this robot interacts with are actors. The actors can be modelled structurally and behaviorally as test models.

As shown in Figure 28, the process is decoupled into four phases:

- Phase 1: Create the test models of actors (i.e., obstacles) by constructing a single structural test model that represent all actors together and a behavioral test model for actors.

- Phase 2: Select proper graph-based coverage criteria to generate internal test paths that cover behaviors of each actor separately.

- Phase 3: Use proper graph-based concurrent coverage criteria (i.e., Combination Coverage Criteria) to combine the internal test paths into interaction test paths



Phase 1: Test Models
Step 1: Structural Test Model
Step 2: Behavioral Test Model
1
Phase 2: Internal Test Paths
Internal Test Path Coverage Criteria
Internal Test Path Generation
3
2
Phase 4: Input-Space Partitioning
Input-Space Partitioning Coverage Criteria
Test-data Generation
Phase 3: Interaction Test Paths
Combination Coverage Criteria
Abstract Behavioral Testcases (ABTCs) Generation
4
Result: Executable Behavioral Testcases

Figure 28: Model-based test generation process

(namely Abstract Behavioral Test Cases (ABTCs)) to represent the behavioral test model.

- Phase 4: Select proper input-space partitioning coverage criteria to generate *test data* from the structural test model in order to transform the generated ABTCs into executable behavioral test cases.

Once a behavioral test model has been created, graph-based testing criteria from [4] can be used to generate internal test paths. Next, the internal test paths are combined to represent the possible interactions among actors. For system-level testing, the ABTCs are generated using a novel coverage procedure. The ABTCs need *test data* to be executable. The *test data* are generated from the *structural test model*'s properties and functions' parameters by using input-space partitioning [4]. The generated ABTCs are then transformed into executable behavioral test cases with each selection from the generated *test data* assigned to an ABTC to produce an executable test case, and so forth.

5.2 System-level Testing for RAMP

System testing level involves generating world states as inputs to the RAMP planner as a System Under Test (SUT) and evaluating the RAMP's output. In the RAMP system, testing is focused on interactions between the RAMP robot and obstacles in close spatial and temporal proximity, as described in more detail below. A world state for RAMP includes the robot's current state and the set of obstacle positions and velocities in a small spatial and temporal neighborhood of the robot's current state. The output is a motion trajectory that is expected to avoid obstacles and lead the robot to its goal.

5.2.1 Interacton with Obstacles

A robot guided by the RAMP system operates in environments that contain dynamic obstacles moving with unforeseen motion. RAMP uses the sensed information of obstacle poses to predict the future motion of obstacles. As the robot is moving towards its goal, the RAMP system continuously plans and adapts the robot motion to obstacle motion based on the latest sensing information.

Since RAMP must adapt to unknown trajectories, the system-level tests need to evaluate RAMP by considering any possible motion from the obstacle. However, modeling long-range and long-term obstacle trajectories is not only infeasible because there are countlessly possible obstacle trajectories but also not very useful in evaluating RAMP's performance because obstacles far from the robot do not affect the robot's short-term motion in significant ways until they become close to the robot.

Hence, instead of modeling obstacle trajectories, our approach is to focus on interaction between the RAMP robot and obstacles by modeling the behaviors of obstacles that are both spatially and temporally close to the robot. An obstacle is temporally close to the robot if its motion will likely lead to a collision with the robot in a short time period. Taking advantage of spatial and temporal coherence, we thus only model obstacle behaviors within a small spatial and temporal neighborhood of the robot's current state in configuration-time space. Evaluating RAMP's performance in guiding the robot move towards its goal while avoiding obstacles in such a neighborhood significantly reduces the total number of possible obstacle behaviors necessary for testing.

For obstacles within a spatially and temporally very small neighborhood around the robot, they are reasonably modeled as moving with arbitrary but constant linear and angular velocities towards the robot, and the RAMP system runs to produce robot motion that both avoids the obstacles and leads the robot to its goal. Note that maintaining the global task of moving towards a goal is important because RAMP is a real-time global motion planner. Otherwise, the tests would only evaluate obstacle avoidance.

5.2.2 Phase 1: Obstacle Test Model

The obstacle structural test model describes the data structures used for representing the behaviors of a dynamic obstacle interacting with the RAMP robot. The obstacle behavioral test models describe those behaviors as states and transitions.

The structural test model is constructed using a UML class diagram shown in

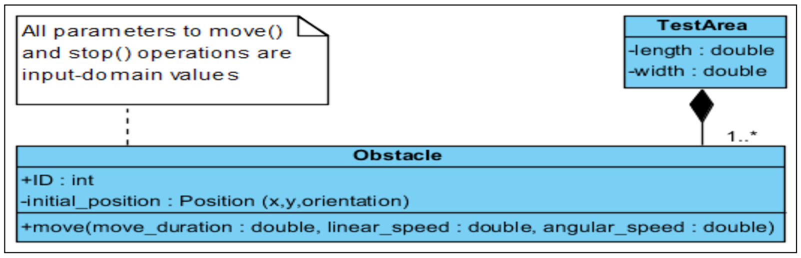

All parameters to move() and stop() operations are input-domain values
TestArea
-length : double
-width : double
1..*
Obstacle
+ID : int
-initial_position : Position (x,y,orientation)
+move(move_duration : double, linear_speed : double, angular_speed : double)

Figure 29: Structural model for a dynamic obstacle

Figure 29. The obstacles are assumed to be moving in a 2-D space, and all obstacles are assumed identical. The *TestArea* class describes a small 2-D region to place obstacles to ensure that the obstacles will have a significant impact on the robot's motion in the region. The robot's initial position will be at one corner, and the goal position will be at the opposite corner along the diagonal of the test area.

In such a small spatial and temporal neighborhood, obstacle motion can be reasonably modeled as having constant linear and angular velocity, and the linear velocity direction is set towards the robot. Therefore, the *Obstacle* class consists of current position, velocity values, and the ID of an obstacle. The parameters *initial_position, linear_speed, angular_speed, stop_duration, and move_duration* are input domains [4], which are exploited to generate *test data*.

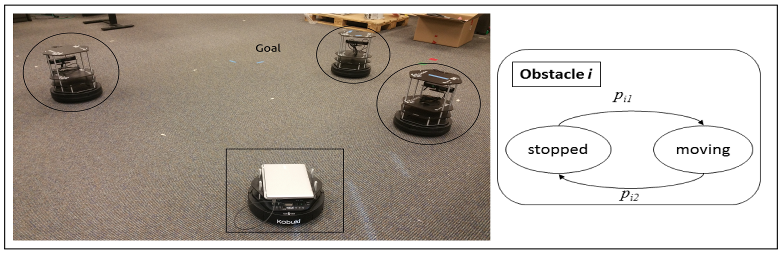


Figure 30: Robot (square) and obstacles (circles) in an example test area, and the obstacle behavioral test model on the right.

As shown in Figure 30, each obstacle is modeled as an EFSM. The obstacles have two behaviors: *move* and *stop*. The behaviors transition the obstacles into one of two states: *stopped* and *moving*. These states are translated as test messages and sent to the RAMP system.

The RAMP system will receive this information, predict obstacle movement, and then attempt to find a trajectory for the robot to avoid obstacles and reach its goal at the opposite corner of the test area. Thus, the functionality of RAMP can be evaluated by running the RAMP system with sequences of messages containing different obstacle EFSM states and different parameter values.

5.2.3 Phase 2: Obstacle Internal Test Paths

Each ith obstacle's EFSM is associated with a set of transitions, P_i, as illustrated in Figure 30. They are ordered as sequences of method invocations (i.e., function calls) to represent an internal test path that shows an obstacle's motion pattern as $\{stop() \rightarrow move(v, \omega) \rightarrow stop()\}$, where v is *linear_speed*, and ω is *angular_speed*.

The obstacle behavioral test model can be defined as a collection of concurrent processes, $OBTM = \{Proc_1, Proc_2, \ldots, Proc_i\}$ where $1 \leq i \leq M$, and M is the number of obstacles. Each process $Proc_i$ is covered by an internal test path.

5.2.4 Phase 3: Obstacle Interaction Test Paths

Since the obstacles interact concurrently, their internal test paths are considered concurrent as well. Therefore, the internal test paths need to be combined to represent all possible interactions of the obstacles. Since all obstacles have the same behavioral model, they are considered identical and indistinguishable to the robot.

Only the obstacle locations and velocities matter. That is, from the perspective of the robot, if an obstacle is closer to the robot than the other obstacles, there is no difference whether the closer obstacle is obstacle 1 or obstacle 2. However, there is no existing coverage criterion that addresses indinstinguishable and concurrent internal test paths.

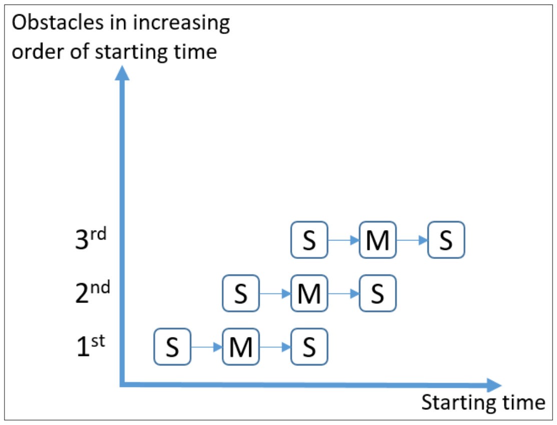
Hence, we have developed a novel and systematic method that enumerates all kinds of concurrency, i.e., ABTCs, among m identical obstacles, where each obstacle's internal test path has three nodes as *stopped* $\rightarrow$ *moving* $\rightarrow$ *stopped*. The effect is similar to what partial order reduction [24] achieves. We define the *starting time* of each obstacle in terms of a number indicating the delay from the earliest moving obstacle: 0 (to mean no delay), 1 (to mean delay by 1 node), 2 (to mean delay by 2 nodes), and so on. If we denote the 1st obstacle as the obstacle that first starts moving in the test region, and the 2nd obstacle as the obstacle that moves next, and so on, then we can describe the possible values of starting time for every obstacle as follows:

1st obstacle: 0

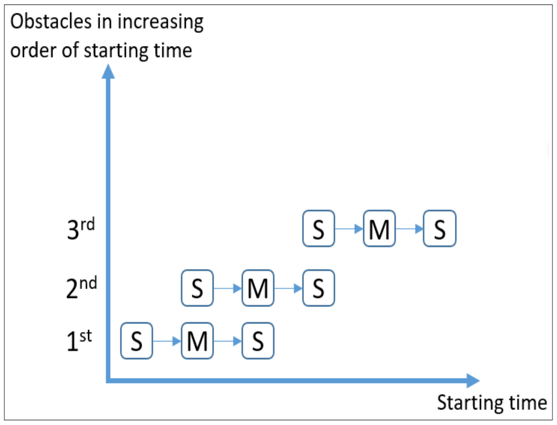
2nd obstacle: 0, 1, 2

ith obstacle: 0, 1, 2, ..., $(i-1)$th obstacle starting time + 2, $3 \leq i \leq m$.

Note that if two or more obstacles have the same starting time value, say 0, it means they start moving simultaneously. If the 2nd obstacle has starting time 1, it means that when the 2nd obstacle starts moving, the 1st obstacle is already moving. If the 2nd obstacle has starting time 2, it means that when the 2nd obstacle starts moving,



(a) ABTC $< 0, 1, 2 >$



(b) ABTC $< 0, 1, 3 >$

Figure 31: Illustration of ABTCs for system-level testing involving 3 obstacles. Obstacles are shown transitioning from a stopped state (S) to a moving state (M). The delays correspond to the number of state changes of the prior moving obstacles. The bottom obstacle is always the obstacle that begins first, i.e. its delay is 0. The 2nd and 3rd obstacles begin at their respective delays after the first obstacle.

the 1st obstacle has already finished moving.

Now we can express every kind of concurrency, i.e., every ABTC, among m obstacles as an m-tuple of values: $< d_1, d_2, ..., d_m >$, such that:

- $d_i, 1 \leq i \leq m$ is the starting time (as defined above) of the ith obstacle,
- $d_i \geq d_k, \forall k < i$.

With this expression and together with the value ranges of obstacle starting time, all possible ABTCs for m obstacles can be enumerated automatically. For $m = 2$, there are 3 ABTCs: $< 0, 0 >$, $< 0, 1 >$, and $< 0, 2 >$.

For $m = 3$, there are 9 ABTCs: $< 0, 0, 0 >$, $< 0, 0, 1 >$, $< 0, 0, 2 >$, $< 0, 1, 1 >$, $< 0, 1, 2 >$, $< 0, 1, 3 >$, $< 0, 2, 2 >$, $< 0, 2, 3 >$, and $< 0, 2, 4 >$. Each tuple represents a unique concurrency case. For example, $< 0, 1, 3 >$ means that the 2nd obstacle starts moving when the 1st obstacle is already moving, and the 3rd obstacle starts moving when the 2nd obstacle has finished moving. Note that the largest value of d_3 (the last number in the 3-tuple) depends on the value of d_2 in the same tuple. For example, if $d_2 = 1$, then 3 is the largest value of d_3, i.e., $< 0, 1, 4 >$ is not a valid tuple. On the other hand, $< 0, 2, 4 >$ is valid, meaning that the 2nd obstacle moves when the 1st obstacle has finished moving, and the 3rd obstacle moves when the 2nd obstacle has finished moving. For $m = 4$, ABTCs can be enumerated in the similar way, and there are 27 ABTCs.

5.2.5 Phase 4: Obstacle Input-Space Partitioning

The input domains of each obstacle represent obstacle parameters (*initial_position*, *linear_velocity*, *angular_velocity*, *stop_duration*, and *move_duration*). We partition the input domains into blocks of values and randomly chose values from each of the blocks based on the Each Choice Criterion (ECC) [4] to generate *test data* for converting the generated ABTCs into executable test cases. As shown in Table 10, the input domains for a single obstacle are partitioned into 12 total blocks.

Table 10: Input domains and blocks of values of a mobile obstacle

Parameter	**Input Domains**	**Blocks of Values**
initial_position	$x = [0.5, 2]m$ $y = [0.5, 2]m$	$[0.5, 1]$,$(1, 1.5]$,$(1.5, 2]$ $[0.5, 1]$,$(1, 1.5]$,$(1.5, 2]$
linear_speed	$v = (0, 0.5)\frac{m}{s}$	$[0, 0.25]$,$(0.25, 0.5]$
angular_speed	$w = (-\frac{\pi}{2}, \frac{\pi}{2})$	$[-\frac{\pi}{2}, 0]$,$(0, \frac{\pi}{2}]$
stop_duration	$dur_s = [0, 10]$s	$[0, 10]$
move_duration	$dur_m = [0, 10]$s	$[0, 10]$

The *initial_position* parameter includes two input domains x and y. These input domains, x and y, consist of values for initial positions with range $[0.5, 2]m$. Based on the initial distance from the robot, the range of x and y, are partitioned into three blocks of values respectively. The blocks are to ensure that obstacles in different distance ranges from the robot in the test region are covered, and there will be obstacles close to the robot to affect significantly the population of robot trajectories. The *linear_speed* consists of one input domain, v, which represents the obstacle's speed. The input domain v is partitioned into two blocks of values: one block of values that are less than the robot's maximum speed $0.25\frac{m}{s}$, and the other block of values that are greater than the robot's maximum speed. The *angular_speed* also consists of one input domain, w, with the range of values $(-\frac{\pi}{2}, \frac{\pi}{2})\frac{rad}{s}$. Similar to the *linear_speed*, the *angular_speed* is partitioned into two blocks of values for negative and positive directions of rotation respectively. Based on the maximum speed of each obstacle and the test area size, the input domains for the two duration values, dur_s and dur_m that represent *stop_duration* and *move_duration* parameters, are set to $[0, 10]s$. Each of those input domains has one block of values. For m obstacles, the number of *test-data* sets needed to satisfy ECC is $3m$. Note that a full tuple representing a test case considers all obstacles and makes sure that each obstacle's position will be unique for a test case.

5.3 Test Execution and Evaluation

We have generated test cases and conducted testing at the system level for a RAMP-H system.

First, a small test area of $2m \times 2m$ is generated. As stated before, the size of the test area is small enough to focus on interactions between the robot and obstacles but is also large enough to allow the robot and obstacles space to maneuver. The

RAMP system tested is the version described in [63]. The initial position and the goal position of the robot are at two diagonal corners of the test area (as shown in Figure 6). The initial orientation of the robot is set to face its goal position.

For testing, m=3 obstacles are considered so that there are 9 ABTCs, which can be enumerated/scripted automatically. For a scripted ABTC, test cases are also generated automatically by assigning test data based on ECC (Each Choice Criterion). Each test case runs for a maximum of 20 seconds, and the results are collected afterwards.

At the start of each test case, the obstacle initial positions are chosen randomly from the blocks of the corresponding input domains and made sure that they are at least a minimum distance of 0.2m apart from each other to avoid overlap. The initial orientation of each obstacle is set to be pointing to the robot. All obstacles remain stopped for one second so that RAMP can build a trajectory population based on the static obstacles as planning history. This is necessary because, unlike a purely reactive planner, RAMP is a global and goal-oriented planner that constantly improves and adapts the robot's motion to changes in the environment as the robot moves from a start location to a goal location. The trajectory population is also seeded with a trajectory in the direction of the robot's orientation.

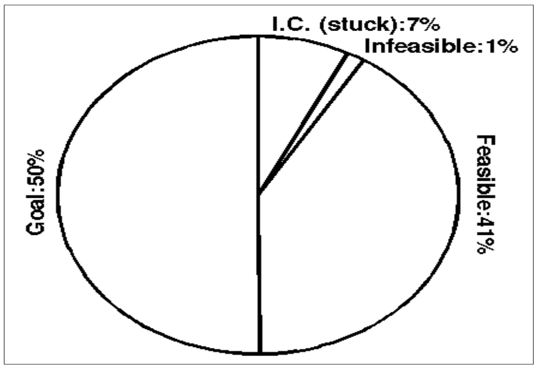
After that initial delay time has passed, the obstacles begin to move concurrently according to the ABTC for testing. All 9 ABTCs were used for automatically generating test data, and a total of 900 test cases were executed (100 test cases for each ABTC).

After each test case, the following results are recorded: *Reached goal*, a boolean

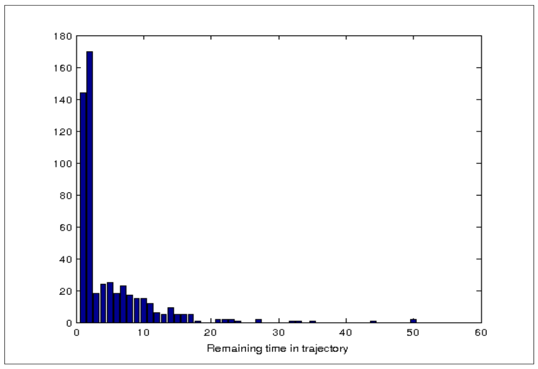
Table 11: For each ABTC, 100 test cases were executed. Each column shows the results for the specific ABTC.

ABTC	1	2	3	4	5	6	7	8	9
Tuple	<0,0,0>	<0,0,1>	<0,0,2>	<0,1,1>	<0,1,2>	<0,1,3>	<0,2,2>	<0,2,3>	<0,2,4>
% Reached Goal	59	62	55	45	45	52	43	44	46
% Feasible Traj	34	31	37	47	45	39	44	45	47
Mean Remaining Time	3.6s	4.4s	3.4s	5.4s	6.2s	3.3s	4.7s	5.0s	4.1s
% Infeasible Traj	0	5	0	1	5	2	0	1	0
% Stuck in I.C.	7	2	8	7	5	7	13	10	7

Figure 32: Overall performance of a RAMP system dealing with three mobile obstacles



(a) RAMP system performance



(b) Histogram of Remaining Time on feasible trajectories

value indicating that the robot has reached its goal within the test period, *Feasible trajectory*, a boolean value to indicate if the robot is moving on a feasible trajectory (see Section 3.1.4), *Remaining time*, the remaining time until the robot reaches the goal if the robot is moving on a feasible trajectory, *Stuck in imminent collision*, a boolean value to indicate that the robot stopped indefinitely, and *Infeasible trajectory*, a boolean value to indicate if the robot is moving on an infeasible trajectory.

Table 10 shows results for each ABTC, and Figure 32 shows the results for all executed test cases. Figure 32 shows that in 7% of the cases, the robot stops indefinitely, which is undesirable. In all other cases, which include cases when the robot pauses for imminent collision and then moves again, the robot either reaches its goal, is on a feasible trajectory to reach its goal in due time, as shown in Figure 32, or is on an infeasible trajectory, trying to get on a feasible one.

One reason that the robot gets stuck in an imminent collision state is due to an unexpected communication delay in the tested implementation of RAMP to command the robot to stop for imminent collision. This can cause the robot to stop too late and too close to the obstacle, i.e. within some inflated collision radius. If the obstacle is stopped, then RAMP will never find a new trajectory that is considered safe to move on.

Another reason is due to one or more obstacles stopped too close to the robot's goal position. In that case, the robot may not be able to get around them to reach the goal. Note that obstacle behavior models used in our testing cases are much harsher to the robot than in most real-world situations: on the one hand, our obstacle models move in constant linear and angular speeds and do not slow down when the robot gets close, and on the other hand, once they stop, they do not move away if they are blocking the robot. However, most moving obstacles in human-centered real environments, such as humans, would not behave this way.

5.4 Extended System Level Behavioral Model

The effectiveness of system-level testing depends on the ability to generate realistic world states to be used as executable test cases. This requires designing behavioral models that are both efficient and expressive. The obstacle behavioral model for system-level testing described in chapter 5 assumes that an obstacle keeps constant speed throughout an entire test case. While this model is efficient, it ignores that real-world obstacles change speed. This section shows how to extend the extended behavioral test model to account for a change in speed during a test case and shows test execution results on a RAMP system.

An extended system-level behavior model is shown in Figure 33. Compared to the model used in Section 5.2.2, the number of states for an obstacle has increased from 2 to 3, and the number of transitions has increased from 2 to 4. The structural model for dynamic obstacles is the same as described in Section 5.2.2. The obstacle internal test paths will be based on a new behavioral model (Figure 33), and the obstacle interaction test paths are systematically generated based on the method described in Section 5.2.4. The input data for a single obstacle can be seen in Table 12.

Figure 33: Obstacle behavioral model that accounts for speed changes

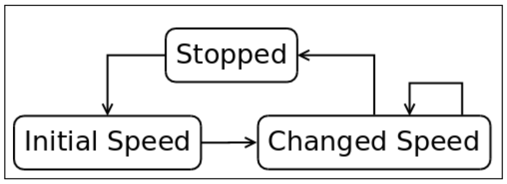


Table 12: Input domains and blocks of values of a dynamic obstacle

Parameter	**Input Domains**	**Blocks of Values**
initial_position	$x = [0.5, 1.75]m$ $y = [0.5, 1.75]m$	[0.5, 1],(1, 1.5],(1.5, 1.75] [0.5, 1],(1, 1.5],(1.5, 1.75]
linear_speed_initial	$v_i = (0, 0.5)\frac{m}{s}$	[0, 0.25],(0.25, 0.5]
linear_speed_final	$v_f = (0, 0.5)\frac{m}{s}$	[0, 0.25],(0.25, 0.5]
angular_speed_initial	$w_i = (-\frac{\pi}{2}, \frac{\pi}{2})$	$[-\frac{\pi}{2}, 0]$,$(0, \frac{\pi}{2}]$

5.4.1 Obstacle Internal Test Paths

Each obstacle has an Extended Finite State Machine (EFSM) that is a set of transitions that cover the obstacle behavioral test model (Figure 33). These EFSMs define an obstacle's internal test path. Each ith obstacle will have an internal test path that is $stop() \rightarrow move(v, w) \rightarrow move(v', w') \rightarrow stop()$, where (v, w) is a pair of initial linear and angular speeds and (v', w') is a different pair of linear and angular speeds. This internal test path will satisfy node coverage of the behavioral test model shown in Figure 33. Each internal test path will have nodes *Stopped* $\rightarrow$ *Initial Speed* $\rightarrow$ *Changed Speed* $\rightarrow$ *Stopped*.

Note that there is only one speed change in the internal test path of an obstacle. Allowing a variable number of speed changes creates a variable number of values in input domain since a new input variable is required for each speed change. Limiting the number of speed changes to one avoids this issue, and is likely more realistic since the obstacles will not have much time to change speeds many times while interacting with the robot within the small spatial and temporal neighborhood of the robot.

5.4.2 Obstacle Interaction Test Paths

Obstacle internal test paths are considered concurrent because obstacles interact concurrently. Generating interaction test paths, or Abstract Behavioral Test Cases (ABTCs), is performed with the systematic method described in Section 5.2.4. Given 4 nodes in each obstacle's internal test path (Section 5.4.1) and $m = 3$ obstacles, there are 16 ABTCs for three obstacles with this behavior model based on the method of systematically generating ABTCs described in Section 5.2.4.

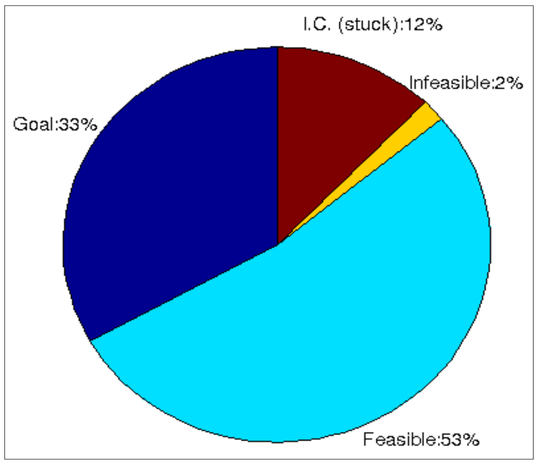
Two assumptions about obstacle behavior are being made when generating ABTCs from the model shown in Figure 33. One assumption is that the obstacles can have infinite acceleration, i.e, they can instantly switch from an initial speed to a different target speed. When they are not switching states, they have zero acceleration. The second assumption is that an obstacle will not complete two states in the time it takes previous obstacles to complete one state. For example, if the first obstacle that moves is in the *Initial Speed* state, the next obstacle that moves will at most be in the *Initial Speed* state when the first obstacle moves to its next state. To maintain this third assumption, a random duration, d_{states}, is generated to represent the maximum time each obstacle will remain in the *Initial Speed*, and *Changed Speed* states. This ensures that an obstacle will not complete multiple states in the amount of time that another obstacle completes one state while both obstacles are moving.

5.4.3 Test Execution

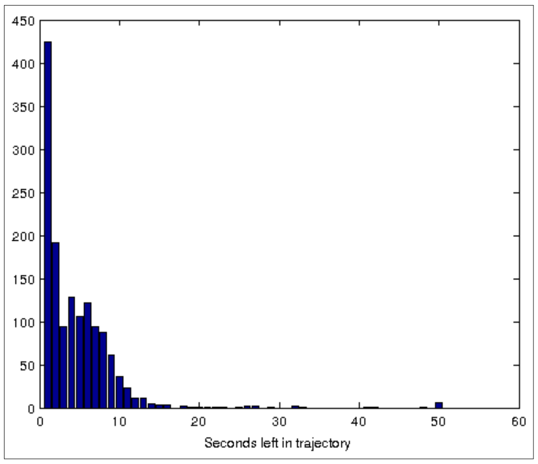
Test Execution is performed in mostly the same manner as in Section 5.3. ABTCs are manually programmed due to not having an automatic generator. For each ABTC, test-data is randomly selected and combined with the ABTC to form an executable test case. A full test-data tuple is a 1 tuple of values for the input shown in Table 12 per obstacle, and a value for d_{states} (Section 5.4.2) in range $[0.5s, 4s]$.

Obstacle positions and velocities is simulated and passed to the planner throughout the duration of a test while the robot executes its motion. The robot's motion is simulated by the Stage simulator. Similar to Section 5.3, when a test begins the obstacles are placed into their initial positions and the RAMP-H planner will initialize

Figure 34: Overall performance of a RAMP system dealing with three dynamic obstacles



(a)



(b)

a population and perform planning cycles for one second. After that initial time has passed, the obstacles begin concurrently executing their behavior (based on the ABTC) and the RAMP-H planner begins to move the robot. Results are collected after each test.

5.4.4 Test Execution Results

The first tier of tests were run with the robot moving in the Stage simulator [91] while obstacle information is published during runtime. All 16 ABTCs were manually scripted: $< 0, 0, 0 >, < 0, 0, 1 >, < 0, 0, 2 >, < 0, 0, 3 >, < 0, 1, 1 >, < 0, 1, 2 >, < 0, 1, 3 >, < 0, 1, 4 >, < 0, 2, 2 >, < 0, 2, 3 >, < 0, 2, 4 >, < 0, 2, 5 >, < 0, 3, 3 >, < 0, 3, 4 >, < 0, 3, 5 >, < 0, 3, 6 >$. For each ABTC, 100 tests were executed. Tables 13 and 14 show the results for each ABTC, and Figure 34 shows the results for all executed test cases combined.

Table 13: Results for the first 8 ABTCs. For each ABTC, 100 test cases were executed. Each column shows the results for the specific ABTC.

ABTC	1	2	3	4	5	6	7	8
Tuple	<0,0,0>	<0,0,1>	<0,0,2>	<0,0,3>	<0,1,1>	<0,1,2>	<0,1,3>	<0,1,4>
% Reached Goal	50	41	39	32	45	42	47	35
% Feasible Traj	42	48	49	49	45	47	42	52
Mean Remaining Time	2.70s	3.64s	4.13s	4.42s	3.32s	3.00s	3.71s	3.57s
% Infeasible Traj	0	0	1	2	2	0	0	0
% Stuck in I.C.	8	11	11	17	8	11	11	13

Table 14: Results for the second 8 ABTCs. For each ABTC, 100 test cases were executed. Each column shows the results for the specific ABTC.

ABTC	9	10	11	12	13	14	15	16
Tuple	<0,2,2>	<0,2,3>	<0,2,4>	<0,2,5>	<0,3,3>	<0,3,4>	<0,3,5>	<0,3,6>
% Reached Goal	22	36	17	26	26	31	19	22
% Feasible Traj	64	48	59	59	58	88	65	62
Mean Remaining Time	6.06s	4.02s	5.85s	4.96s	5.44s	4.71s	5.33s	4.67s
% Infeasible Traj	4	3	3	2	2	4	1	5
% Stuck in I.C.	10	13	21	13	14	8	15	11

Compared with the results presented in Section 5.3, the most significant change is that the number of test cases resulting in the robot reaching the goal is significantly fewer than the number from Section 5.3. This is likely related to the increase in tests that end with the robot moving on a feasible trajectory. If the robot has to change trajectories due to the obstacles changing speeds, then the robot may not have time to reach the goal. In these scenarios, the test ends with the robot moving on a feasible trajectory rather than reaching the goal before the test ends. Figure 34(b) shows a large amount of test cases ending with only a few seconds remaining in the trajectory, which indicates that the robot is close to the goal.

Another significant change is the increase in the number of test cases that end with the robot stuck in imminent collision. Although the obstacle's direction will not change when its speed changes, the feasibility of a trajectory can change if an obstacle's change in speed is large enough. This can cause the robot to stop for imminent collision, but not quickly finding a better trajectory.

CHAPTER 6: CONCLUSIONS AND FUTURE WORK

Autonomous robot navigation is a fundamental problem in robotics that combines research issues in path and trajectory planning, motion control, and sensing and perception. Solving the problem in an unknown and uncertain environment is to enable a robot to sense changes in an environment and adapt its path/trajectory accordingly on the fly while executing a motion smoothly that satisfies various optimization constraints.

Much existing research is focused on addressing certain aspects of the problem, but not the whole problem. This dissertation has presented a general and holistic approach for autonomous non-holonomic robot navigation in an unknown environment with arbitrary static or dynamic obstacles, and a novel model-based approach for system-level test generation to measure the effectiveness of a system that implements autonomous robot motion in the presence of obstacles that move in unforeseen ways.

While other recent approaches to motion planning focus on learning about the specific obstacles in the environment, the RAMP-H approach maintains generality when representing obstacles. It assumes very little about the obstacles that the robot may need to avoid, and represents them all in a generic way. It bypasses computationally expensive object classification in order to focus on adapting the robot's motion to avoid whatever oncoming obstacle it senses.

6.1 Contributions

The RAMP approach has been extended to enable non-holonomic motion and to incorporate real-time sensing and perception from real sensing data. The presented planner, RAMP-H, converts holonomic trajectories to non-holonomic motion on-the-fly. It utilizes Bézier curves to connect straight-line segments along a trajectory, and to generate motion to switch trajectories via smooth non-holonomic curves. Control cycle frequency was modified so that it adapts to the robot's dynamics and the need to avoid obstacles. A method to sync control and planning at real-time was introduced into a RAMP system to account for motion error during execution.

Real-time sensing and perception was achieved by modeling dynamic obstacles with efficient circle representations to allow for fast collision detection and the ability to represent any arbitrary shape. This real-time sensing and perception module allows RAMP-H to detect and track unknown obstacles that may be any shape, move at any speed, and may enter or leave the environment throughout execution.

The Hilbert maps framework [76] was utilized to learn the occupancy of an environment and incorporate the knowledge of past executions into the real-time execution of RAMP-H. During runtime, the learned information is combined with real sensing data, and both are considered in the evaluation of trajectories. Essentially, this work embeds high-level decision making about navigating an environment into the evaluation function of RAMP-H to avoid regions of high likelihood of occupancy which leads to smoother and safer motion.

The RAMP-H framework has been validated with physical robot experiments. The experiments demonstrate that the framework is capable of reaching a goal position while adapting to unknown arbitrary static and dynamic obstacles whose changes are only known by real-time detection and tracking based on real sensing data. Details about the frequencies and duration of various procedures in RAMP-H have been presented. Further, leveraging past experience in motion planning was validated by both simulation and physical robot experiments showing that regions with a high likelihood of occupancy in an environment can be avoided by incorporating learned data about an environment's occupancy into RAMP-H's evaluation function.

An approach to generate system-level test cases for a robot motion planning system that moves a robot in the presence of dynamic obstacles with unforeseen motion was introduced, and test execution was performed on a RAMP-H system. The approach models the interactions between a robot and obstacles so that the number of possible Abstract Behavior Test Cases (ABTCs) is drastically reduced. System-level testing of a RAMP system offers insight for evaluating an implementation that cannot be obtained otherwise. First, system-level tests help assess issues in overall system performance, such as getting stuck in an imminent collision state. Secondly, these tests can identify bottlenecks in a system to show how often a certain problem occurs. Third, system-level tests can measure exact performance differences after changes in a system's implementation, such as GPU optimization. Lastly, the tests systematically validate that specific system-level requirements are satisfied by a system. For instance, before putting a robot in a real environment with dynamic obstacles moving in unpredictable ways, the system-level test suite can check if the robot can reach its goal some percentage of the time in a simulated version of the environment.

6.2 Future Work

The RAMP-H framework is a powerful modular system that can be extended and built upon both on the theoretical level and the implementation. Different methods for perception, collision checking, and trajectory generation can be used in the framework. Additionally, more learning-based methods can be utilized in the framework to improve the overall performance.

6.2.1 Learning

Incorporating more learning into the framework is an area with much future work. Currently the evaluation function is set empirically after viewing the planner's performance in an environment. However, utilizing machine learning techniques to optimize the evaluation function for different environments can likely provide an optimized and more effective set of coefficients.

Currently, obstacle trajectories are predicted as maintaining constant velocity for a small period of time (one sensing cycle). This naive approach works well in practice, but it is likely that the overall performance can be improved by learning the motion patterns for an obstacle type and using that to more accurately predict obstacle motion, as some of the human-aware motion planning literature does.

Learning the occupancy of an environment can be researched further. One open question is how to update our model of the environment with sensor reading from navigation. Currently the model is trained on data covering the entire environment so it cannot handle occlusions in the training data. Updating the model on only the regions a robot can view at a certain point in time can allow the model to be updated and trained based on actual navigation results rather than needing to view the entire environment to obtain training data.

6.2.2 Testing

An extremely time-consuming component of system-level testing for a robot motion system is test execution. In order to reduce the time it takes to do test execution, a fewer number of executable test cases should be run. Test cases that are considered *edge cases* are ones where the system has a high chance of failing, such as a test case where the obstacles breach a clearance radius for the robot. These cases should be identified so that only those test cases can be run, and other cases can be assumed to either pass or fail. This can significantly reduce the time spent in test execution while maintaining reasonably accurate results. However, identifying edge cases is nontrivial due to the deterministic nature of RAMP.

The obstacle behavioral models should be extended further. One aspect of obstacle behavior not considered here is acceleration. Considering acceleration in the behavior model may tie the ABTCs to the input data because the values for acceleration will affect when an obstacle transitions between states that are classified by speed, such as *Constant Speed*, *Increased Speed*, etc. Another limitation of the behavior model is the assumption that all transitions have equal likelihood of occurring. Using a probabilistic model to determine how likely an obstacle is to change states may lead to more accurate behaviors.

The number of ABTCs for an obstacle will grow as the obstacle behavior models become more complex. Currently, all ABTCs are manually scripted because there has been a low number. However, an automatic scripting tool for implementing ABTCs will be necessary to achieve test execution when the number of ABTCs grows too large.

6.2.3 Implementation

There are many identical tasks that are run in the RAMP framework that can be sped up with parallel computing techniques. For instance, each control cycle requires that each trajectory in the population be updated with a new current state and requires some trajectory re-planning. Currently the trajectories are updated sequentially, but it can almost certainly be sped up with parallel computing. Further, it may be possible to maintain many populations of trajectories in parallel in order to increase a robot's number of options and diversity in those options.

The RAMP-H framework was implemented as a ROS metapackage. Each component (planner, control, trajectory generation, trajectory evaluation, path modification, and sensing and perception) runs in a separate ROS node, which is a separate process. The advantage of this method is that it allows components to be easily replaced with other components. The disadvantage is that it introduces network latency to pass data between all these processes. This latency can have a significant effect on the overall performance of the system, so future work can attempt to remove this issue. One option is to couple some of these nodes to remove the message passing between them. Another option is to migrate the system to ROS 2.0 and make use of the nodelet concept, which allows nodes on the same computer to pass pointers to data rather than the data itself to significantly speed up message passing.

6.3 Applications

This dissertation presented a general approach for robot motion planning in the presence of unknown obstacles. This approach is potentially useful for any application in an unstructured environment where a robot does not know how the other entities in an environment will behave. Common unstructured indoor environments include homes, restaurants, hospitals, and airports. All of these environments require a robot to navigate around human obstacles that can move in unpredictable ways. This work may also be used in environments where robots and humans collaborate to achieve tasks, such as in warehouse environments, or even environments containing only robots if the robots do not move on fixed paths. Outdoor environment applications, such as delivery robots, may make use of this work since the outside world is usually unstructured.

Measuring the effectiveness of an autonomous robotic system's effectiveness has many applications. For example, one can measure a system's reliability in a specific environment to verify that strict performance requirements can be met before a robot is introduced into the environment. Another application is measuring the change in performance after modifying part(s) of the code, such as implementing optimization or replacing modules. Further, system-level tests can elicit areas of improvement and/or identify bottlenecks in a system.

REFERENCES

[1] M. Abdelgawad, S. McLeod, A. Andrews, and J. Xiao. Model-based testing of Real-time Adaptive Motion Planning (RAMP). In *IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, 2016.

[2] M. Abdelgawad, S. McLeod, A. Andrews, and J. Xiao. Model-based testing of a real-time adaptive motion planning system. *Advanced Robotics*, 31(22):1159–1176, 2017.

[3] N. M. Amato, O. B. Bayazit, L. K. Dale, C. Jones, and D. Vallejo. OBPRM: An obstacle-based PRM for 3D workspaces, 1998.

[4] P. Ammann and J. Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[5] A. Andrews, M. Abdelgawad, and A. Gario. Active world model for testing autonomous systems using CEFSM. In *Workshop on Model-Driven Engineering, Verification and Validation*, 2015.

[6] A. Andrews, M. Abdelgawad, and A. Gario. Towards world model-based test generation in autonomous systems. In *International Conference on Model-Driven Engineering and Software Development*, pages 1–12. SciTePress Digital Library, 2015.

[7] A. Andrews, M. Abdelgawad, and A. Gario. World model for testing autonomous systems using Petri nets. In *2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE)*, pages 65–69, Jan 2016.

[8] A. Andrews, M. Abdelgawad, and A. Gario. World model for testing urban search and rescue (USAR) robots using Petri nets. In *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 663–670, 2016.

[9] F. Aurenhammer. Voronoi diagrams a survey of a fundamental geometric data structure. *Association of Computing Machinery Computing Surveys (CSUR)*, 23(3):345–405, 1991.

[10] J. Barraquand and J.-C. Latombe. Nonholonomic multibody mobile robots: Controllability and motion planning in the presence of obstacles. *Algorithmica*, 10(2-4):121–155, 1993.

[11] R. Bohlin and L. E. Kavraki. Path planning using lazy PRM. In *Proceedings of the IEEE International Conference on Robotics and Automation, 2000.*, volume 1, pages 521–528.

[12] V. Boor, M. H. Overmars, and A. F. van der Stappen. The Gaussian sampling strategy for probabilistic roadmap planners. In *Proceedings of the IEEE International Conference on Robotics and Automation, 1999.*

[13] D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the Association of Computing Machinery*, 30(2):323–342, Apr. 1983.

[14] O. Brock and E. Kavraki. Decomposition-based motion planning: A framework for real-time motion planning in high-dimensional configuration spaces. In *Proceedings of the IEEE International Conference on Robotics and Automation, 2001*, volume 2, pages 1469–1474.

[15] O. Brock and O. Khatib. Elastic strips: A framework for motion generation in human environments. *The International Journal of Robotics Research*, 21(12):1031–1052, 2002.

[16] J. Bruce and M. Veloso. Real-time randomized path planning for robot navigation. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, 2002.*, volume 3, pages 2383–2388.

[17] W. Burgard, A. B. Cremers, D. Fox, D. Hähnel, G. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun. Experiences with an interactive museum tour-guide robot. *Artificial Intelligence*, 114(1-2):3–55, Oct. 1999.

[18] T. Campbell, M. Liu, B. Kulis, J. P. How, and L. Carin. Dynamic clustering via asymptotics of the dependent Dirichlet process mixture. In *Advances in Neural Information Processing Systems*, pages 449–457, 2013.

[19] B. Chazelle. Approximation and decomposition of shapes. *Algorithmic and Geometric Aspects of Robotics*, 1:145–185, 1985.

[20] Y. F. Chen, M. Everett, M. Liu, and J. P. How. Socially aware motion planning with deep reinforcement learning. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and System, 2017.*

[21] Y. F. Chen, S.-Y. Liu, M. Liu, J. Miller, and J. P. How. Motion planning with diffusion maps. In *Proceedings of IEEE/RSJ International Conference on Robots and Systems, 2016.*

[22] W. Cheng, Z. Tang, C. Zhao, L. Tang, and Z. Guo. Path planning for nonholonomic car-like mobile robots using genetic algorithms. In *2006 8th International Conference on Signal Processing*, volume 4, 2006.

[23] J.-W. Choi and K. Huhtala. Constrained path optimization with Bézier curve primitives. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems, 2014*, pages 246–251.

[24] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking.* MIT press, 1999.

[25] ClearpathRobotics. *Turtlebot 2 Platform.* `https://www.clearpathrobotics.com/turtlebot-2-open-source-robot/`.

[26] E. Coumans et al. Bullet physics library. *Open source: bulletphysics. org*, 15, 2013.

[27] M. De Berg, M. Van Kreveld, M. Overmars, and O. C. Schwarzkopf. Visibility graphs. In *Computational Geometry*, pages 307–317. Springer, 2000.

[28] A. Dias-Neto, R. Subramanyan, M. Vieira, and G. H. Travassos. A survey on model-based testing approaches: A systematic review. In *Association of Computing Machinery International Workshop on Empirical Assessment of Software Engineering Languages and Technology*, WEASELTech '07, pages 31–36, New York, NY, USA, 2007.

[29] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[30] A. Elfes. Using occupancy grids for mobile robot perception and navigation. *Computer*, 22(6):46–57, 1989.

[31] G. Erinc and S. Carpin. A genetic algorithm for nonholonomic motion planning. In *Proceedings of IEEE International Conference on Robotics and Automation, 2007*, pages 1843–1849, April.

[32] D. Ferguson, N. Kalra, and A. Stentz. Replanning with RRTs. In *Proceedings of IEEE International Conference on Robotics and Automation, 2006*, pages 1243–1248.

[33] X.-S. Ge, H. Li, and Q.-Z. Zhang. Nonholonomic motion planning of space robotics based on the genetic algorithm with wavelet approximation. In *Proceedings of IEEE International Conference on Control and Automation, 2007*, pages 1977–1980, May 2007.

[34] G. Grisetti, C. Stachniss, and W. Burgard. Improved techniques for grid mapping with Rao-Blackwellized particle filters. *IEEE Transactions on Robotics*, 23(1):34–46, 2007.

[35] J. Guan and J. Offutt. A model-based testing technique for component-based real-time embedded systems. In *IEEE International Conference on Software Testing, Verification, and Validation Workshops, 2015.*

[36] D. Hsu, R. Kindel, J.-C. Latombe, and S. Rock. Randomized kinodynamic motion planning with moving obstacles. *The International Journal of Robotics Research*, 21(3):233–255, 2002.

[37] D. Jung and P. Tsiotras. On-line path generation for small unmanned aerial vehicles using B-spline path templates. In *American Institute of Aeronautics and Astronautics Guidance, Navigation and Control Conference*, volume 7135, 2008.

[38] R. E. Kalman et al. A new approach to linear filtering and prediction problems. *Journal of Basic Engineering*, 82(1):35–45, 1960.

[39] S. Karaman and E. Frazzoli. Sampling-based algorithms for optimal motion planning. *The International Journal of Robotics Research*, 30(7):846–894, 2011.

[40] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580, 1996.

[41] A. Kelly and B. Nagy. Reactive nonholonomic trajectory generation via parametric optimal control. *The International Journal of Robotics Research (IJRR)*, 22(7-8):583–601, 2003.

[42] H. Khambhaita and R. Alami. Viewing robot navigation in human environment as a cooperative activity. In *International Symposium on Robotics Research (ISRR)*, 2017.

[43] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *The International Journal of Robotics Research*, 5(1):90–98, 1986.

[44] N. Koenig and A. Howard. Design and use paradigms for Gazebo, an open-source multi-robot simulator. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systemse, 2004*.

[45] S. Koenig and M. Likhachev. Fast replanning for navigation in unknown terrain. *IEEE Transactions on Robotics*, 21(3):354–363, 2005.

[46] H. Kretzschmar, M. Spies, C. Sprunk, and W. Burgard. Socially compliant mobile robot navigation via inverse reinforcement learning. *The International Journal of Robotics Research*, 35(11):1289–1307, 2016.

[47] T. Kröger and F. M. Wahl. Online trajectory generation: Basic concepts for instantaneous reactions to unforeseen events. *IEEE Transactions on Robotics*, 26(1):94–111, 2010.

[48] F. Lamiraux, D. Bonnafous, and O. Lefebvre. Reactive path deformation for nonholonomic mobile robots. *IEEE Transactions on Robotics*, 20(6):967–977, 2004.

[49] J.-P. Laumond. Feasible trajectories for mobile robots with kinematic and environment constraints. In *Proceedings of International Conference on Intelligent Autonomous Systems*, pages 346–354, 1986.

[50] J.-P. Laumond, P. Jacobs, M. Taix, and R. Murray. A motion planner for nonholonomic mobile robots. *IEEE Transactions on Robotics and Automation*, 10(5):577–593, Oct 1994.

[51] J. Laval, L. Fabresse, and N. Bouraqadi. A methodology for testing mobile autonomous robots. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems, 2013.*

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

[53] S. M. LaValle and J. J. Kuffner. Randomized kinodynamic planning. *The International Journal of Robotics Research*, 20(5):378–400, 2001.

[54] J. Levinson, J. Askeland, J. Becker, J. Dolson, D. Held, S. Kammel, J. Z. Kolter, D. Langer, O. Pink, V. Pratt, et al. Towards fully autonomous driving: Systems and algorithms. In *Proceedings of Intelligent Vehicles Symposium (IV), 2011 IEEE*, pages 163–168.

[55] Y. Li and J. Xiao. On-line planning of nonholonomic trajectories in crowded and geometrically unknown environments. In *Proceedings of IEEE International Conference on Robotics and Automation, 2009*, pages 3230–3236.

[56] R. Lill and F. Saglietti. Testing the cooperation of autonomous robotic agents. In *Proceedings of IEEE International Conference on Software Engineering and Applications (ICSOFT-EA), 2014*, pages 287–296.

[57] A. X. P. LIVE. *Specifications.* `https://www.asus.com/us/3D-Sensor/Xtion_PRO_LIVE/specifications/`.

[58] D. V. Lu, D. Hershberger, and W. D. Smart. Layered costmaps for context-sensitive navigation. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems, 2014*, pages 709–715.

[59] J. MacDougall and G. Tewolde. Tour guide robot using wireless based localization. In *Proceedings of the IEEE International Conference on Electro/Information Technology (EIT)*, pages 1–6, May 2013.

[60] R. MacLachlan and C. Mertz. Tracking of moving objects from a moving vehicle using a scanning laser rangefinder. In *2006 IEEE Intelligent Transportation Systems Conference*, pages 301–306.

[61] E. Marder-Eppstein. *ROS navigation stack*, 2017. `https://github.com/ros-planning/navigation`.

[62] E. Marder-Eppstein, E. Berger, T. Foote, B. Gerkey, and K. Konolige. The office marathon: Robust navigation in an indoor office environment. In *Proceedings of IEEE International Conference on Robotics and Automation, 2010.*

[63] S. McLeod and J. Xiao. Real-time adaptive non-holonomic motion planning in unforeseen dynamic environments. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2016.

[64] S. McLeod and J. Xiao. Autonomous robot navigation in unknown dynamic environments. *Submitted to IEEE Transactions on Robotics*, 2018.

[65] S. McLeod and J. Xiao. Navigating dynamically unknown environments leveraging past experience. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2019.

[66] C. Mertz, L. E. Navarro-Serment, R. MacLachlan, P. Rybski, A. Steinfeld, A. Suppe, C. Urmson, N. Vandapel, M. Hebert, C. Thorpe, et al. Moving object detection with laser scanners. *Journal of Field Robotics*, 30(1):17–43, 2013.

[67] N. Metropolis and S. Ulam. The Monte Carlo method. *Journal of the American Statistical Association*, 44(247):335–341, 1949.

[68] J. Miller, A. Hasfura, S.-Y. Liu, and J. P. How. Dynamic arrival rate estimation for campus mobility on demand network graphs. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, 2016*, pages 2285–2292.

[69] M. Montemerlo, J. Becker, S. Bhat, H. Dahlkamp, D. Dolgov, S. Ettinger, D. Haehnel, T. Hilden, G. Hoffmann, B. Huhnke, et al. Junior: The Stanford entry in the urban challenge. *Journal of Field Robotics*, 25(9):569–597, 2008.

[70] M. Montemerlo, S. Thrun, D. Koller, B. Wegbreit, et al. Fastslam: A factored solution to the simultaneous localization and mapping problem. In *Association for the Advancement of Artificial (AAAI) Intelligence Innovative Applications of Artificial Intelligence Conference (IAAI)*, pages 593–598, 2002.

[71] R. Murray and S. Sastry. Nonholonomic motion planning: steering using sinusoids. *IEEE Transactions on Automatic Control*, 38(5):700–716, May 1993.

[72] C. Nissoux, T. Siméon, and J.-P. Laumond. Visibility based probabilistic roadmaps. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems, 1999.*, volume 3, pages 1316–1321.

[73] M. Pfeiffer, M. Schaeuble, J. Nieto, R. Siegwart, and C. Cadena. From perception to decision: A data-driven approach to end-to-end motion planning for autonomous ground robots. In *Proceedings of IEEE International Conference on Robotics and Automation, 2017*, page 15271533.

[74] M. Pfeiffer, U. Schwesinger, H. Sommer, E. Galceran, and R. Siegwart. Predicting actions to act predictably: Cooperative partial motion planning with maximum entropy models. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems, 2016*, pages 2096–2101.

[75] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. ROS: an open-source Robot Operating System. In *Proceedings of IEEE International Conference on Robotics and Automation Workshop on Open Source Software*, volume 3, page 5. Kobe, 2009.

[76] F. Ramos and L. Ott. Hilbert maps: Scalable continuous occupancy mapping with stochastic gradient descent. *The International Journal of Robotics Research*, 35(14):1717–1730, 2016.

[77] J. Reeds and L. Shepp. Optimal paths for a car that goes both forwards and backwards. *Pacific Journal of Mathematics*, 145(2):367–393, 1990.

[78] M. I. Ribeiro. Kalman and extended Kalman filters: Concept, derivation and properties. *Institute for Systems and Robotics*, 43, 2004.

[79] R. Senanayake and F. Ramos. Bayesian Hilbert maps for dynamic continuous occupancy mapping. In *1st Annual Conference on Robot Learning (CoRL)*, 2017.

[80] J. Snape, J. van den Berg, S. Guy, and D. Manocha. The hybrid reciprocal velocity obstacle. *IEEE Transactions on Robotics*, 27(4):696–706, Aug 2011.

[81] A. Stentz. Optimal and efficient path planning for partially-known environments. In *Proceedings of IEEE International Conference on Robotics and Automation, 1994.*, pages 3310–3317.

[82] A. Stentz et al. The focussed D* algorithm for real-time replanning. In *Proceedings of the International Joint Conference on Artificial Intelligence, 1995*, volume 95, pages 1652–1659.

[83] L. Tai, G. Paolo, and M. Liu. Virtual-to-real deep reinforcement learning: Continuous control of mobile robots for mapless navigation. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems, 2017*, pages 31–36.

[84] S. Thrun, M. Bennewitz, W. Burgard, A. Cremers, F. Dellaert, D. Fox, D. Hahnel, C. Rosenberg, N. Roy, J. Schulte, and D. Schulz. MINERVA: a second-generation museum tour-guide robot. In *Proceedings of the IEEE International Conference on Robotics and Automation, 1999*, volume 3, pages 1999–2005 vol.3.

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

[86] S. Thrun, M. Montemerlo, H. Dahlkamp, D. Stavens, A. Aron, J. Diebel, P. Fong, J. Gale, M. Halpenny, G. Hoffmann, et al. Stanley: The robot that won the DARPA grand challenge. In *The 2005 DARPA Grand Challenge*, pages 1–43. Springer, 2007.

[87] P. Trautman and A. Krause. Unfreezing the robot: Navigation in dense, interacting crowds. In *Proceedings of IEEE/RSJ International Conference on Robots and Systems, 2010*, pages 797–803.

[88] C. Urmson, J. A. Bagnell, C. R. Baker, M. Hebert, A. Kelly, R. Rajkumar, P. E. Rybski, S. Scherer, R. Simmons, S. Singh, et al. Tartan racing: A multi-modal approach to the DARPA urban challenge. Technical report, 2007.

[89] J. Vannoy and J. Xiao. Real-time Adaptive Motion Planning (RAMP) of mobile manipulators in dynamic environments with unforeseen changes. *IEEE Transactions on Robotics*, 24(5):1199–1212, 2008.

[90] A. Vasquez, M. Kollmitz, A. Eitel, and W. Burgard. Deep detection of people and their mobility aids for a hospital robot. In *2017 European Conference on Mobile Robots (ECMR)*, pages 1–7. IEEE, 2017.

[91] R. Vaughan. Massively multi-robot simulation in Stage. *Swarm intelligence*, 2(2-4):189–208, 2008.

[92] S. A. Wilmarth, N. M. Amato, and P. F. Stiller. MAPRM: A probabilistic roadmap planner with sampling on the medial axis of the free space. In *Proceedings of IEEE International Conference on Robotics and Automation, 1999.*, volume 2, pages 1024–1031.

[93] J. Xavier, M. Pacheco, D. Castro, A. Ruano, and U. Nunes. Fast line, arc/circle and leg detection from laser scan data in a Player driver. In *Proceedings of IEEE International Conference on Robotics and Automation, 2005.*, pages 3930–3935.

[94] J. Xiao, Z. Michalewicz, L. Zhang, and K. Trojanowski. Adaptive evolutionary planner/navigator for mobile robots. *IEEE Transactions on Evolutionary Computation*, 1(1):18–28, 1997.

[95] Y. Yang and O. Brock. Elastic roadmaps motion generation for autonomous mobile manipulation. *Autonomous Robots*, 28(1):113–130, 2010.

[96] A. Yershova, L. Jaillet, T. Siméon, and S. M. LaValle. Dynamic-domain RRTs: Efficient exploration by controlling the sampling domain. In *Proceedings of IEEE International Conference on Robotics and Automation, 2005*, pages 3856–3861.

[97] Z. Zhang. Microsoft Kinect sensor and its effect. *IEEE multimedia*, 19(2):4–10, 2012.