REAL TIME MOTION PLANNING FOR PATH COVERAGE WITH APPLICATIONS IN OCEAN SURVEYING

BY

Alexander F. Brown

BS of Computer Science, University of New Hampshire, 2018

THESIS

Submitted to the University of New Hampshire
in Partial Fulfillment of
the Requirements for the Degree of

Master of Science

in

Computer Science

September, 2020

ALL RIGHTS RESERVED

©2020

Alexander F. Brown

This thesis was examined and approved in partial fulfillment of the requirements for the degree of Master of Science in Computer Science by:

Thesis director, Wheeler Ruml,
Professor of Computer Science
University of New Hampshire

Val Schmidt,
Research Engineer
Center for Coastal and Ocean Mapping,
University of New Hampshire

Momotaz Begum,
Assistant Professor of Computer Science
University of New Hampshire

On August 3, 2020

Approval signatures are on file with the University of New Hampshire Graduate School.

TABLE OF CONTENTS

LIST OF TABLES . . . vi

LIST OF FIGURES . . . xii

ABSTRACT . . . xiii

Chapter 1 Introduction **1**

1.1 Motivation . . . 1

1.2 Ocean Surveying . . . 2

1.3 Path Coverage . . . 3

1.4 Overview . . . 6

1.5 Contributions . . . 7

Chapter 2 Related Work **8**

Chapter 3 An Algorithm for Path Coverage **12**

3.1 Features of Path Coverage that Impact Algorithm Design . . . 12

3.2 Algorithm Description . . . 14

3.3 Biasing Sampling and Search . . . 19

3.4 Theoretical Properties . . . 29

Chapter 4 Approximate Model Predictive Controller **40**

Chapter 5 Architecture for Autonomous Ocean Surveying **45**

5.1 Executive . . . 45

5.2 Project11 . . . 45

Chapter 6 Experimental Results **49**

6.1 Experimental Setup . . . 49

6.2 Comparing the Planners . . . 50

6.3 Experimenting with Parameters . . . 57

Chapter 7 Conclusion **65**

7.1 Discussion . . . 65

7.2 Lessons Learned . . . 67

7.3 Further Research . . . 69

7.4 Conclusions . . . 71

Appendices **77**

Chapter A Discussion of Parameters **78**

A.1 Planner Parameters . . . 78

A.2 Controller Parameters . . . 81

A.3 Simulation Parameters . . . 83

Chapter B Descriptions of Scenarios **84**

B.1 Specific Behavior Scenarios . . . 84

B.2 Line Following Scenarios . . . 105

LIST OF TABLES

6-1 Number of completed instances out of 10 trials for several scenarios comparing RBPC and Potential Field 51

6-2 Percentage of plans from each algorthm which the controller deemed feasible. . . 53

LIST OF FIGURES

1-1 Diagram illustrating ocean surveying, the motivation for the path coverage problem. The ASV's pose is shown with several trajectories leaving it, marked either impossible, unsafe, reasonable, or best. The best trajectory is marked as such because it achieves some coverage of the survey line while avoiding obstacles. The unsafe trajectory passes in front of another vessel, which is undesirable. The impossible trajectory attempts to pilot through an island. 4

3-1 A slice at $\theta = \pi$ of the partition into word-invariant cells for the Dubins car. The circle is centered on the origin. Figure from LaValle (2006). 21

3-2 Two slices of path type partitions with different starting headings. 21

3-3 Dubins path shown with an incorrect re-computed suffix. This demonstrates numerical instability in Dubins curves. 22

3-4 A coverage path vs a Dubins path to the start pose for beginning line coverage. When a large-radius turn can be completed within the line width it is far better to do so than to take a Dubins path to the center endpoint of the line. 24

3-5 An example robust coverage trajectory in more detail. It consists of two curves: a turn inwards, towards the center of the line, and a turn outwards, for the final alignment. When both curves can be completed within the line width this is desirable, as no portion of the segment will be left uncovered. 25

3-6 ASV is headed away from the center of the path segment. It executes a turn towards the center of the segment followed by a turn in the opposite direction, ending aligned with the center line. 26

3-7 ASV is headed towards the center of the path segment. By executing a turn away from the center line, it would not cross the center, so it turns first to the left, towards the center line. Then, it turns to the right, ending aligned with the center line. 27
3-8 ASV is headed towards the center of the path segment. The ASV turns to the right, away from the center line, but still crosses it. It follows up with a turn in the opposite direction, ending aligned with the center line. 28

5-1 User interface for controlling autonomous vehicles in Project 11. 46
5-2 Overview of communication between ROS nodes in the relevant portion of Project 11. 47

6-1 Plot showing score of the potential field planner and of the RBPC planner. . . . 52
6-2 Plot showing percentage of plans from each algorithm which the controller deemed feasible. 54
6-3 Plot showing score of the potential field planner and of RBPC in more realistic ocean surveying instances instances. 55
6-4 Maze map. Grey squares are blocked. 56
6-5 Satellite image of Pepperrell Cove. The moored boats in the harbor present a navigational challenge for an ASV. 58
6-6 Pepperrell Cove scenario visualized in CAMP. Static obstacles, shown in grey, are taken from the satellite image to represent the coastline and moored boats in the harbor. Survey lines appear in the East and West, and the ASV starts the scenario next to Frisbee Wharf. 58
6-7 Plot showing scores with different dynamic obstacle representations in relevant scenarios. 59
6-8 Plot showing score with different time horizons. 60
6-9 Plot showing score with slowing down allowed or not allowed. 61
6-10 Plot showing score with two different turning radii. 62

6-11 Plot showing percentages of achievable plans with two different turning radii. . . 63
6-12 Plot showing scores in a single scenario with different heuristics 64

B-1 100mNorthGauntlet: ASV is positioned below a survey line with many dynamic obstacles approaching from both sides of the line. It must slow to avoid collision. This scenario tests slowing down to avoid obstacles, avoiding obstacles in general, and how several obstacles affect behavior (more obstacles implies more computational effort). 85
B-2 adjacent_multi_line: ASV is positioned to the left of three parallel survey lines. This scenario tests handling of multiple lines. 86
B-3 adjacent_single_line: ASV is positioned to the left of a single survey line, partway through it. This scenario tests how the planner decides to begin to cover the line. 87
B-4 ahead_long: ASV is positioned below a survey line. This scenario tests staying on a line that was easy to start. 88
B-5 cannon_crossfire: ASV starts below a horizontal survey line with three dynamic obstacles of differing speeds crossing its path. This scenario tests handling of multiple obstacles cutting across the ASV's path, and slowing to avoid obstacles. 89
B-6 cannon_cut_across: ASV starts below a horizontal survey line with two dynamic obstacles headed across the line as well. This scenario tests handling multiple obstacles interrupting a survey line, and slowing to avoid obstacles. 90
B-7 cannon_follow: ASV starts below a dynamic obstacle with a horizontal survey line also above. The static map blocks a large portion of the middle of the space. This scenario tests following a dynamic obstacle at a safe distance. 91
B-8 cannon_go_around: ASV starts below a large, slow dynamic obstacle with a horizontal survey line also above. The static map blocks a large portion of the midle of the space. This scenario tests the ability to discover a faster route going around the island in the middle of the map to reach the survey line, as opposed to following the dynamic obstacle. 92

B-9 cannon_wait_1: ASV is positioned below a survey line blocked by a large dynamic obstacle, moving slowly towards the ASV. The static map allows two corridors: one upwards from the ASV, containing both the survey line and the dynamic obstacle, and a second to the right of the ASV. This scenario tests the ability to wait in the open corridor for the dynamic obstacle to safely pass by. 93

B-10 cannon_wait_2: ASV is positioned below a survey line blocked by a small dynamic obstacle, moving towards the ASV. The static map allows two large corridors, one above the ASV, containing the survey line and the dynamic obstacle, and a second short one to the right of the ASV. This scenario tests the ability to avoid the small obstacle on the way to the survey line. 94

B-11 dynamic_wall_1: ASV is positioned to the left of a horizontal survey line. The path between them is blocked by three unmoving dynamic obstacles. This scenario tests the ability to avoid unmoving dynamic obstacles. 95

B-12 long_single_line: ASV starts to the left of a long survey line. This scenario tests the ability to follow a line over a longer period of time. 95

B-13 Test01: ASV is positioned below a short survey line. This scenario tests basic short-term line following. 96

B-14 Test02a: ASV starts below two connected survey lines with a wide angle between them. This scenario tests the ability to plan around a slight corner in a survey. . 97

B-15 Test02b: ASV starts below two connected survey lines with a narrower angle between them than in Test02a (Figure B-14). This scenario tests the ability to plan around a moderate corner in a survey. 98

B-16 Test03: ASV starts below a short survey line with a single stationary dynamic obstacle to the right of the line. This scenario tests the ability to ignore the obstacle, which is a safe distance away for its size and direction, and cover the line. 99

B-17 Test04: ASV is positioned below a short survey line with a stationary dynamic obstacle in the middle of the line. This scenario tests the ability to trade off the mission goals for safety, as the survey cannot be completed safely. 100

B-18 Test12a: ASV is positioned below a survey line passing through a static obstacle. This scenario tests the ability to navigate around the static obstacle to complete as much of the survey as possible. 101
B-19 Test12b: ASV is positioned below a static obstacle forming a local minimum in all implemented heuristics. This scenario tests the ability to navigate outside of such a minimum, and is possible using a reasonable time horizon. 102
B-20 Test12c: ASV is positioned below a survey line passing through a large static obstacle. This scenario tests the ability to navigate around the static obstacle to complete as much of the survey as possible. 103
B-21 Test12d: ASV is positioned below a large static obstacle forming a local minimum in all implemented heuristics. This scenario tests the ability to navigate outside of such a minimum, but is too large to be possible using any time horizon feasible for RBPC as presented in this thesis. 104
B-22 narrow_wide: ASV must pass through a narrow corridor before navigating around a seawall to a short survey line. 105
B-23 Test05. ASV must navigate along a survey line with a moving obstacle from the right which requires 50m (a), 100m (b), 300m (c), 500m (d), and 2000m (e) avoidance. A video of RBPC running in the Test05 scenario can be found here: `https://youtu.be/hioDkJkH2vM` 106
B-24 Test06. ASV must navigate along a survey line with a moving obstacle from the right and left which require 50m (a), 100m (b), 300m (c), 500m (d), and 2000m (e) avoidance. A video of RBPC running in the Test06 scenario can be found here: `https://youtu.be/fhztkAFL7V4` 107
B-25 Test07. ASV must navigate along a survey line with an oncoming obstacle to the right of the line which requires 50m (a), 100m (b), 300m (c), 500m (d), and 2000m (e) avoidance. A video of RBPC running in the Test07 scenario can be found here: `https://youtu.be/RHPIoS2MSgM` 108

B-26 Test08. ASV must navigate along a survey line with an oncoming obstacle to the left of the line which requires 50m (a), 100m (b), 300m (c), 500m (d), and 2000m (e) avoidance. A video of RBPC running in the Test08 scenario can be found here: `https://youtu.be/ru4Pj5HE_4s` 109

B-27 Test09. ASV must navigate along a survey line with a faster obstacle from astern which requires 50m (a), 100m (b), 300m (c), 500m (d), and 2000m (e) avoidance. A video of RBPC running in the Test09 scenario can be found here: `https://youtu.be/SMx9qOK-iwE` 110

B-28 Test10. ASV must navigate along a survey line with a faster obstacle from *directly* astern which requires 50m (a), 100m (b), 300m (c), 500m (d), and 2000m (e) avoidance. A video of RBPC running in the Test10 scenario can be found here: `https://youtu.be/xufyfoqKZSs` 111

B-29 Test11. ASV must navigate along a survey line with moving obstacles from ahead and astern which require 50m (a), 100m (b), 300m (c), 500m (d), and 2000m (e) avoidance. A video of RBPC running in the Test11 scenario can be found here: `https://youtu.be/wOEnj0JaS1Y` 112

ABSTRACT

REAL TIME MOTION PLANNING FOR PATH COVERAGE
WITH APPLICATIONS IN OCEAN SURVEYING

by

Alexander F. Brown

University of New Hampshire, September, 2020

Ocean surveying is the acquisition of acoustic data representing various features of the seafloor and the water above it, including water depth, seafloor composition, the presence of fish, and more. Historically, this was a task performed solely by manned vessels, but with advances in robotics and sensor technology, autonomous surface vehicles (ASVs) with sonar equipment are beginning to supplement and replace their more costly crewed counterparts. The popularity of these vessels calls for advances in software to control them.

In this thesis we define the problem of path coverage to represent and generalize that of ocean surveying, and propose a real-time motion planning algorithm to solve it. We prove theorems of completeness and local asymptotic optimality regarding the proposed algorithm, and evaluate it in a simulated environment. We also discover a lack of robustness in the Dubins vehicle model when applied to real-time motion planning. We implement a model-predictive controller and other components for an autonomous surveying system, and evaluate it in simulation. The system documented in this thesis takes a step towards fully autonomous ocean surveying, and proposes further extensions that get even closer to that goal.

CHAPTER 1

Introduction

1.1 Motivation

This work is motivated by the problem of autonomous ocean surveying - piloting an autonomous surface vehicle (ASV) while operating a sonar in such a way as to collect meaningful data from the ocean floor. While this is a task often conducted by human operators or simple controllers, both options mandate a high degree of human supervision. Sonar operation imposes constraints on vehicle motion, and the ocean is dynamic, which makes controlling a vehicle difficult. The ocean also contains many hazards, including rocks, buoys, land, and other ships. These factors combined make manually operating an ASV challenging.

As researchers of robot autonomy, we strive to reduce the level of human attention required for robots to operate safely. Increased autonomy allows fewer humans to operate an ASV, or more ASVs to be operated by a single human. Reducing the required manpower can bring down the cost of a survey and increase the efficiency. Efficient surveying will be critical for meeting the ambitious goals established by Mayer et al. (2018): mapping the entire seafloor by the year 2030. Humans operators also occasionally make mistakes, so it is another goal to create a system that will eventually be more reliable than its human counterparts.

In addition to being a useful problem to solve, autonomous ocean surveying presents interesting computational and motion planning challenges. Guaranteeing real time responsiveness in a dynamic, large environment is difficult, and the non-trivial objective of surveying makes it an even more unique problem.

1.2 Ocean Surveying

In this section we briefly describe ocean surveying. Ocean surveying is a complex task of data acquisition which pulls together work from underwater acoustics, signal processing, positioning systems, and more.

There are many reasons to survey a region of ocean. Hydrography, of course, is a common goal, but also common are searching for lost items such as sunken ships or crashed airplanes, examining marine habitats, and planning placement of oil pipelines and other offshore engineering. After identifying a region to survey, plotting a survey which effectively covers the area can take time and tools. One must consider how the depths in the region affect the width of a sonar swath from a vehicle, and plan to drive in such a way as to keep the ship as stable as possible during data collection.

Sonar devices for measuring seafloor depth often generate a swath of sound at a fixed angle, which results in a depth-dependent area of seafloor ensonification. To accurately deduce depth from these sounding data, one must remove the vehicle's motion from the measurements. Surveys are very often performed in a boustrophedon, or lawnmower, pattern - either at fixed spacing, for relatively flat seafloor, or with depth-dependent spacing in sloped seafloors to guarantee both efficiency and complete coverage. Surveys are often planned completely in advance, but can also be generated adaptively, line by line, based on data collected so far. Survey specifications can include requirements for data density, coverage of the seafloor, and limits on uncertainty, which depend on the intended uses of the survey data. Surveys are planned specifically to meet these objectives, and may need to be adjusted in the field to satisfy them.

Crewed vessels are often piloted by human mariners, who can react to obstacles or changes in the environment gracefully. Without obstacles of any kind, a proportional-integral-derivative (PID) controller minimizing across-track error can work very well in varying wind and currents to pilot an ASV along a survey line. While on a survey line, sonar is operated to collect data about the seafloor, objects between the surface and the bottom, or even profiling below the seafloor. It is important to keep the vessel steady during this process, as the quality of data quickly deteriorates otherwise. The path coverage problem is an attempt to automate the line following portion of this

while accounting for obstacles, as a simple controller is not equipped to.

Surveys can span hundreds of square kilometers and take months to complete. For a single crewed survey vessel, operation can cost in excess of $5,000 per day, with larger survey ships exceeding $40,000 per day. Anything that causes the vehicle to deviate from the planned survey path can drastically increase the cost or decrease the quality of a survey. While simply avoiding obstacles is critical, it is also important to minimize deviations from the planned survey. Thus, it is desirable to balance marginal safety with surveying quality and efficiency. Human mariners implicitly navigate this tradeoff, and it is important that automated agents do so as well.

1.3 Path Coverage

In the interest of modeling ocean surveying, let us define a *path coverage problem.* See Figure 1-1 for an illustration of an instance of the problem. Since we will be planning for a robot, we require a *configuration space.* In our case, for ocean surveying, this is spatial coordinates in a local map reference frame or *workspace*, (x and y), combined with *heading*, *speed* and *time.* We also require a static map which tells us the *obstacle space* (and thus, the *free space*), i.e. where the robot cannot go.

For reasons discussed later, we require a *steering function.* A steering function computes a feasible trajectory between two *poses* (generally, points in the configuration space), and is typically robot-specific. We choose to model an ASV as a Dubins car (Dubins 1957), which allows us to compute Dubins curves as a steering function. See Section 3.3.3 for a description of Dubins curves.

As part of the problem we need to allow for the input of one or more *path segments*, or survey lines in the running example of ocean surveying, in the workspace. These are, in general, lines along which we ought to pilot the robot, and, specifically in ocean surveying, they are lines over which we should operate the sonar. We represent these as a set of line segments with an associated width, or tolerance, within which it is acceptable to survey. The robot is expected to cover all of these path segments in their entirety.

As we are operating in a real, dynamic environment, we need to guarantee real-time decision making, so as part of the input to the problem we include a computation time bound. This guarantee

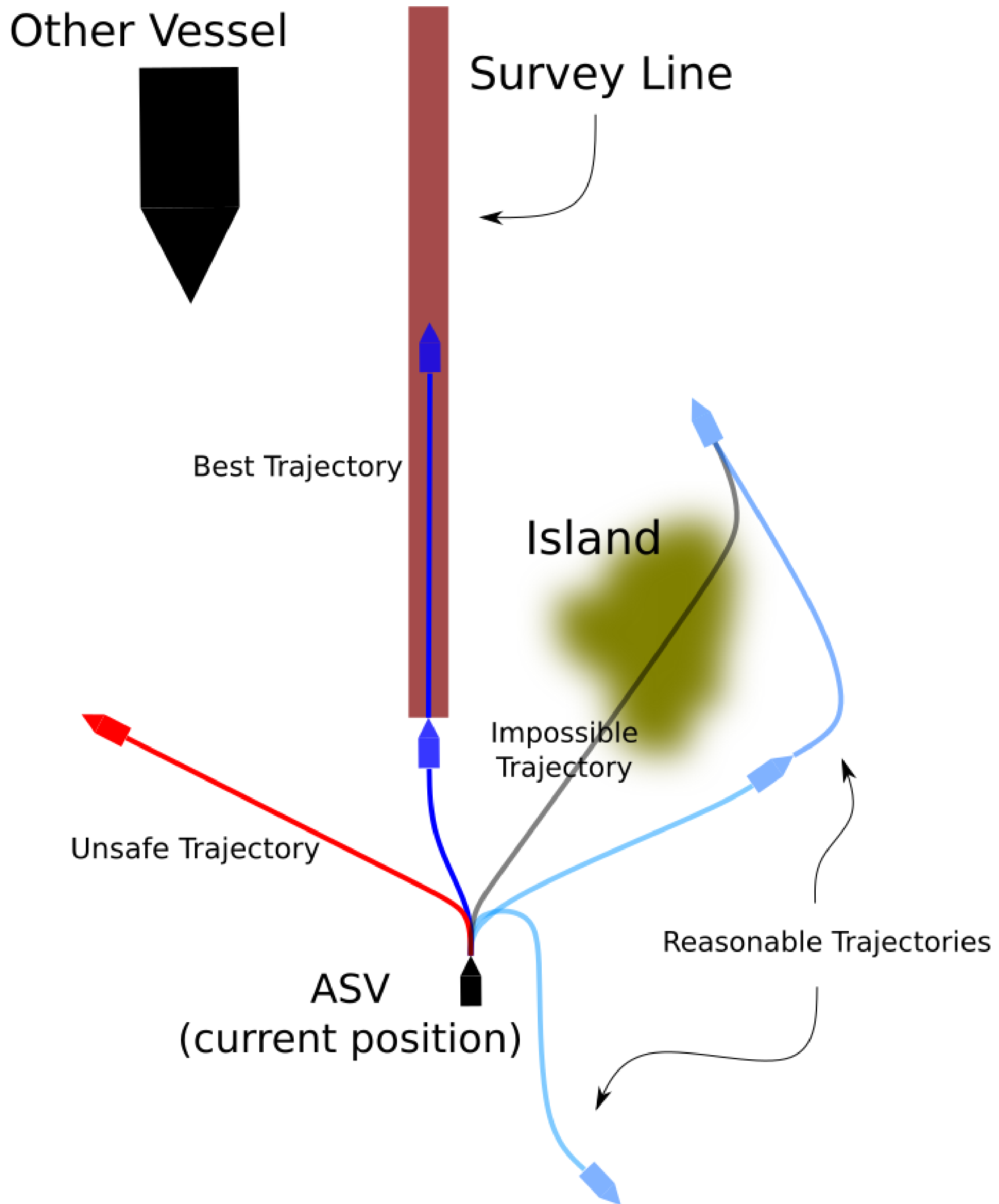


Figure 1-1: Diagram illustrating ocean surveying, the motivation for the path coverage problem. The ASV's pose is shown with several trajectories leaving it, marked either impossible, unsafe, reasonable, or best. The best trajectory is marked as such because it achieves some coverage of the survey line while avoiding obstacles. The unsafe trajectory passes in front of another vessel, which is undesirable. The impossible trajectory attempts to pilot through an island.

is important so operators can be confident the robot will respond to new or updated information. For example, in the ocean surveying context, new radar data might suggest the presence of another ship in the vicinity. The robot may need to avert its course quickly before a collision is inevitable. We also need to guarantee that, if something goes wrong in the planning phase, our last plan is long enough that the robot can continue to operate safely until an exterior actor, human or otherwise, can take control or fix the system. This parameter is represented as a minimum trajectory duration.

These parameters will typically be the same between problem instances, but we have two more dynamic aspects which will tend to change every instance. We need a starting pose for the robot from which to begin planning. This could be approximated by the current pose of the robot, but note that planning takes (a specific amount of) time, so it is preferable to have an estimate of what the pose will be at the *end* of the planning cycle, one computation-time-bound from when it begins. Finally, we wish to plan while considering dynamic obstacles as well as static ones, so we need some representation of their current locations and future behavior. This representation must allow for time-based queries of likelihood of collision, because plans will need to check for collisions at a fine resolution. We implement two simple representations which meet these criteria.

Bringing all these features together, we define the path coverage problem as follows. Given

- configuration space,
- obstacle space and free space,
- steering function,
- path segments,
- computation time bound,
- minimum trajectory duration,
- starting pose,
- dynamic obstacle information,

find a kinematically feasible trajectory

- longer than the minimum trajectory duration, and
- computed within the computation time bound.

Solutions are preferred which avoid potential collisions with dynamic obstacles and which traverse portions of the path segments.

The path coverage problem is, at its core, a motion planning problem: the input contains a robot pose and the output is a trajectory for the robot. While existing motion planning algorithms can easily handle static and dynamic obstacles and take advantage of a steering function, its complicated objective and real-time nature make the path coverage problem worth studying. No existing real-time motion planning algorithm is suitable for path coverage without adaptation.

To solve this problem we require an algorithm that terminates in real time, and can optimize for efficiency, coverage, and safety together. We propose such an algorithm in Chapter 3, and describe it in detail.

1.4 Overview

There are two core contributions around which this thesis is written: a motion planner for the new path coverage problem, and other architecture and engineering which is needed to apply it in practice. Prior work, the bulk of which is more related to the motion planning contribution, is presented in Chapter 2. The motion planning algorithm is described in detail in Chapter 3. We provide pseudocode for the algorithm and some implementation details for success and performance improvements. We also prove three theoretical properties of the algorithm in that same chapter.

The two subsequent chapters transition to the architecture and engineering contributions. The model-predictive controller presented in Chapter 4 has undergone a lot of development to specifically support a real-time motion planning algorithm. It follows motion plans as closely as possible, estimating external disturbances, and provides pose estimates to the planning algorithm. Chapter 5 discusses the rest of the relevant architecture in place for autonomous ocean surveying.

In Chapter 6 we provide experimental results comparing our motion planner with a planner employing an artificial potential field. The potential field planner uses the same architecture and

controller as the motion planner we present. Finally, Chapter 7 gives some discussion and limitations of the system, prescribes future work, and concludes the thesis.

1.5 Contributions

We formalize a new path coverage problem to model autonomous ocean surveying. We design and evaluate a real-time motion planning algorithm for the path coverage problem. While discussing the algorithm, we describe an important property of the Dubins vehicle, which results in several unforeseen modifications to the algorithm and system. We design an approximate model-predictive controller to accompany our motion planning algorithm, issuing lower-level controls. We analyze the results of several experiments with the system, comparing the motion planner to a planner using an artificial potential field, and varying parameters in the system.

CHAPTER 2

Related Work

Prior work in planning for ASVs is insufficient to tackle the path coverage problem in its entirety. For example, Song, Liu, and Bucknall (2017) develop a fast marching method for point to point path planning, considering environmental factors. While avoiding obstacles effectively, their planner does not optimize a well-defined objective, which makes it poorly suited to path coverage planning because for two trajectories with the same starting and ending poses, one may be unequivocally preferable, in that it covers a larger portion of a survey line.

Shah et al. (2014) implement a planner for ASVs that focuses on following the International Regulations for the Prevention of Collisions at Sea (COLREGs) to minimize collisions with other vessels, and also develops contingency plans in case those other vessels breach COLREGs. They follow up this work with an adaptive version for faster, suboptimal planning Shah et al. (2016). Obeying COLREGs and reasoning about COLREGs is important for any vessel at sea, and are possible extensions to the research presented here, but are beyond the scope of this thesis. We direct the reader to (Shah 2016) for a very thorough discussion of the state of the art in COLREGs-compliant motion planning. Shah et al. (2014) also develop a trajectory planner, but it relies on motion primitives which prevent it from being asymptotically complete.

Reed and Schmidt (2016) implement reactive obstacle avoidance for ASVs by turning the vehicle away from obstacles as it approaches them, but their strategy makes no attempt at avoiding collisions with foresight, allowing for the vehicle to easily be diverted well away from its intended path, trapped in a loop, or pushed into a state of inevitable collision.

Artificial Potential Fields (Barraquand, Langlois, and Latombe; Choset et al. 1992; 2005) have been used extensively in robot motion plannning, but suffer from the same struggle with local minima. We implement a motion planner using a potential field and compare it to our algorithm.

Sampling-based motion planning is a well-studied problem. Perhaps among the most well-known and foundational algorithms is the construction of Rapidly-exploring Random Trees (RRTs) (LaValle and Kuffner 1999), which forms the basis of more sophisticated sampling-based motion planning algorithms. RRTs are designed to cope with higher dimensional spaces than are feasible for dynamic programming, and they are built by sampling and connecting to the nearest existing branch of a motion tree, a process which is very quick in practice, and is informally also called RRT. What RRT lacks in optimality is made up for in an extension, RRT*, which adds rewiring to the tree construction in such a way as to be sure that any path along the motion tree is the shortest possible path through the sampled points (Karaman et al. 2011). RRT* is shown to converge to an optimal motion plan with infinite samples. A later work, AO-RRT (Kleinbort et al. 2019), brings only small changes to the original RRT algorithm which make it asymptotically optimal as well. There are many other flavors of RRT-based motion planning (Kuffner and LaValle; Hsu et al.; Frazzoli, Dahleh, and Feron; LaValle; Otte and Frazzoli 2000; 2002; 2002; 2006; 2016).

The early tree-based motion planning approaches do little to guide their growth, but in the world of graph search, the use of heuristics is essential for efficient search times. A strict generalization of Dijkstra's famous algorithm (Dijkstra 1959), A* (Hart, Nilsson, and Raphael 1968) is the de facto standard for simple, efficient heuristic search. Not only does it guarantee optimal solutions with an admissible, consistent heuristic, it also provides optimal computational efficiency: any complete algorithm for optimal graph search using the same heuristic will touch at least as many vertices as A*.

At the junction of sampling-based motion planning and heuristic search lies the work of Gammell et. al. in their algorithm Batch Informed Trees (BIT*) (Gammell, Srinivasa, and Barfoot; Gammell, Barfoot, and Srinivasa 2015; 2020). This work uses batches of samples to build and refine a motion plan in an any-time fashion, guided by heuristics. Sampling is refined to only include new samples which could contribute to an improved solution, and expensive cost calculations (i.e. collision checking) are deferred as long as possible. BIT* also guarantees asymptotic optimality by extending the proof of the same for RRT*. Later works extend BIT* by accelerating local search in RABIT* (Choudhury et al. 2016), adding a simplified reverse search to improve heuristics with AIT* (Strub

and Gammell 2020), and suboptimally searching batches of samples for speed in ABIT* (Strub and Gammell 2020).

Sophisticated algorithms for heuristic graph search can deal with changing edge costs in efficient ways (Stentz; Koenig, Likhachev, and Furcy; Koenig and Likhachev 1995; 2004; 2005). Incremental search appears to supersede these algorithms in later work (Koenig and Sun 2009). In particular, Anytime Dynamic A* (AD*) (Likhachev and Ferguson 2009) was employed successfully on an autonomous passenger car to with the DARPA Urban Challenge. A drawback of these algorithms is that they are most effectively used planning backward from a goal state, so as to re-use most of the search when replanning. In the path coverage problem planning backwards is infeasible, as there are many potential goal poses. Determining the best one appears to be as computationally hard as solving the problem, because a goal requires that all path segments be covered. In order to prove all path segments were covered when reaching a pose, one must know the exact trajectory taken.

In his 1957 work "On Curves of Minimal Length with a Constraint on Average Curvature, and Prescribed Initial and Terminal Positions and Tangents", L. E. Dubins shows that paths of minimal length between two points with initial angles and a fixed minimum radius for turning (not unlike configurations for an ASV) can be composed of three segments of three specific types and can be analytically computed (Dubins 1957). This is convenient for planning for ASVs (Karaman et al. 2011), because it is fast to find these geodesic paths, and an ASV is typically constrained by a minimum turning radius. We use Dubins curves for our steering function.

A variety of work has considered coverage planning in various forms. The reader is directed to (Galceran and Carreras 2013) for a comprehensive overview. Galceran and Carreras (2012) propose an algorithm for autonomous marine surveying which applies a cellular decomposition to the region of interest and generates a covering plan which sweeps the resulting cells individually, avoiding obstacles in the space. This algorithm, and others in the coverage path planning domain, treat obstacles as static, and plan once. We aim for a real-time approach where some obstacles can move unpredictably, which necessitates constant replanning. Bircher et al. (2018) use a receding horizon in their approach to 3D exploration and surface inspection with unmanned aerial vehicles,

for the sake of computational complexity. We take inspiration from this work in the receding horizon of our algorithm.

Although existing work is equipped to handle various aspects of the path coverage problem separately, to our knowledge no algorithm combines all these requisite features. Hence, we present a novel motion planning algorithm, described in detail in the following chapter.

CHAPTER 3

An Algorithm for Path Coverage

In this chapter we present an algorithm for path coverage. First, we discuss features of the path coverage problem defined in Chapter 1 that impact algorithm design. We then provide a description of the algorithm, along with pseudo code. We introduce several modifications which help the algorithm succeed in realistic applications, some of which are necessitated by a property of Dubins curves that we also describe. Finally, we provide theoretical analysis of the algorithm, proving theorems of real-time termination, local asymptotic optimality, and probabilistic completeness.

3.1 Features of Path Coverage that Impact Algorithm Design

To motivate a novel algorithm, let us first review features of the path coverage problem that impact algorithm design. We must offer a guarantee that planning computation will be fast enough to react to updated information while safe actions can still be taken in order to operate in a dynamic and potentially dangerous world. Algorithms that can offer this guarantee are categorized as *real-time*, and they must restrict their computation effort to be constant in the size of their input. Algorithms that are real-time allow us to make useful guarantees in the dynamic real world.

Dynamic motion planning allows costs to change, as obstacles are discovered or updated, and allows the robot to move through the search space. If changes are not drastic, most of the search can be re-used in the re-planning triggered by updates. Dynamic motion planning algorithms (Stentz; Likhachev and Ferguson 1995; 2009) take advantage of this by planning backwards from the goal pose to the start pose, which lets them re-use most of the motion tree when applicable. The path coverage problem has no single goal pose, as the objective is a state of the world, not just the robot. If the optimal order in which to cover the path segments were known a priori, one could plan backwards from the end of the last segment, but, since dynamic obstacles and the robot's

current pose might impact this order, the order is challenging to decide, and could change with each dynamic obstacle update or new pose. With this in mind, we judged it to be impractical to attempt to re-use any portion of previous motion trees.

We design an algorithm similar to BIT* (Gammell, Barfoot, and Srinivasa 2020) because it has several desirable properties. As a sampling-based algorithm it can more easily handle higher-dimensional spaces than grid-based or lattice-based planners. Poses are sampled randomly, so the authors prove probabilistic completeness and asymptotic optimality without having to prove reachability for motion primitives, but, with a steering function, kinematic feasibility is still assured. The batch sampling and planning naturally provides an anytime-like procedure, where the algorithm improves an initial solution as time allows. BIT* incorporates heuristic guidance, and has been used recently in high-performance applications. We borrow all of these ideas, discussed in Section 3.2 with our algorithm description.

3.1.1 Adaptations for Path Coverage

The path coverage is an unusual motion planning problem because collision penalty calculation depends on time: dynamic obstacles must be projected in time along each trajectory in order to determine collision probability. We discuss the changes this quirk necessitates in RRT*(Karaman et al. 2011) and BIT*(Gammell, Barfoot, and Srinivasa 2020).

As mentioned in Chapter 2, RRT* extends RRT to provide asymptotic optimality. It grows a motion tree toward randomly sampled poses, as in RRT, but also checks the motion tree near each new vertex to determine if the new vertex provides a shorter route to any vertices already in the tree. Vertices whose path is improved by going through the new vertex are "re-wired" to come from the new, better branch. This change has a ripple effect of costs along the subtree of each re-wired vertex: they now have a better earlier trajectory and thus a lower cost. These improvements provably converge to an optimal solution.

BIT* iteratively samples batches of poses and performs heuristic graph search in a random geometric graph specified by the sampled poses and a connectivity radius. The graph search forms a motion tree, similar to RRT and RRT*, but the tree is grown in batches rather than incrementally.

Collision checking is often expensive in motion planning, so it is deferred as long as possible, guided by an additional heuristic. Re-wiring occurs in the same fashion as in RRT*. Granularity increases as the samples become more dense, asymptotically converging to the optimal solution.

In the path coverage problem, vertices in the motion tree have an associated time. If a re-wiring of vertex v occurs, a new trajectory has been discovered to arrive at v's pose. Both the coverage and collision penalty must be updated in v to reflect the new parent trajectory. Costs for the v's subtree must also be updated, to reflect v's new cost. If the time to get to v has changed, collision penalty must be recalculated for v's *entire subtree*, because the times for trajectories in that subtree have all changed. Repeating collision checking for an entire subtree at each re-wiring is prohibitively expensive, especially when we consider that the depth of the motion tree tends to grow as finer-grained trajectories are discovered. Thus, it is impractical to allow re-wiring in the path coverage context.

3.2 Algorithm Description

Before describing the algorithm itself it is important to specify what it is trying to optimize. The objective is composed of three distinct pieces: safety, coverage, and efficiency. The safety term captures our desire to avoid obstacles, and increases as collisions become more probable. The coverage term represents the remaining path segments, and decreases as we come closer to covering them all. The efficiency term denotes how long we have taken thus far in our efforts to complete the task. We try to minimize the weighted sum of these three terms with Algorithm 1.

$$objective = efficiency + w * safety + coverage \tag{3.1}$$

Efficiency and coverage are in units of time, explained below, so they should have equal weights. It is thus convenient for those weights to be 1, letting w be the only weight in Equation 3.1.

Our motion planning aglorithm, Real-time BIT* adapted for Path Coverage (RBPC), is presented in simplified form in Algs. 1, 2 and 3. For the sake of the reader's understanding we present the algorithm in this pure form first, and describe the nuances subsequently in more detail. Struc-

turally RBPC is very similar to BIT* (Gammell, Barfoot, and Srinivasa 2020), in that it cycles between batch sampling and planning (Alg 1). It takes an initial vertex *start*, containing an initial pose and the path segments not yet covered, and repeatedly searches an increasingly fine random geometric graph whose edges are formed by connecting sampled poses with the steering function. RBPC searches more eagerly than BIT* using A* (Alg. 2), bounded by the incumbent solution cost. We note that the laziness in BIT* comes from delayed edge evaluation, based on an additional heuristic, which RBPC lacks. The function *Sample* takes an integer and returns a set of sampled poses with that cardinality. $TimeRemaining$ determines if the computation time bound has elapsed, or if more planning time remains. $bestVertex$ stores the incumbent goal vertex, and *bestCost* stores the incumbent solution cost. $f(v)$ is used to denote the lower bound on a solution to the overall path coverage problem which passes through vertex v, and is composed of cost to come to v, denoted by $g(v)$, and a lower bound heuristic estimate of cost to go from v, denoted $h(v)$. The queue Q in Alg. 2 is ordered on increasing $f(v)$, breaking ties on low h: $Pop(Q)$ returns the minimum f vertex in Q, and if there are vertices with equal f, one such vertex with minimal h in that set is returned. Relating these terms to the objective in Equation 3.1,

$$f(v) = g(v) + h(v) = (\textit{efficiency} + w * \textit{safety}) + (\textit{coverage})\,. \tag{3.2}$$

Collision penalty and time measure safety and efficiency along the trajectory to vertex v, and the heuristic estimate of coverage time remaining $h(v)$ is the coverage term.

Algorithm 1 RBPC

Input: $start$

1: $samples \leftarrow Sample(initialSamples)$
2: $bestVertex \leftarrow nil$
3: $bestCost \leftarrow \infty$
4: **while** $TimeRemaining()$ **do**
5: $\quad v \leftarrow \mathbf{A^*}(samples, start, bestCost)$
6: $\quad$ **if** $f(v) < bestCost$ **then**
7: $\quad\quad bestVertex \leftarrow v$
8: $\quad\quad bestCost \leftarrow f(v)$
9: $\quad samples \leftarrow samples \cup Sample(|samples|)$
10: **return** $TracePlan(bestVertex)$

Algorithm 2 A*

Input: $samples, start, bestCost$

1: $Q \leftarrow start$
2: **while** $Q \neq \emptyset$ **do**
3:   **if** $\neg TimeRemaining()$ **then**
4:     **return** nil
5:   $v \leftarrow Pop(Q)$
6:   **if** $Goal(v)$ **then**
7:     **return** v
8:   **Expand**$(v, bestCost, samples)$
9: **return** nil

To a reader familiar with A* (Hart, Nilsson, and Raphael 1968), these concepts will be familiar but confusing in this context. By definition, A* should return goal vertices, and goal vertex v should have heuristic cost to go $h(v) = 0$, so the assignment $bestCost \leftarrow f(v)$ could normally be simplified to $bestCost \leftarrow g(v)$. This brings us to the first nuance of the algorithm. In order to guarantee real-time termination, we bound the search space with a time horizon. The function *Goal* in Alg. 2 determines whether a vertex's state is at the horizon, with no consideration of coverage. $h(v)$ estimates the cost to cover the remaining path segments, regardless of the horizon, so evaluating $h(v)$ for a vertex v on the time horizon is perfectly valid and will often return a non-zero cost to go. By ordering Q on f, when f is monotone, **A*** must select the minimum f goal in the graph.

Algorithm 3 Expand

Input: $v, bestCost, samples$

1: $radii \leftarrow \{default, coverage\}$
2: $speeds \leftarrow \{default, slow\}$
3: **for** $configuration \in speeds \times radii$ **do**
4:   **for** $p \in GetNearbyPoses(v, configuration, samples)$ **do**
5:     $v' \leftarrow Connect(v, p)$
6:     $CollisionAndCoverageCheck((v, v'))$ {Compute cost to come $(g(v'))$}
7:     $ComputeCostToGo(v')$ {Compute heuristic estimate of cost to go $(h(v'))$}
8:     **if** $f(v') \leq bestCost$ **then**
9:       $Push(Q, v')$

Algorithm 3, **Expand**, creates vertices from a set of poses near v by the steering function (Alg. 3, lines 4-5). This set can be determined by selecting poses within a radius which shrinks iteration,

given by:

$$radius(q) := 2\eta \left(1 + \frac{1}{n}\right)^{\frac{1}{n}} \left(\frac{\lambda\left(X_{\hat{f}}\right)}{\zeta_n}\right)^{\frac{1}{n}} \left(\frac{log(q)}{q}\right)^{\frac{1}{n}}, \tag{3.3}$$

where q is the number of vertices currently in the graph, n is the dimensionality of the configuration space, $X_{\hat{f}}$ is the set of states which could contribute to a better solution than found so far, the function $\lambda(\cdot)$ represents the Lebesque measure of a set, ζ_n represents the Lebesque measure of an n-dimensional unit ball, and $\eta \geq 1$ is a tuning parameter (Gammell, Barfoot, and Srinivasa 2020). This radius is sufficient to guarantee connectivity in the random geometric graph which RBPC explores. For simplicity, the implementation used in the experiments approximates an alternative approach to connectivity by selecting the K nearest poses, where K is a well-chosen parameter. Poses to which the trajectory would take more time than remains between v and the time horizon are shifted so they lie exactly on the horizon.

Each edge (v, v') must be checked for collisions and coverage to compute $f(v')$ and determine priority in Q. Both coverage and collision checking are performed together at a fixed increment along the trajectory represented by (v, v') (Alg. 3 lines 6-7). Edges are generated by four different configurations: two turning radii and two speeds (Alg. 3 lines 1-4). At the smaller turning radius, *default*, coverage of path segments is not allowed, as the heading rate is too high for collecting good data. The larger radius, *coverage*, causes a sufficiently slow heading rate that surveying can be performed. Coverage is allowed on straight Dubins segments from either configuration. This constraint on heading rate is enforced solely in the application of autonomous ocean surveying: turning quickly tends to leave gaps in sonar data. In other applications coverage might have different restrictions. Two different speeds are considered: a default speed, *default*, and a slow speed, *slow*. The default speed should be used for most coverage and traversal. The slow speed gives the planner an option to slow down to avoid dynamic obstacles, as this is often the preferred avoidance action.

We implement two representations of dynamic obstacles, which, while treated the same during collision checking, have rather different ideologies. One option considers dynamic obstacles as a rectangle that moves at a fixed velocity. If the robot is found to be inside an obstacle during a

collision checking step, it incurs a very large penalty. While this is a simplistic model, and captures no uncertainty in obstacle position, it makes it easy to explain the planner's choices, and one can be very certain the planner will avoid dynamic obstacles if at all possible. The other option represents dynamic obstacles as a multivariate Gaussian probability distribution, again moving at a fixed velocity. Under this model the robot incurs a collision penalty scaled by the probability density function of each obstacle evaluated at the robot's position. This representation is advantageous because it lets the penalty scale continuously with proximity, so the planner can balance proximity with coverage, or between multiple obstacles. Unfortunately, it is difficult to determine how to scale the probability density function, subject to different covariances and obstacle sizes, such that desirable behavior is achieved in all scenarios.

We implement three heuristics to guide search and evaluate trajectories at the time horizon in RBPC (mutually exclusively). We also describe an adaptation to two of them to deal with large numbers of path segments. To discuss these heuristics, let us first define two measures of distance in the autonomous ocean surveying configuration space. Let $d_{Euclidean}(x_1, x_2)$ be the Euclidean distance in the workspace between poses x_1 and x_2. Let $d_{Steering}(x_1, x_2)$ be the length of the minimal trajectory between poses x_1 and x_2 by the steering function. For the purpose of this discussion, let a path segment start pose be a pose on one end of a path segment whose heading aligns the robot with the segment in the correct direction to begin covering it, and let a path segment end pose be a similar pose whose heading points the robot to leave the segment, as though coverage has just been completed. Note that these poses are each valid on either end of a segment, but are distinguished by differing headings.

A simple, computationally easy heuristic h_{sum} takes $d_{Euclidean}$ to the nearest segment start pose, plus the sum length of all path segments. This serves to pull search greedily towards the nearest path, which tends to be a good idea. It is clearly admissible, as $d_{Euclidean}$ is strictly no greater than $d_{Steering}$, which is how the vehicle actually drives, and all path segments must be covered. It is also consistent: neither total path length nor distance to the nearest path segment can decrease by more than the distance traveled by the robot. We also implement two heuristics that find traveling salesman tours through the path segments, to better estimate the minimal distance required to

finish the instance. They each find the optimal tour considering cities to be path segments, where the robot must enter at a start pose and leave by the corresponding end pose, having covered the segment. h_{point} ignores kinodynamics and uses $d_{Euclidean}$ to find edge costs for the tour, whereas $h_{Steering}$ uses the more informed $d_{Steering}$.

We note that h_{point} is both admissible and consistent, similarly to h_{sum}, but $h_{Steering}$ is not. For two poses close in $d_{Euclidean}$, the steering function may have to perform a looping maneuver to line up other state variables, such as heading. This is only inadmissible in cases where coverage is possible without going through the exact start pose for a path. See Sections 3.3.3-3.3.6 for more details about when this happens and what we can do about it during search. The approaches suggested in those sections are not general enough to make $h_{Steering}$ admissible.

For large numbers of path segments it is intractable to compute optimal tours, so we propose an adaptation to the latter two heuristics. We restrict the branching factor in the problem to a constant: passage between cities is only allowed for the J closest neighbors for any given city. While this prohibits admissibility for even h_{point}, we feel that often enough a close-to-optimal solution can be found, making this approach useful in practice. It is a point of future work to investigate other approaches to approximate solutions.

C++ source code for RBPC is available at `https://github.com/afb2001/path_planner`.

3.3 Biasing Sampling and Search

3.3.1 Biasing sampling

Sampling is restricted to an area bounded by Euclidean distance equal to the time horizon multiplied by the maximum speed of the robot, because poses outside this area will be unreachable in search. Poses are sampled from a uniform distribution in the workspace and in heading, but in order to help the algorithm find good plans quickly, we bias sampling to occasionally sample on the path segments. We bias search towards path segment endpoints, but not onto path segments partway through. Sampling on the path segments reliably provides this option to the algorithm.

3.3.2 Biasing search

Biasing search has proven vital to reasonable performance in even unrealistically good conditions. Not only does the *GetNearbyPoses* (Alg. 3 line 4) function find a set of near poses, it also finds the nearest path start pose and connects a vertex there. If the vertex is already on a path, *GetNearbyPoses* finds the opposite end pose instead. By always expanding to either of these poses we always provide a good option for getting onto the nearest path segment or continuing to cover a segment. This practice led to the discovery of the property of Dubins curves described in the following section.

3.3.3 A note about Dubins paths

In the context of frequent re-planning, Dubins curves are not robust. There are two sources of error which make this fact almost debilitating: real-world disturbances and numeric instability. Real-world disturbances are present in most robotics applications, whether they be actuation noise or true external forces, and, while numeric instability may be implementation-dependent, the author is not able to prevent it. Regardless of the source, shifting poses by a small amount in the Cartesian plane or in heading can increase the minimal length Dubins curve between them drastically.

Dubins curves are composed of three segments, or words, of three different types. These word types are: R, a maximal right turn, L, a maximal left turn, and S, a straight line. Straight words can only fall between two curve words, and no word can be repeated immediately (e.g. LRL is allowed but LLR is not). This allows for six possible path types: LSL, LSR, RSL, RSR, RLR, LRL.

For a fixed ending pose and starting heading, the Cartesian plane can be partitioned into regions where a specific path type is optimal. Example partitions are shown in Figure 3-1 and Figure 3-2. We observe that the example RSR curve (Figure 3-2a) originates from the RSR partition, but we could draw a reciprocal curve from the upper, LSL partition. Any curve we draw in that figure will have the same sequence of words as others in the same partition. The same applies to Figure 3-2b, but the partitions have been drawn for a different starting heading.

While stepping along a Dubins path at small intervals, the poses close to the endpoint often

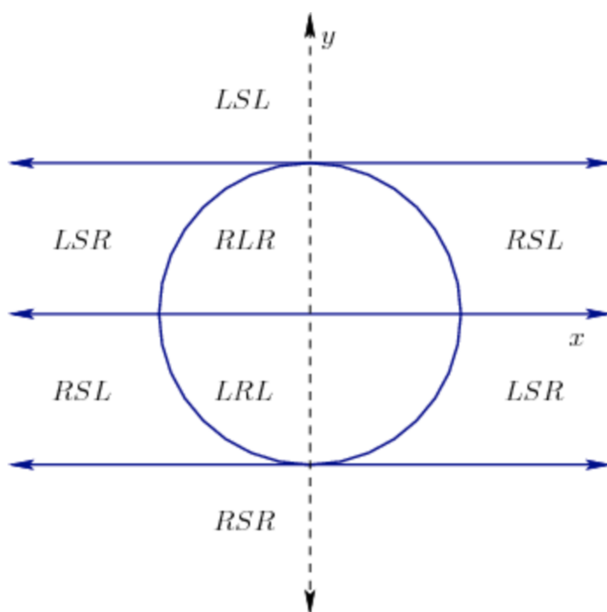
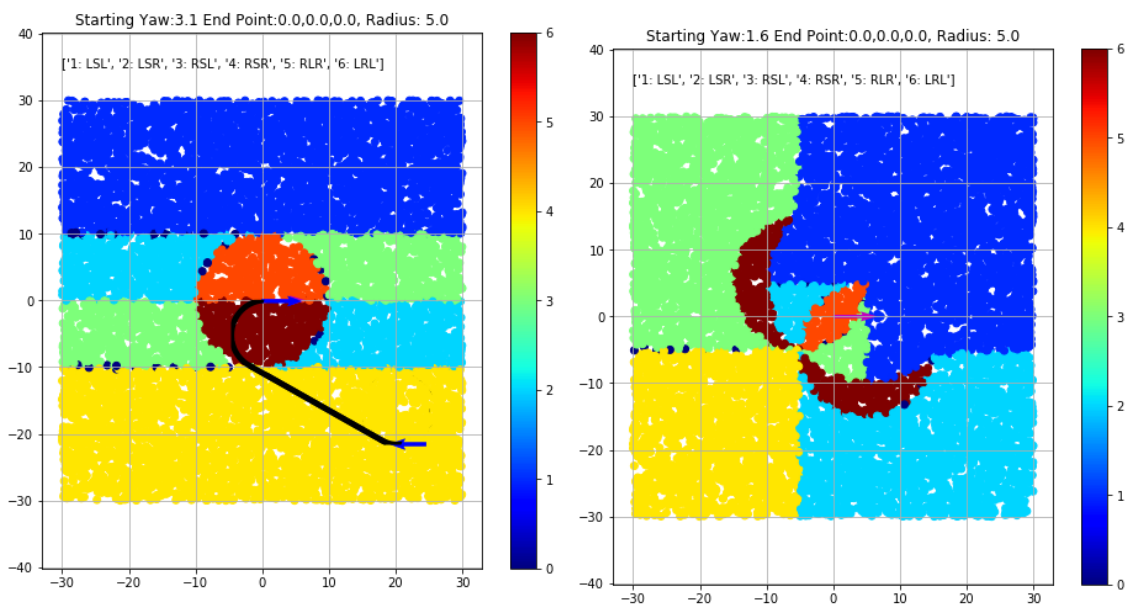


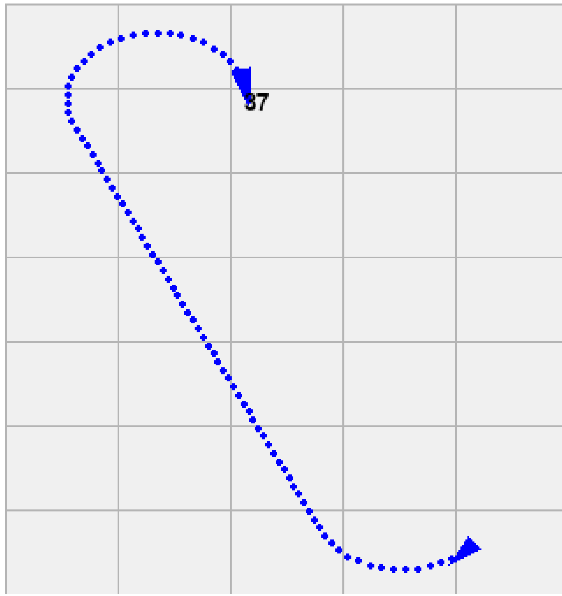
Figure 3-1: A slice at $\theta = \pi$ of the partition into word-invariant cells for the Dubins car. The circle is centered on the origin. Figure from LaValle (2006).



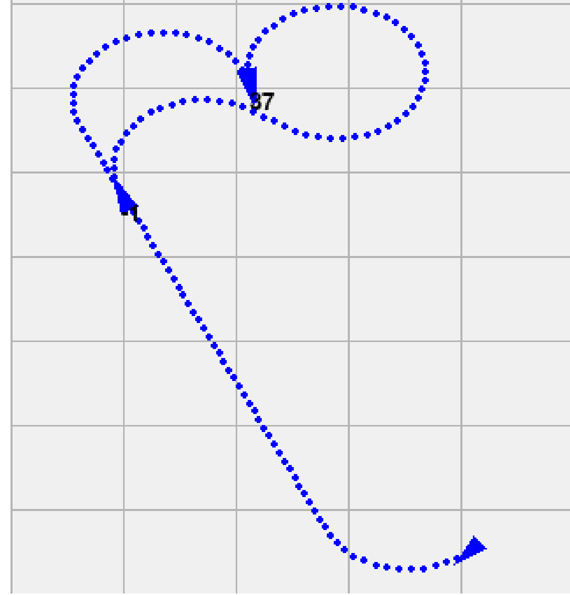
(a) Slice of path type partitions with starting heading of π radians North of East. A sample curve is shown to illustrate the partitioning.

(b) Slice of path type partitions with starting heading of 1.6 radians North of East.

Figure 3-2: Two slices of path type partitions with different starting headings.



(a) Example Dubins path



(b) Dubins path with a re-computed suffix drawn as well. We would expect the re-calculated path to match the end of the original, but it deviates and forms a loop.

Figure 3-3: Dubins path shown with an incorrect re-computed suffix. This demonstrates numerical instability in Dubins curves.

walk the boundaries of these partitions. When computing a single curve this is unimportant, but when re-computing the remaining curve, or suffix, a slight disturbance can push the suffix starting configuration into another partition, changing the optimal path type and potentially increasing the length. An example of this is shown in two parts in Figure 3-3. Figure 3-3a shows a Dubins curve, and Figure 3-3b overlays the suffix computed partway through, which deviates from the original curve.

Re-calculating curves in this way can occur when repeatedly running an algorithm like RBPC, partially executing plans it produces during each planning cycle. In particular, it will happen when the best (lowest f) local trajectory is a direct extension of the best trajectory discovered in the previous cycle. When there is only one nearby path segment and no nearby dynamic obstacles, the best trajectory tends to be one which goes immediately to the path segment and begins covering it. Thus, the lack of robustness will affect planning as the robot approaches path segments. This observation is confirmed by pilot experiments.

The fragility of Dubins curves, caused by numerical instability or real-world disturbances, can prevent RBPC from even beginning to cover paths. A video of this effect can be found here: `https://youtu.be/qJMch2bXbeg`. RBPC attempts to approach the survey line, but cannot reliably extend the previous plan when the robot gets close to the path segment, because of Dubins instability. We describe three strategies for overcoming these effects in the following sections.

3.3.4 Exact plan tracking

We allow the planner to be separated from reality with the simplifying assumption that the robot has not deviated from the previous plan. We only allow this abstraction if the controller determined that the previous plan was feasible upon receipt. See Chapter 4 for a description of how the controller determines this. Assuming we have not deviated from the previous plan removes any effects of external disturbances or actuation noise, but it does not completely mitigate the lack of robustness documented in the previous section: the controller may not always be able to achieve plans, and numerical instability can still push Dubins curve recalculation across partition boundaries and make it challenging to cover a path segment.

3.3.5 Plan re-use

If the robot has not deviated from the previous plan, we can find good plans that extend it without recalculating the first portion of the previous plan by including it directly in search. We accomplish this by collision checking the previous plan and adding the resulting vertex to the open list Q along with the starting vertex *start* in Alg. 2 line 1. Because the plan is already precisely known, we do not need to calculate a fragile Dubins suffix, and can thus circumvent numeric instability. In the case where the previous plan begins to cover a line, adding that additional vertex makes the same option of starting coverage available. It has the side effect of seeding search with an almost-complete good plan, which helps the first search, unbounded by an incumbent cost, find a solution quickly. The previous plan may not be feasible to extend if we do not assume the robot has not deviated from it. Re-using plans in this way has shown to be very helpful in pilot experiments.

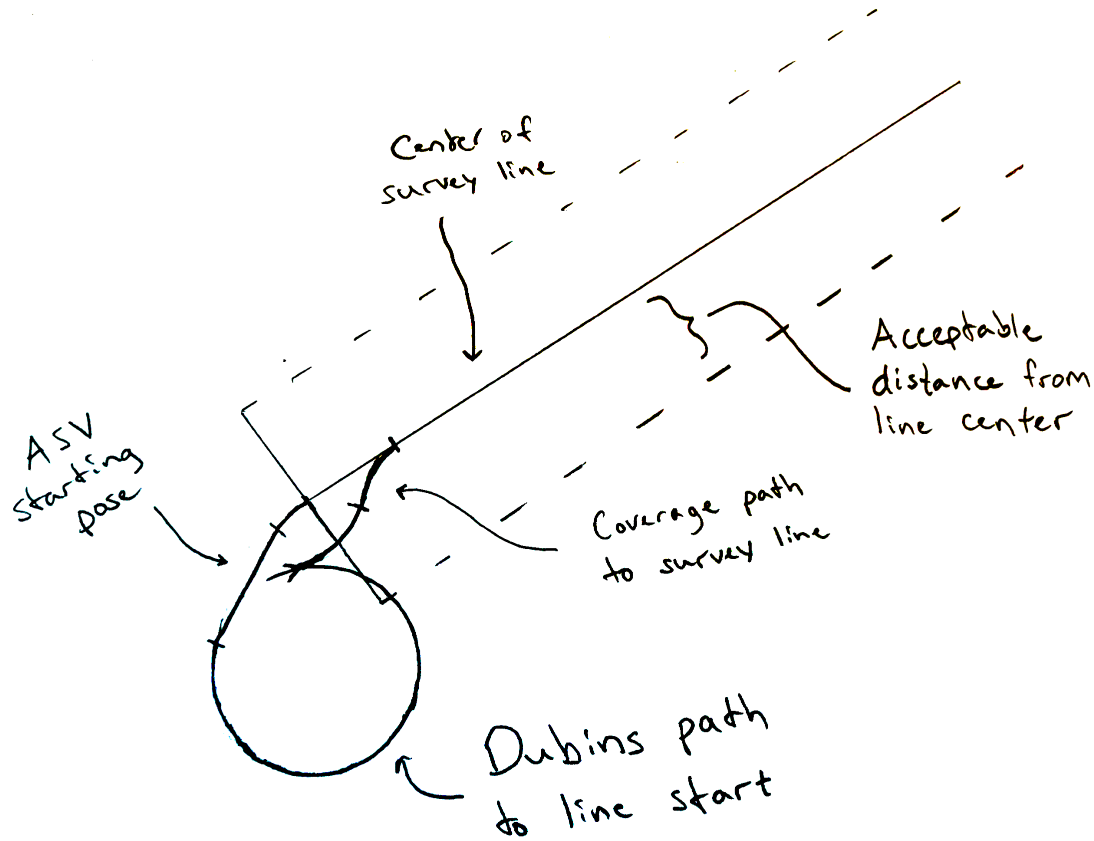


Figure 3-4: A coverage path vs a Dubins path to the start pose for beginning line coverage. When a large-radius turn can be completed within the line width it is far better to do so than to take a Dubins path to the center endpoint of the line.

3.3.6 Coverage Paths

We also design robust coverage trajectories which bring the robot onto a nearby path in a more robust way than going to a precise start pose for the segment. The robust coverage trajectories connect a pose to a nearby pose on a path segment. Unlike the earlier search bias, these trajectories do not necessarily target a path starting pose. Instead, using a turning radius for which coverage is allowed, they find a point along the path that can be connected while staying as close to the center of the path as possible. These paths are most beneficial when only a small turn is required and can be completed withing the path width, ensuring complete coverage. We note that this situation naturally occurs as the robot approaches a path segment and experiences a small disturbance. An example of this situation is shown in Figures 3-4 and 3-5.

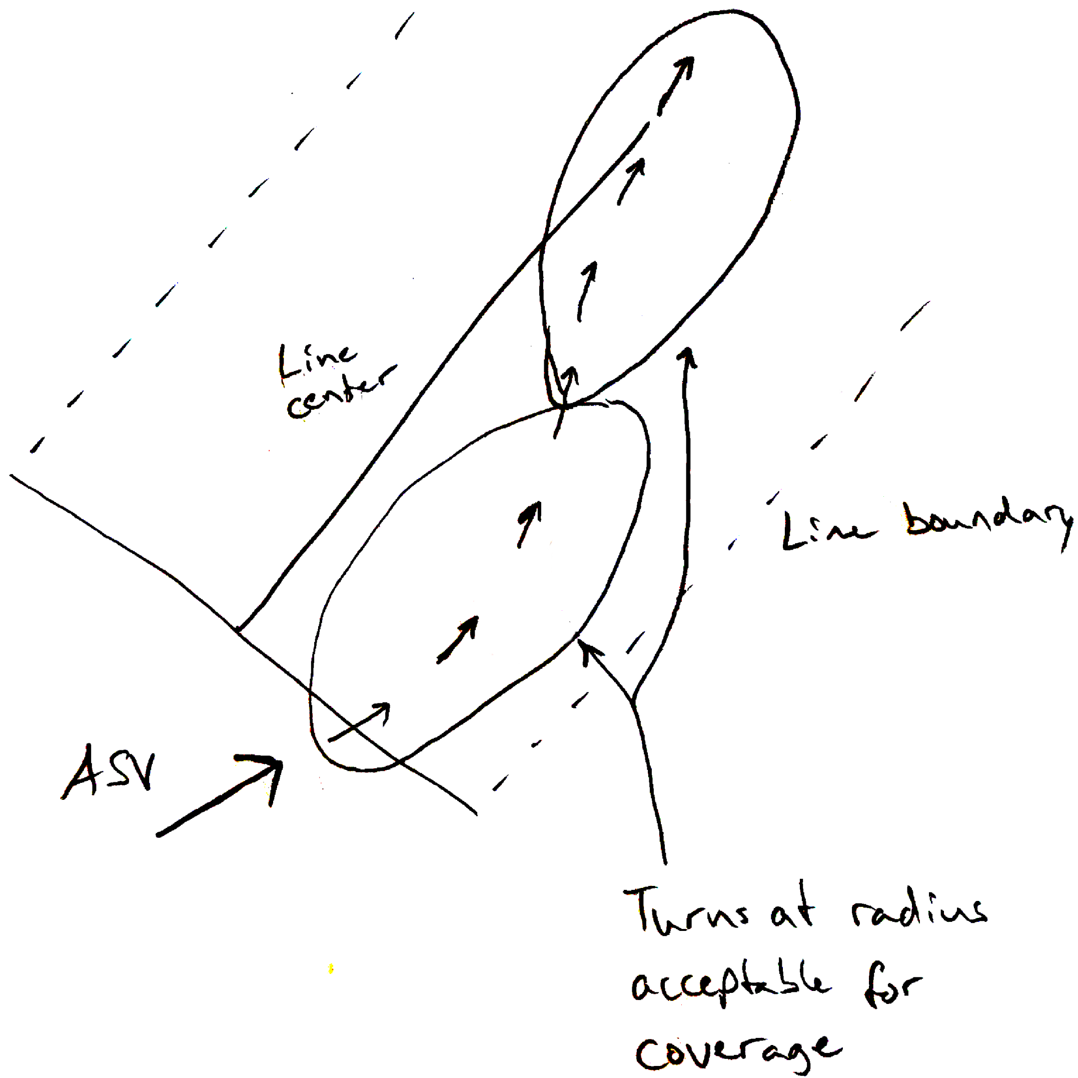


Figure 3-5: An example robust coverage trajectory in more detail. It consists of two curves: a turn inwards, towards the center of the line, and a turn outwards, for the final alignment. When both curves can be completed within the line width this is desirable, as no portion of the segment will be left uncovered.

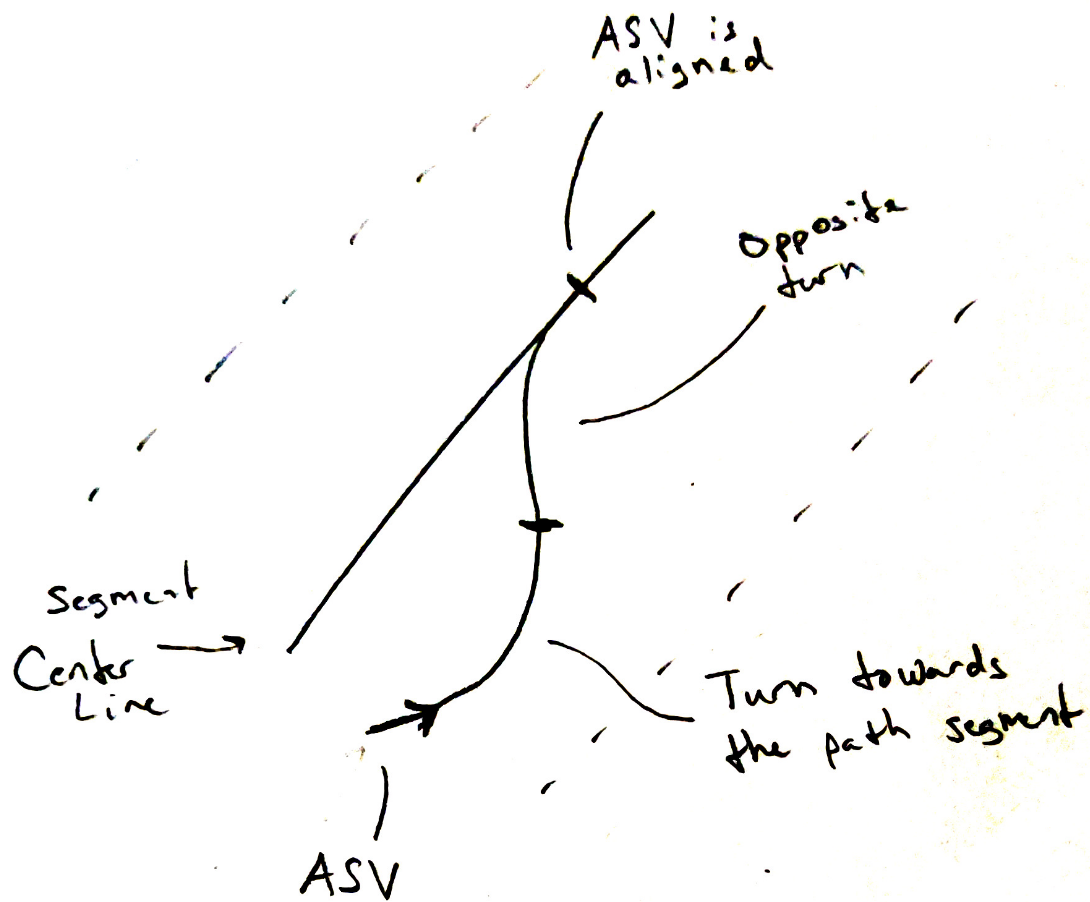


Figure 3-6: ASV is headed away from the center of the path segment. It executes a turn towards the center of the segment followed by a turn in the opposite direction, ending aligned with the center line.

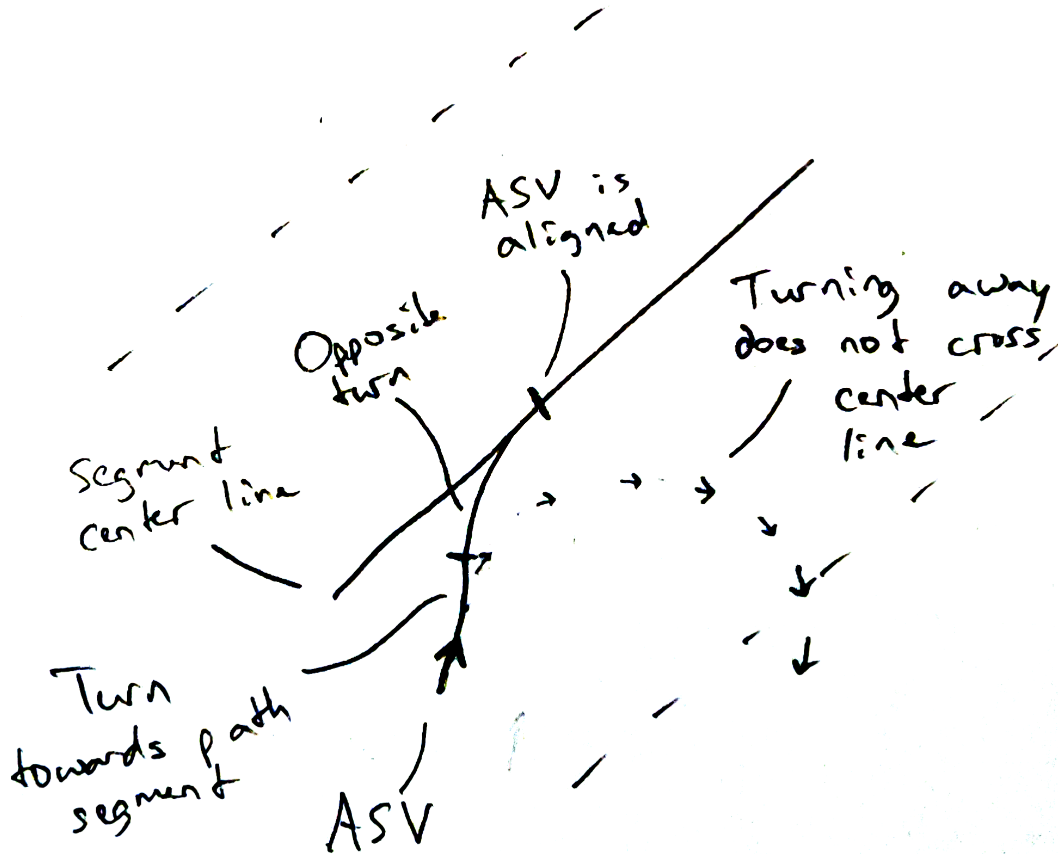


Figure 3-7: ASV is headed towards the center of the path segment. By executing a turn away from the center line, it would not cross the center, so it turns first to the left, towards the center line. Then, it turns to the right, ending aligned with the center line.

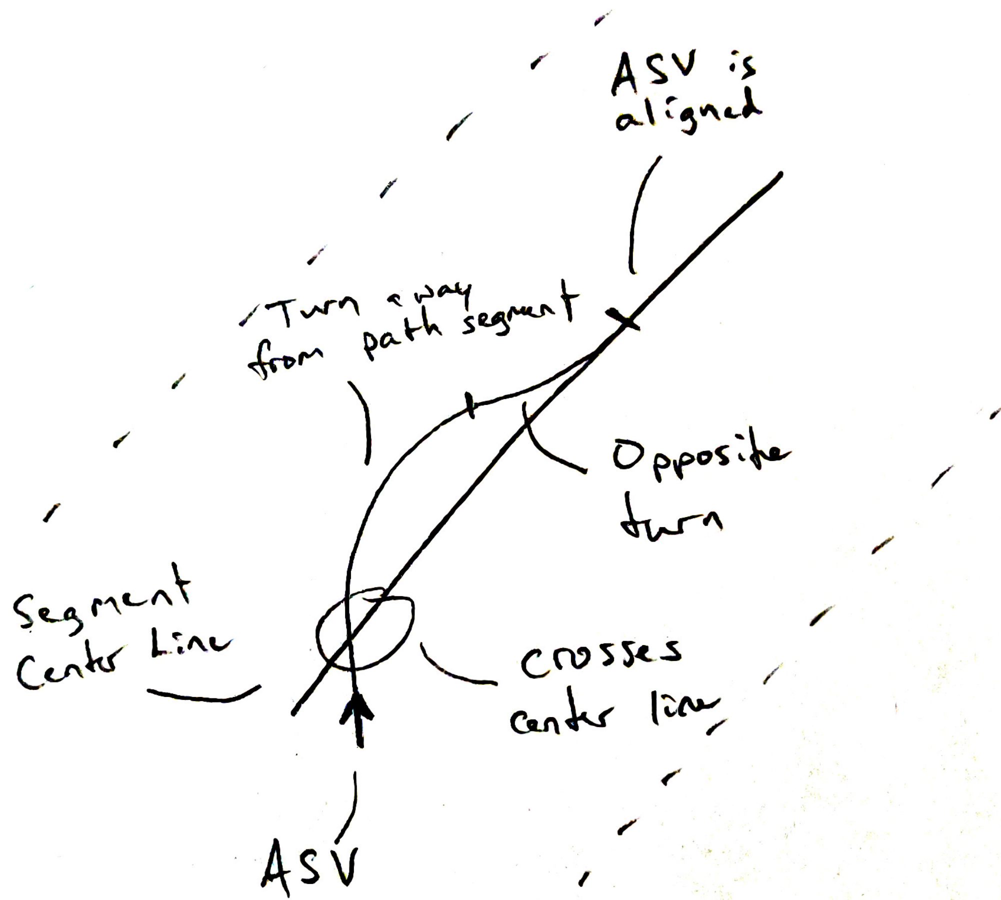


Figure 3-8: ASV is headed towards the center of the path segment. The ASV turns to the right, away from the center line, but still crosses it. It follows up with a turn in the opposite direction, ending aligned with the center line.

The computation of robust coverage trajectories depends on the robot's distance from a path segment and its heading with respect to the segment. We ignore segments from which the robot is more than two turning radii away, as those cases are better served by driving to the segment start pose with a standard Dubins curve. We separate the remaining cases by whether the robot is heading towards the path segment or away. In each case, the coverage trajectory is a specialized Dubins curve composed of two turning segments in opposite directions (with a zero-length segment, to be a valid Dubins curve). Figures 3-6, 3-7, and 3-8 show the different cases, with exaggerated heading differences for clarity. If the robot is facing away from the segment, it should execute a turn towards the segment, followed by a turn in the opposite direction. An example of this is shown in Figure 3-6. The second turn aligns the robot to the path segment. If the robot is facing away from the segment, but a turn away would not pass through the center line of the segment, the robot should turn first towards the segment and then in the opposite direction, as in the first case. An example of this is shown in Figure 3-7. Otherwise, if turning away from the path segment would bring the robot through the center line of the segment, the robot should execute that turn away, passing through the segment, followed by a turn in the other direction. An example of this situation is shown in Figure 3-8.

We find that the inclusion of these specific trajectories is very helpful to robustly cover segments without looping around. Vertices on the end of these trajectories are injected into the open list Q at the same time as the starting vertex *start* is added in Alg. 2 line 1. A video showing the success of these adjustments can be found here: `https://youtu.be/h6yc8jLrVUA`.

3.4 Theoretical Properties

We will now discuss theoretical properties of RBPC. We prove that it is real-time, asymptotically optimal within its local search space, and probabilistically complete.

3.4.1 Real-time

In order to prove that RBPC can meet a reasonable computation time bound we must prove that it performs a constant number of computations as the size of the input increases. The main way the

input size can change is in the length and number of path segments, and their distance from the robot. Before we prove the runtime is constant in this dimension, let us discuss other parameters on which the runtime depends.

As in many motion planning algorithms, a lot of computational effort is put into collision checking trajectory segments. In RBPC this is done at a fixed distance increment. Collision checking trajectories costs time linear in the length of the trajectory but also inverse linear in the size of this increment: the smaller the increment the more time it will take to check a trajectory. We assume the increment will remain fixed, and is large enough to allow RBPC to terminate within a reasonable time bound. Computational effort in collision checking could also depend on the number and complexity of dynamic obstacles. We assume that these are bounded low enough so that they do not prevent the algorithm from terminating within a time bound as well.

Assumption 1. *The time complexity for collision checking a trajectory is linear in trajectory length and bounded by a constant in all other dimensions, including collision checking density and dynamic obstacle number and complexity.*

If we were to plan a trajectory to cover every path segment, under Assumption 1 our collision checking complexity would be linear in the sum length of segments and in their distance from the robot. Hence, we bound our trajectory computation out to a fixed time horizon as discussed earlier in this chapter. This allows us the following lemma:

Lemma 1. *Collision checking in RBPC has constant time complexity in the size of the input.*

Proof. At a maximum speed, a fixed time horizon also defines a distance horizon. By Assumption 1 we know that collision checking takes linear time in the distance to be checked, and constant elsewhere. Since the distance is bounded by a constant, collision checking computation is also bounded by a constant. □

Theorem 1. *RBPC terminates within its given real-time bound.*

Proof. By Lemma 1, collision checking has constant time complexity in the size of the input. All other factors in time complexity are not controlled by the input, so there exists a number of initial samples that allows RBPC to terminate in real-time. □

It is important that RBPC terminate in real time to satisfy the time bound given in the path coverage problem defined in Chapter 1. We impose this requirement because we need to be able to guarantee responsiveness for the robot, as it operates in a dynamic hazardous environment.

3.4.2 Asymptotic local optimality

In the following we prove RBPC is asymptotically optimal in the local search space. To prove this, we show that RBPC considers every edge RRT* (Karaman et al. 2011) would consider, so, since RRT* is asymptotically optimal, RBPC must be as well.

Theorem 2. *The probability that RBPC converges asymptotically towards the lowest f trajectory on the time horizon when given infinite samples is one.*

Proof. Theorem 2 is proven by showing that RBPC considers at least the same edges as RRT* for a sequence of samples $samples = (x_1, x_2, \dots, x_q)$ and a connection limit $r_{RBPC} \geq r_{RRT^*}$. We note that (3.3) uses the same connection radius for a batch that RRT* would use for the first sample in the batch and the connection radius of both are monotonically decreasing, so the condition $r_{RBPC} \geq r_{RRT^*}$ holds. The structure of this proof is similar to that found in the proof of Theorem 3 in Gammell, Barfoot, and Srinivasa (2020), Section 4.

RRT* incrementally builds a tree from a sequence of samples. For each pose in the sequence, $x_k \in samples$, it considers the neighborhood of earlier samples that are within the connection limit,

$$X_{near,k} := \{x_j \in samples \mid j < k, ||x_k - x_j||_2 \leq r_{RRT^*}\}.$$

It selects the connection from this neighborhood that minimizes the cost-to-come to the sample and then evaluates the ability of connections from this sample to reduce the cost-to-come of the other states in the neighborhood.

Given the same sequence of samples, RBPC processes them in a series of batches, $(Y_1, Y_2, \dots, Y_\ell)$, where each batch adds new samples at an exponential rate:

$$Y_i = \{x_j \in samples \mid j < initialSamples * 2^i\}.$$

For each sample in the batch, $y \in Y_k$, it considers the neighborhood of samples from the same batch within the connection limit,

$$X_{near,k} := \{x \in Y_k \mid ||y - x||_2 \leq r_{RBPC}\}.$$

It adds the edge in this neighborhood to the tree that minimizes the cost-to-come to the sample and considers all the outgoing edges to connect its neighbors. This set of edges contains all those considered by RRT* for an equivalent connection limit, $r_{RBPC} \geq r_{RRT^*}$.

As $r_{RBPC} \geq r_{RRT^*}$, this shows that RBPC considers at least the same edges as RRT*. Because each batch starts a fresh motion tree, re-wiring of the tree need not be considered, as those edges that would be created by re-wiring will occur naturally during edge generation. With a consistent heuristic, trajectories in the motion tree are optimal with respect to the samples thus far, because the motion tree is explored by A* (Alg. 2). RBPC is therefore almost-surely asymptotically optimal. □

We strive for asymptotic optimality because it separates the real-time algorithm from the hardware on which it is run: increasing computation speed should improve the quality of solutions. Since safety is part of the objective function, given enough time, RBPC will find safe trajectories if they exist. The objective also considers efficiency of coverage, so safe trajectories that allow coverage will be preferred over safe trajectories which do not. RBPC will not lead the robot further away than necessary from coverage in pursuit of safety, within the limits of its lookahead.

3.4.3 Probabilistic completeness

Probabilistic completeness for RBPC is different than other similar algorithms because of its receding horizon. Solutions returned within the time bound are not required to fully solve an instance, so it makes more sense to talk of completeness over a period of interwoven planning and execution.

We observe that the path coverage problem can be broken down into a sequence of traditional motion planning problems which each provide positive coverage, where the goal is some fixed pose, rather than completion of all coverage. Completeness for a single problem implies completeness for

the sequence, because each problem in the sequence could then be solved, so for the rest of this subsection we discuss only a single motion planning problem.

Our proof by contradiction follows those of Korf (1990) for RTA* and Fickert et al. (2020) for Nancy: we first prove that incompleteness implies a set of regions within which RBPC circulates forever. Then we prove that, with dynamic programming-like heuristic updates, such a set cannot exist. In order to prove probabilistic completeness we make the following assumptions.

Assumption 2. *The controller is able to follow planned trajectories accurately.*

This is reasonable to assume given that we have a steering function which should simplistically model the dynamics of the robot, and that the robot should not be operating in extreme conditions.

Assumption 3. *There are no dynamic obstacles.*

An antagonistic dynamic obstacle could always intercept the robot at exactly the right times to prevent safe coverage, so we must make this assumption to prove the theoretical property of probabilistic completeness. We note that there may exist some slightly weaker assumption about behavior of dynamic obstacles which would also suffice here, but its existence and subsequent derivation is left as a matter of future work.

Assumption 4. *A goal is reachable from every pose.*

This is perhaps the most limiting assumption presented thus far, because it prevents the workspace from having dead ends. We direct the reader to Fridovich-Keil, Fisac, and Tomlin (2019) for a framework to extend existing motion planning algorithms to avoid dead ends.

Assumption 5. *A motion planning algorithm builds a motion tree of states, extending from an initial pose. These states are referred to as vertices in the context of graph search to build the motion tree.*

Sampling-based motion planning algorithms like RBPC build a motion tree in this way. Probabilistic completeness for other approaches to motion planning are not discussed in this thesis.

Let X denote the space of all poses. Note that we have defined the configuration space to include time, but that poses are time-invariant, so X is not the configuration space.

Definition 1. For two poses $x_1, x_2 \in X$, we say that x_2 is ***reachable***$_A$ from x_1 if and only if a real-time motion planning algorithm A will expand x_2 from a search started at x_1 with positive probability.

Let us extend the definition of reachability to sets of poses. For any two sets of poses $r_1, r_2 \subseteq X$, we say that r_2 is *reachable*$_A$ from r_1 if and only if $\exists_{x_1 \in r_1}$: $\exists_{x_2 \in r_2}$: x_2 is *reachable*$_A$ from x_1. We also define the related notion of *fully reachable*$_A$ such that region r_2 is *fully reachable*$_A$ from r_1 if and only if $\forall_{x_1 \in r_1}$: $\forall_{x_2 \in r_2}$: x_2 is *reachable*$_A$ from x_1.

Assumption 6. *The space X is bounded.*

We need to assume the space of all poses is bounded in order to define the following partitioning of X.

Definition 2. We define a partitioning R of the space of all poses X into a finite set of disjoint regions such that, given a real-time motion planning algorithm A:

- Every region is fully reachable$_A$ from itself. $\forall_{r \in R}$: r is *reachable*$_A$ from r. [1]
- Every region has at least one other *reachable*$_A$ region. $\forall_{r_1 \in R}$: $\exists_{r_2 \in R}$: r_2 is *reachable*$_A$ from r_1 and $r_1 \neq r_2$.
- Heuristic values $\hat{h}(r)$ of region r are applied to all poses in r in search.

Definition 3. We define notation pertaining to regions and search:

- we denote the region containing pose x to be $r(x)$,
- we denote the pose of vertex v by $x(v)$,
- we denote the composition $r(x(v))$ as simply $r(v)$.

Definition 4. We define a directed region graph $G_{R,A} = (R, E_R)$ over the regions R such that edges $(r_1, r_2) \in E_R$ between two regions $r_1, r_2 \in R$ have finite, positive cost $c(r_1, r_2)$ and exist if and only if r_1 is not r_2 and for motion planning algorithm A, r_2 is *reachable*$_A$ from r_1.

[1] This definition may be more restrictive than necessary, and a less restrictive definition is worth investigating.

The edge costs in $G_{R,A}$ are not theoretically important, as long as they are positive and finite, but for an implementation of a heuristic employing the learning learning described later in this section, it might be helpful for them to represent in some way the cost to transfer between regions. We note that under Assumption 4, for a single goal $G_{R,A}$ will connected, or, if there are multiple goals, each component of $G_{R,A}$ must contain at least one goal.

Assumption 7. *For any region r, the heuristic estimate, $\hat{h}(r)$, is initially finite.*

Similar to the edge costs, the initial heuristic values are not theoretically important, but it might be helpful to an implementation of RBPC for them to represent an estimate of the cost to go from a region.

Definition 5. A real-time motion planning algorithm is *goal-aware* if, upon discovering a goal in its lookahead, it commits to a path toward it.

Lemma 2. *RBPC is goal-aware with a heuristic that is zero at goals and non-zero elsewhere.*

Proof. RBPC selects the lowest f vertex on the time horizon. If a goal is reachable in local search, it will have the lowest f value, as its heuristic will be zero and we assume no dynamic obstacles (Assumption 3). □

Lemma 3. *If a motion planning algorithm A is incomplete, in the limit of computation speed, under Assumption 4 it must run forever.*

Proof. Under Assumption 4, a goal is reachable from all poses. This means there are no dead ends where the robot could get stuck and never leave to find a goal. In the limit of infinite computation speed, A will always find trajectories to any $reachable_A$ regions, and thus be able to go to them. Hence, the only way for a motion planning algorithm to terminate on a problem instance is to reach a goal. Thus any motion planning algorithm that does not find a goal must run forever. □

Lemma 4. *A goal-aware motion planning algorithm A will achieve a goal in any region fully $reachable_A$ from the starting pose with positive probability.*

Proof. A goal pose in a *fully* $reachable_A$ region will be expanded by A with positive probability (Definition 1). A goal-aware motion planning algorithm will commit to a path to a goal when one

is found in lookahead (Definition 5). Thus, with positive probability, algorithm A will achieve a goal if one is *reachable*$_A$ from the starting pose. □

Definition 6. A set of regions $R_\circ$ is called a *circulating set* if there exists a time $t_\circ$ after which the agent will only visit regions $r \in R_\circ$ and will visit each infinitely often.

Lemma 5. *A circulating set $R_\circ$ for a goal-aware real-time motion planning algorithm A cannot contain a goal-containing region.*

Proof. Suppose, for contradiction, that a circulating set $R_\circ$ for a goal-aware real-time motion planning algorithm A contains a goal-containing region. Regions in $R_\circ$ are visited infinitely often (Definition 6). Regions are $fully\ reachable_A$ from themselves (Definition 2). A will achieve a $reachable_A$ goal with positive probability (Lemma 4). Thus A will achieve the goal in $R_\circ$, terminating, contrary to the definition of $R_\circ$ (Definition 6). □

Lemma 6. *If a goal-aware real-time motion planning algorithm A is incomplete it must have a non-empty circulating set $R_\circ$.*

Proof. If a motion planning algorithm is incomplete, by Lemma 3 it will run forever. Because the set of regions R is finite (Definition 2), there must exist a subset of regions $R_\circ \subset R$ that the robot will visit an infinite number of times after some initial time t. By Lemma 5 no regions in $R_\circ$ can contain a goal. If there exist regions not visited infinitely often, let time t_s be the last time such a region was visited. Then $t_\circ := max(t, t_s)$ satisfies the claim. □

Lemma 7. *Under Assumptions 4 and 6, if a goal-aware real-time motion planning algorithm A has a circulating set $R_\circ$, then*

- *there exists a finite set of non-goal-containing regions $R_\infty \supseteq R_\circ$ that are reachable$_A$ infinitely often,*
- *the set of reachable$_A$ frontier regions $R_F := R_\infty \setminus R_\circ$ is non-empty,*
- *there is a time t_1 after which no region in R_F is visited.*

Proof. Lookaheads from all regions $r \in R_\circ$ are performed infinitely often, so there must be a set of regions R^r_∞ which are $reachable_A$ from r infinitely often (Definition 2). Their union over $r \in R_\circ$ is R_∞. R_∞ is finite, because the set of regions is finite (Definition 2). The frontier regions R_F are $reachable_A$ infinitely often from regions in R_∞. As they are not in $R_\circ$ they are visited only a finite number of times, so the claimed t_1 exists. $R_\circ$ does not contain a goal (Lemma 5), but a goal is reachable from all regions (Assumption 4), so R_F must be non-empty and contain at least one region containing either a goal or poses on a path to a goal. R_F is finite because it is a subset of R_∞, and of R, which are both finite (Definition 2). □

Definition 7. A real-time motion planning algorithm does *dynamic programming-like learning* if it updates heuristic values of regions visited by the robot such that afterwards the heuristic value of region r whose successors in $G_{R,A}$ are denoted by R' satisfy

$$\hat{h}(r) = \min_{r' \in R'} (c(r, r') + \hat{h}(r')). \tag{3.4}$$

A standard result is that if updates of this form are performed infinitely often, on a finite graph with positive edge cost, from finite initial values, the $\hat{h}$ values will eventually converge, satisfying Equation 3.4 (Bertsekas and Tsitsiklis (1996), Proposition 2.3). We have a finite graph of regions $G_{R,A}$ with positive edge costs (Definition 4) and finite initial values (Assumption 7), so this result is applicable. The reader is directed to Koenig and Sun (2009) for an in-depth example of dynamic programming-like learning in real-time heuristic search.

Lemma 8. *Under Assumptions 3-6, if a real-time motion planning algorithm that performs dynamic programming-like learning has a circulating set $R_\circ$, then there exists a time t_2 after which heuristic values have converged to satisfy Equation 3.4 for every region in $R_\circ$.*

Proof. $\hat{h}$ values in regions $r_f \in R_F$ are never updated, since they are never visited. All update operations performed on R_∞ are well-defined, i.e., consider visited regions in R_∞. By construction, the update procedure is called infinitely often in every region in $R_\circ$. Due to the convergence of the dynamic programming-like learning procedure with positive edge cost (Definition 4) and bounded initial $\hat{h}$ values (Assumption 7), the $\hat{h}$ values in these regions will eventually converge to a solution

of Equation 3.4 as claimed. ☐

In the following we will use f_v and g_v to denote the f and g value, respectively, of a vertex in a lookahead search space with respect to the current root vertex v of the search.

Definition 8. A real-time motion planning algorithm is called *persistent* if, whenever it generates a vertex v_f in region r_f in lookahead from starting vertex v with $\hat{f}_v(v_f) < \hat{f}_v(v)$ or $\hat{f}_v(v_f) = \hat{f}_v(v)$ and $\hat{h}(r_f) < \hat{h}(r(v))$, it will eventually visit r_f, or a region r' containing a vertex v' where $\hat{f}_v(v') < \hat{f}_v(v_f)$ or $\hat{f}_v(v') = \hat{f}_v(v_f)$ and $\hat{h}(r') < \hat{h}(r_f)$.

Lemma 9. *Under Assumptions 3-6, in the limit of infinite computation speed, an asymptotically persistent and goal-aware real-time motion planning algorithm that does dynamic programming-like learning cannot have a non-empty circulating set.*

Proof. Assume, for the sake of contradiction, that the search algorithm does have a circulating set $R_\circ$. Then there must be sets R_∞ and R_F as per Lemma 7. By Lemma 8 we know that $\hat{h}$ will eventually converge on all regions in $R_\circ$ at some point in time t_2. Let $r_f \in R_F$ be a region with minimal $\hat{h}$ among regions in R_F. Since r_f is reachable infinitely often, a vertex v_f at the time horizon with a pose in r_f will be generated in some lookahead at time $t_3 > t_2$. Since heuristic values are converged in $R_\circ$, for all $r \in R_\circ$, $\hat{h}(r_f) < \hat{h}(r)$. In the limit of computation speed, v_f will be expanded. There are no dynamic obstacles (Assumption 3), so all vertices at the time horizon must be uniform in g, and v_f has minimal $\hat{h}$ among those vertices. Since search is persistent, it will eventually visit r_f, or a region r' containing a vertex v' with $\hat{f}_v(v') < \hat{f}_v(v_f)$ or $\hat{f}_v(v') = \hat{f}_v(v_f)$ and $\hat{h}(r') < \hat{h}(r_f)$. Observe that, for every region r in $R_\circ$, we have $\hat{h}(r) > \hat{h}(r_f)$ due to convergence satisfying Equation 3.4. Therefore r' cannot be in $R_\circ$. Thus the search must eventually visit a region not in $R_\circ$, implying that it must eventually visit a region in R_F, in contradiction to its definition. ☐

Theorem 3. *Under Assumptions 3-6, a persistent and goal-aware real-time motion planning algorithm with dynamic programming-like learning will eventually reach a goal.*

Proof. If a goal is never reached, the algorithm has a circulating set by Lemma 6, which by Lemma 9 is not possible. ☐

Assumption 8. *RBPC breaks ties in Q by low h.*

Lemma 10. *Under Assumption 8, in the limit of computation speed RBPC is persistent.*

Proof. RBPC searches in order of increasing $f(v)$ (Alg. 2 line 5), breaking ties on low $h(v)$ (Assumption 8), choosing first vertex on the time horizon expanded to be the goal. In the limit of computation speed it is guaranteed to consider all reachable regions in local search, so it must find trajectories into the region with lowest f, or, in case of ties, lowest h, each local search, by Theorem 2 and Assumption 8. Thus it is asymptotically persistent. □

Corollary 1. *Under Assumptions 3-8, in the limit of computation speed, if RBPC uses dynamic programming-like learning it will eventually reach a goal.*

Proof. RBPC is an asymptotically persistent (Lemma 10) and goal-aware (Lemma 2) real-time (Theorem 1) motion planning algorithm, and if it uses dynamic programming-like learning it is probabilistically complete under Assumptions 3-8 via Theorem 3. □

To the author's knowledge, this is the first proof of probabilistic completeness of this nature for a real-time motion planning algorithm. With these properties: real-time, asymptotic optimality, and probabilistic completeness, we can feel confident using RBPC for planning in the path coverage problem.

CHAPTER 4

Approximate Model Predictive Controller

A motion planner is essential for autonomous ocean surveying, but, as designed, it is not the only required component. We allow the planner to be separated from reality by some simplifying assumptions that will not hold in the real world, which necessitate a controller to execute low-level actuation. Several sources of disturbance are assumed not to exist when planning trajectories. Actuation noise and waves, which affect the boat in difficult-to-predict short-term ways, make it unwise to pursue a single control for the duration of a planning iteration. Wind and current act together for a more consistent, long-term disturbance. While the Dubins car has been adapted to account for a consistent wind force (Techy and Woolsey 2009), we choose to allow a controller to estimate and account for external forces. Running a controller at a higher frequency than the planner enables it to react to short-term disturbances. For exmaple, RBPC is run at 1Hz with a 5s minimum trajectory duration, while the controller is designed to follow plans by issuing rudder and throttle controls at 10Hz.

Additionally, specific to a Dubins-based motion planner, it is helpful to allow the planner to assume the vehicle has not deviated from the last plan. The non-robustness and numerical instability discussed in Chapter 3 necessitate this abstraction, as it frees the planner to re-use a previously computed Dubins curve, instead of attempting to calculate a similar one from scratch.

We design an approximate model-predictive controller to fill this role in our autonomous ocean surveying system. This controller receives a motion plan, hereafter referred to as a *reference trajectory* in this chapter, from the planning algorithm, and issues low level controls at a high frequency in order to follow this trajectory as closely as possible. The controller uses a model of the vehicle to simulate various controls and compares the resulting trajectories to the reference trajectory with a customizable scoring function. The initial controls in the best-scoring trajectory

are issued to the robot, and the process begins anew with an updated starting pose. Simulating and scoring trajectories allows the controller some limited lookahead with which it can anticipate difficult maneuvers and choose controls likely to lead to long-term success. The reader is directed to `https://youtu.be/2-fAVah2YMg` for a video which demonstrates effects of this lookahead. This video shows the controller following several trajectory segments. The segments are pieced together in a way which is not dynamically feasible. By looking ahead, the controller discovers that it can most closely match the trajectory corners by turning early, effectively cutting the corners. This is preferable to a controller without lookahead. A PID controller, for example, cannot anticipate the corners in this way, but instead corrects afterwards. This video, `https://youtu.be/MJcPnscloy8`, shows a PID controller driving a square pattern, and this video, `https://youtu.be/HQtPtGHUBKM`, shows the model-predictive controller driving the same pattern.

The controller simulates and scores trajectories at a fixed time step, and extends lookahead as time allows and future reference trajectory remains. It issues controls at a fixed frequency, and simulates trajectories continually in the meantime, to take full advantage of the time it has. Algorthms 4 and 5 show the structure of the computations it performs. Algorithm 5, **SimulateTrajectories**, is shown here recursively for clarity, but in practice, it is implemented in a loop mostly without dynamic memory access, for speed.

Algorithm 4 MPC

Input: *start*

1: $depth \leftarrow 1$
2: $bestControl \leftarrow \emptyset$
3: **while** $TimeRemaining() \wedge depth * timestep \leq ReferenceTrajectoryTotalTime()$ **do**
4: $\quad bestControl, _ \leftarrow$ **SimulateTrajectories**($start$, $depth$)
5: $\quad depth = depth + 1$
6: **return** $bestControl$

Trajectories are explored in a depth-first iterative deepening order (Alg. 4 lines 4-5), and controls picked from longer simulated trajectories are always preferred over controls from earlier depths (Alg. 4, line 4). At each depth of search, controls are chosen by the function *GetControls*, explained below, and are applied to the current state s (Alg. 5 line 6). The field *timestep* is a

Algorithm 5 SimulateTrajectories

Input: s, d

1: **if** $d \leq 0$ **then**
2:     **return** $\emptyset, 0$
3: $bestScore \leftarrow \infty$
4: $bestControl \leftarrow \emptyset$
5: **for** u in $GetControls()$ **do**
6:     $s' \leftarrow Simulate(s, u, timestep)$
7:     $_, c \leftarrow$**SimulateTrajectories**$(s', d-1)$
8:     $c \leftarrow c + Score(s')$
9:     **if** $c < bestScore$ **then**
10:         $bestScore \leftarrow c$
11:         $bestControl \leftarrow u$
12: **return** $bestControl, bestScore$

global parameter to the controller, and controls the granularity at which the simulated trajectories branch and are scored. The recursion in Alg. 5 finds the best score for trajectories passing through s' (Alg. 5 line 7). If the score for s' plus the score for the best trajectory passing through s' (Alg. 5 line 8) is lower than the best score seen thus far (Alg. 5 line 9), the current score and control are saved (Alg. 5 lines 10-11). Finally, the best control and score are returned (Alg. 5 line 12). The loop in Alg. 4 line 3 is bounded by both wall clock time and reference trajectory depth: the controller must run at a fixed rate, and simulated states beyond the end of the reference trajectory cannot be scored.

The branching factor of the depth-first search is equal to the number of unique controls considered by *GetControls*. A naive approach would pick controls evenly across the control space, but this creates a tradeoff between lookahead and granularity: more controls means a higher branching factor, which could reduce the final depth of the trajectory search. Instead, we propose a more limited set of controls which we believe still captures enough good possible controls at each depth. This limited set is motivated by a hypothesis about our application domain, autonomous ocean surveying. In this domain, the robot most often pilots straight along a survey line while fighting actuation noise and consistent external disturbances. We believe that, in this case, the best control is often similar to the previous control. With this in mind, we design the limited control set relative to the previous control. The limited control set contains all extreme controls, the previous control,

and neighboring controls to the previous control, as defined by a fixed discretization or granularity. We find this works well in practice to allow a fine granularity of controls without sacrificing lookahead depth. Granularity values and their effects are discussed in Appendix A.

The calculation in the function *Score* is integral to good behavior from the model-predictive controller. In our ocean surveying domain, *Score* compares simulated states to their counterparts on the reference trajectory as a function of Euclidean distance, heading difference, and speed difference, shown in Equation 4.1. The two parameters in Equation 4.1 are states being compared, one of which is s' in Alg. 5 line 8, and the other is s''s time-equivalent counterpart on the reference trajectory. The result, shown in Equation 4.1 is exponential in Euclidean distance and linear in the other two terms, exposing a weight for each term to the user.

$$\mathit{score}(s_1, s_2) = w_d * e^{\mathit{distance}(s_1, s_2)} + w_h * \mathit{headingDiff}(s_1, s_2) + w_s * \mathit{speedDiff}(s_1, s_2) \tag{4.1}$$

This flexibility allows for easy reconfiguration of the scoring function, and thus the behavior of the controller, when environmental conditions change. As mentioned in Chapter 3, the controller determines whether the reference trajectory is achievable. It determines this with a user-defined threshold in terms of the scoring function: only trajectories that score below the threshold are considered achievable. A good threshold is likely to be dependent on environmental factors, and could change during operation, because it is a measure of how confident the controller should be in its ability to achieve reference trajectories. In rough conditions, it should be lower, to prevent to robot getting pushed too far off course, but in generally good conditions, when similar disturbances are recoverable, it should be higher, to let the planner re-use plans more often. Example weights and threshold are provided in Appendix A, with some more specific discussion about how they affect behavior.

External disturbances are measured and estimated, in order for the controller to react appropriately. When controls are issued, the current state and issued control are recorded in a buffer with fixed size. When a disturbance estimate is desired, the oldest state in the buffer is simulated to the present time by applying each control for the duration it was applied to the robot (until the next state in the buffer, or the present time), assuming no external disturbance. The difference between

this final state and the current state provides an estimate of disturbances affecting the robot over time. This estimate is used to adjust the simulated state s' (Alg. 5 line 6) in order to increase the accuracy of simulation. The buffer, once full, acts as a shifting time window over which disturbance is estimated.

C++ source code for the controller is available at `https://github.com/afb2001/mpc`. A playlist of controller-related videos can be found here: `https://www.youtube.com/playlist?list=PLNcViETgkSde4xr7hfMH920t1a2w8D1-h`.

CHAPTER 5

Architecture for Autonomous Ocean Surveying

5.1 Executive

In addition to a planner and controller, we implement an *executive* to supervise solving the path coverage problem. For the rest of this thesis we will use the term *executive* to refer to this piece of software. The executive is responsible for monitoring path coverage and robot pose as they change in real time, and calling the planner with up-to-date parameters. It loads maps, monitors dynamic obstacles, handles errors from the planner, and facilitates planner-controller communication.

As planning takes time, the starting pose given to the planner should be a future estimate of the pose after planning time has elapsed. The best way to get such an estimate is from the controller, which, as discussed in Chapter 4, simulates trajectories with different controls for as far a lookahead as time allows. Since a new pose estimate is required to start a planning iteration, right after the previous iteration has finished, the controller's reference trajectory has just been updated. Thus we must wait one full controller cycle before an updated estimate can be obtained. Once the cycle is completed, the controller passes the estimate to the executive, which feeds it along to the planner. If the controller returns an error, the executive can naively estimate the pose instead, from the current pose. When the controller declares the reference trajectory is feasible, the executive passes a pose taken directly from the last plan instead, allowing the planner to extend the previous plan directly.

5.2 Project11

There is substantial other architecture necessary for autonomous ocean surveying which we will briefly describe in this section. The components discussed here make up the Project 11 framework

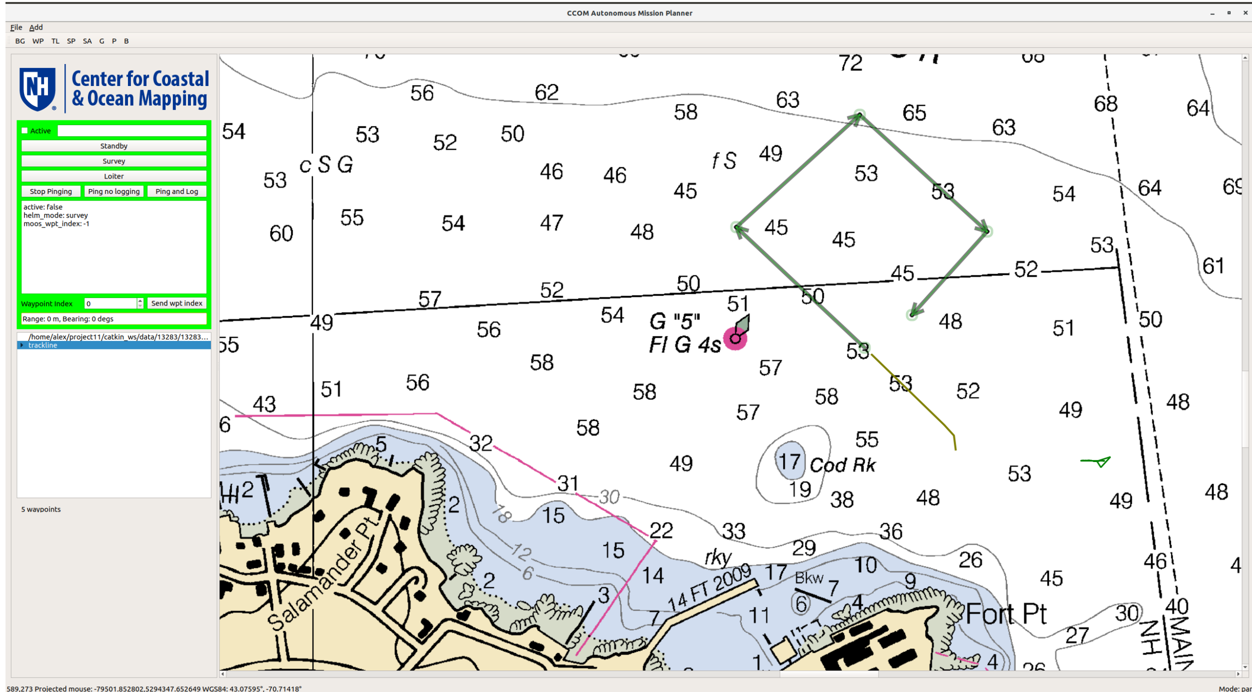


Figure 5-1: User interface for controlling autonomous vehicles in Project 11.

(Arsenault and Schmidt; Arsenault 2020; 2020). Project 11 was developed by the University of New Hampshire Center for Coastal and Ocean Mapping (CCOM) to provide a framework for marine robotics. Project 11 is built with the Robot Operating System (ROS) middle-ware for robotics, so it consists of several relatively independent nodes which communicate with one another through topics and services. An illustration of this communication relevant to the work presented in this thesis is shown in Figure 5-2. Communication passing through the UDP Bridge in Figure 5-2 is communication between the robot and the operator station. In simulation this communication still happens, but it is local.

The graphical user interface for *Project 11*, CCOM Autonomous Mission Planner (CAMP) (Arsenault 2020), has been fine-tuned over many field operations to provide streamlined mission planning capabilities to operators. Figure 5-1 shows a screenshot of the interface, displaying an autonomous vehicle and some user-drawn survey lines. The user can easily draw and change track-lines and generate survey patterns with a three-click interface. It displays the ASV and dynamic obstacles in the vicinity overlayed on nautical charts, bathymetry grids, etc. Its additional visualization capabilities have been very helpful for the development of the work in this thesis:

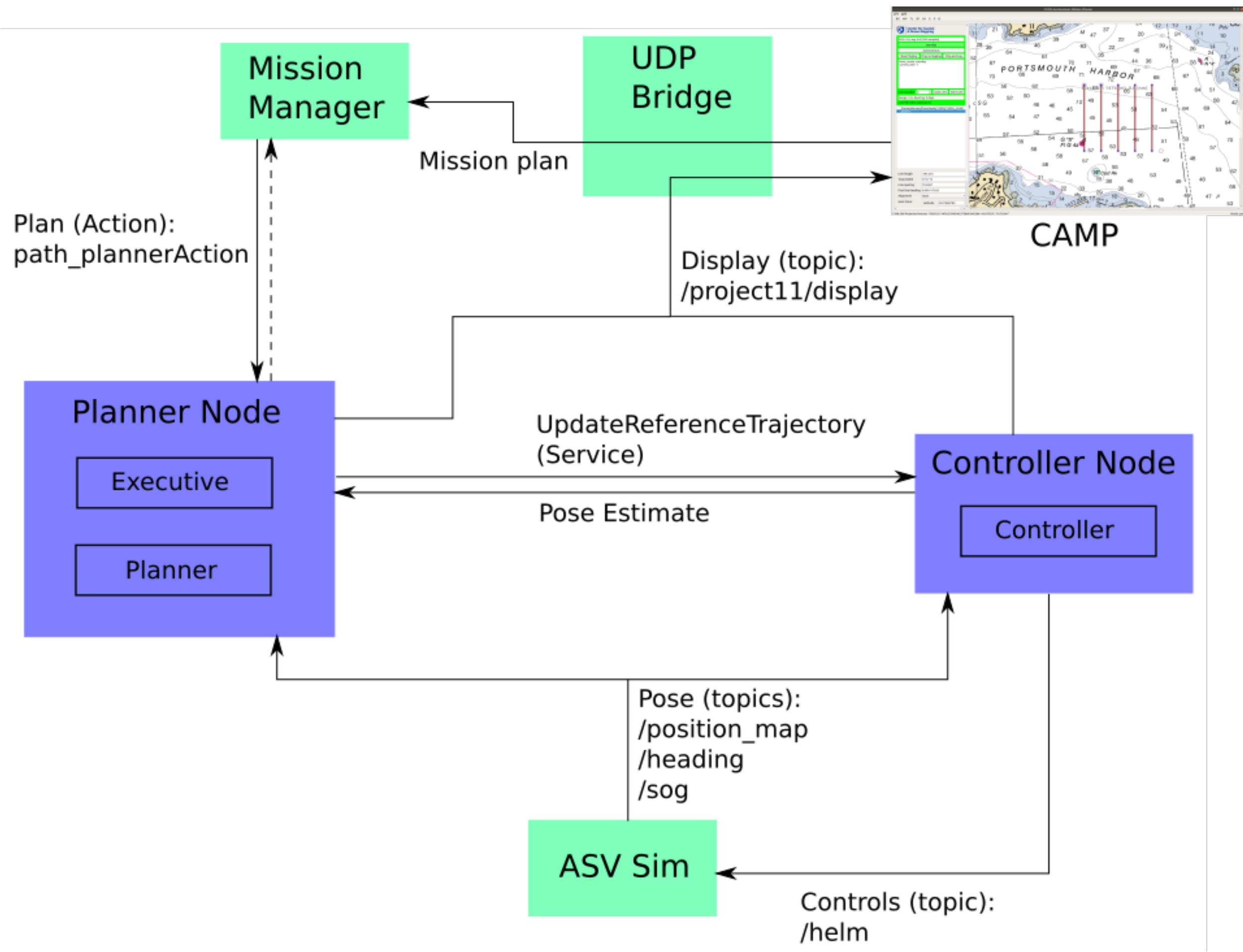


Figure 5-2: Overview of communication between ROS nodes in the relevant portion of Project 11.

CAMP has a straightforward API for displaying points, lines and arbitrary polygons in real time.

The high level state of the vehicle is modeled in a state machine housed in the mission manager (Arsenault 2020). The mission manager handles survey requests sent by the user from AMP, and passes them to the path coverage system or to a PID controller. It is also responsible for handling the vehicle's behavior after a survey has been completed.

The simulator (Arsenault 2020) is extremely helpful for motion planning algorithm development, as it attempts to model the dynamics of various ASVs in real time. It publishes to the same topics as the true vehicles, which makes transitioning between simulator and reality seamless. It allows the user to configure dynamics, actuation noise, and currents on the fly. All experiments described in the next chapter are run in this simulator.

There are many other components to this system, but they are not directly relevant to the work presented in this thesis. The curious reader is encouraged to visit `https://github.com/CCOMJHC/project11/blob/master/documentation/Installation.md` for more details, and to download and install the simulation environment for themselves.

CHAPTER 6

Experimental Results

6.1 Experimental Setup

We present the results of several experiments to demonstrate the behavior of our algorithm in a simulated ocean surveying environment. The experiments are run on a Intel(R) Core(TM) i5-4690K CPU @ 3.50GHz with 16GB of RAM. Notably, the planner and controller each take up one of the four cores in the system, and the surrounding ROS framework requires computational effort as well, so performance is expected to drop significantly in systems with fewer than three available cores. Neither the planner nor controller is implemented to take advantage of more than one core, so systems with more cores will not see much better performance.

We evaluate the system on a set of scenarios of interest. The scenarios can be split into two categories. There are scenarios which each test for specific behavior, for example, cannon_go_around (Figure B-8) tests whether the robot can find a plan that goes extra distance around a static obstacle to avoid a large dynamic obstacle. These specific scenarios are all present in Table 6-1, and vary in line number, relative line positioning, dynamic obstacle number and relative positioning, and static obstacles. There are also scenarios which test basic line following with a variety of dynamic obstacle configurations and sizes. For example, Test05 (Figure B-23) requires the ASV survey a line with a dynamic obstacle approaching from the right of varying sizes. These scenarios are all present in Figure 6-3. For a detailed description of each test scenario, see Appendix B. There are several parameters to the system, which makes it infeasible to exhaustively evaluate the parameter space, but for the experiments shown here the system uses a default configuration unless otherwise specified. The default configuration is the result of experimentation during the development of the system. This process, as well as the parameters themselves, is discussed in Appendix A.

We compare planners and configurations by keeping track of score, similar to how it is evaluated in collision checking, and report it over the whole trial. Every 1s planning cycle the robot incurs collision penalty according to its relation to dynamic obstacles. This total penalty is added to the score upon completion of a trial. Scenario completion occurs only when the robot has completed coverage required for the scenario.

Python 2.7 source code for the test harness is available at `https://github.com/afb2001/test_scenario_runner`. The test harness reads scenario descriptions from text files, calls the planner, and records results. The text file descriptions of all scenarios used in this chapter are located in the same repository. Videos of many of the scenarios can be found at this link: `https://www.youtube.com/playlist?list=PLNcViETgkSdceqn0dB9kG2_fTC_lEgDra`.

6.2 Comparing the Planners

In this section we compare the real-time motion planning algorithm RBPC to an artificial potential field based approach. The artificial potential field is implemented as a sum of forces acting on the robot. Path segment endpoints pull the robot towards them with magnitude inversely proportional to distance. Once the robot is on a path segment it is pulled strongly to the opposite endpoint. When the robot is far away from the endpoint, it is pulled towards a point 10m beyond the segment endpoint (in line with the segment), in order to help dynamic feasibility. If the robot were to be pulled directly to the endpoint, and then directly to the other side of the segment, it could be required to make an impossible turn to begin coverage. Dynamic obstacles exhibit a repulsive force proportional to the obstacle's area which decays exponentially with distance. Nearby static obstacles also exhibit an exponentially decaying repulsive force. C++ source code for the potential field planner can be found at `https://github.com/afb2001/path_planner/blob/master/path_planner/src/planner/PotentialFieldsPlanner.cpp` and the accompanying header file.

We note that artificial potential fields are a well-studied branch of robotics, and the implementation here is intended more as a baseline than a state-of-the-art comparison. There are several more advanced techniques to potential field construction that could allow a potential field to more successfully navigate scenarios presented in this chapter. For example, to avoid local minima in the

	100mNorthGauntlet	adjacent_multi_line	adjacent_single_line	ahead_long
RBPC	10	10	10	10
Potential Field	10	9	10	10
	cannon_crossfire	cannon_cut_across	cannon_follow	cannon_go_around
RBPC	10	10	10	10
Potential Field	10	9	10	0
	cannon_wait_1	cannon_wait_2	dynamic_wall_1	long_single_line
RBPC	10	10	10	10
Potential Field	0	10	10	10
	test01	test02a	test03	test04
RBPC	10	10	10	0
Potential Field	10	10	10	0
	test12a	test12b	test12c	narrow_wide
RBPC	0	10	0	10
Potential Field	0	0	0	0

Table 6-1: Number of completed instances out of 10 trials for several scenarios comparing RBPC and Potential Field

static map, one could run the wave-front planner, described in Choset et al. (2005), to the nearest line endpoint, following lines in a nearest-first order. This approach explicitly separates obstacle avoidance from navigation: the robot always navigates towards the nearest line endpoint, avoiding obstacles along the way. Because RBPC combines navigation and obstacle avoidance in a single objective, we restrict the potential field to do the same, but a more sophisticated potential field could separate them to great benefit.

Table 6-1 shows the number of instances out of 10 completed within a 600s time limit for both algorithms. Table 6-1 shows only a subset of the scenarios which are small enough to be completed in 600s. It should be noted that scenarios test04 (Figure B-17), test12a (Figure B-18), and test12c (Figure B-20) are unsolvable, as a path segment passes through an obstacle. See Appendix B for descriptions of each scenario. Test12b (Figure B-19) and narrow_wide (Figure B-22) are notable because they force the vehicle to escape a seawall which separates the starting pose from the path segment. narrow_wide is further complicated by a narrow corridor which must be traversed before the seawall. The potential field approach is unable to guide the vehicle to complete either scenario, but RBPC successfully does so 10 times out of 10. narrow_wide is particularly challenging to a potential field in general because it requires different static obstacles to be treated separately:

the seawall must be circumvented, but the allowable proximity to obstacles must be low to fit through the corridor. Scenario cannon_wait_1 (Figure B-9) is also notable, because the potential field planner is forced to run aground by a large dynamic obstacle. RBPC plans ahead and is able to avoid such states of inevitable collision.

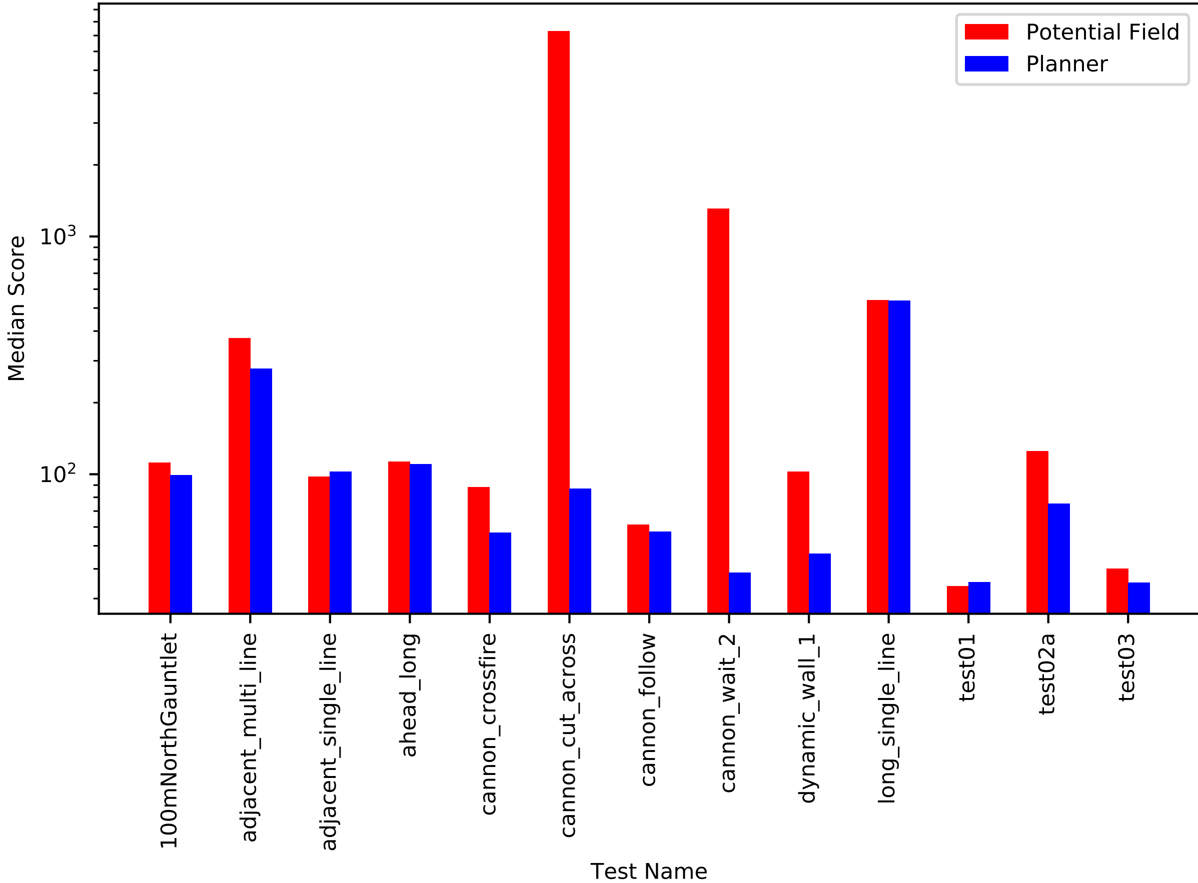


Figure 6-1: Plot showing score of the potential field planner and of the RBPC planner.

In addition to completion rates, we also study scores of test instances completed by both algorithms. Figure 6-1 shows a comparison of median scores on the same scenarios as Table 6-1, generated from 10 trials for each scenario. In general, RBPC tends to score slightly better than potential field, and scores substantially better in cannon_cut_across (Figure B-6) and cannon_wait_2 (Figure B-10). In the scenarios where the score difference is small, RBPC was able to find better

	Potential Field	RBPC
Achievable Percentage	80.629 %	96.752 %

Table 6-2: Percentage of plans from each algorthm which the controller deemed feasible.

plans than potential field, which lead to the earlier completion of the scenario. In the instances with large differences, the potential field planner collided with dynamic obstacles, incurring substantial collision penalty.

A big factor in these differences, gleaned from observation of the potential field planner, is that the plans from RBPC are dynamically feasible, whereas the potential field plans are often not. This can be seen in the fraction of achievable reference trajectories for each system, shown in Table 6-2 and Figure 6-2, which both represent the same trial set as Table 6-1. Figure 6-2 is broken down by scenario, and Table 6-2 is averaged over all trials. Dynamic feasibility is a major advantage of RBPC.

We study several more scenarios separately from the ones presented above. These are more realistic ocean surveying scenarios, where an ASV is tasked with following a single line with one or more dynamic obstacles of varying sizes interrupting the survey. They are only recorded once for each system. These scenarios are designed to be very challenging for a real-time system, as they specify dynamic obstacles which must be avoided by a very large margin. RBPC uses a 60s time horizon for these scenarios. A score comparison between RBPC and potential field in these scenarios is shown in Figure 6-3. Scenarios are structured such that testXXa - testXXe are similar scenarios but with increasing dynamic obstacle clearance required. A substantial difference in score for RBPC is evident between the *a* scenarios and the *c*-*e* ones. This is because the time horizon in RBPC is large enough to find plans which avoid the avoidance area entirely in the *a* scenarios only, and able to quickly escape the avoidance area in many of the *b* scenarios. The potential field implementation was re-tuned to avoid obstacles in select *a* scenarios for this experiment. For several *a* scenarios it performs well, avoiding obstacles entirely (albeit in a less efficient manner than RBPC). but it fails in the larger scenarios, willingly driving through the clearance zone. In the very largest scenarios this ends up being better than RBPC.

We compare the planning systems in two final scenarios. The maze scenario, albeit unrealistic,

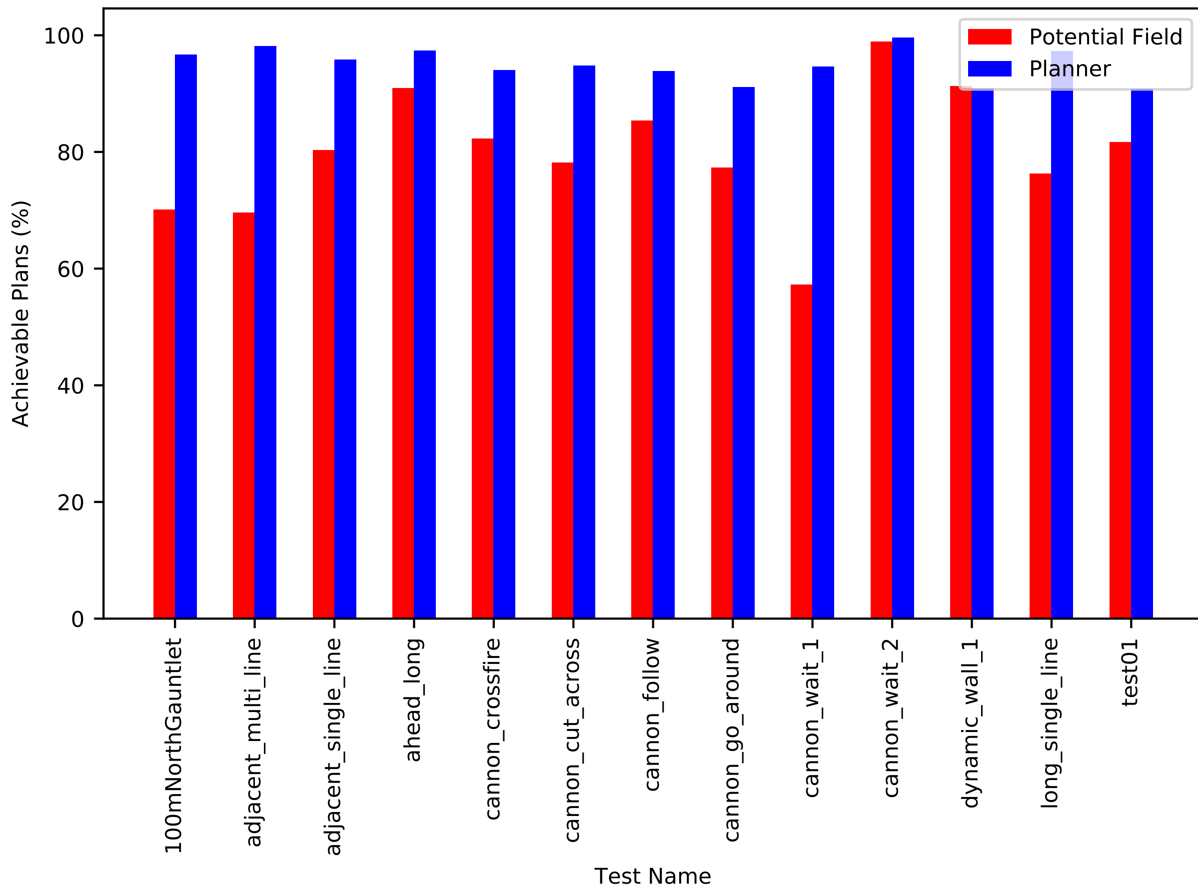


Figure 6-2: Plot showing percentage of plans from each algorithm which the controller deemed feasible.

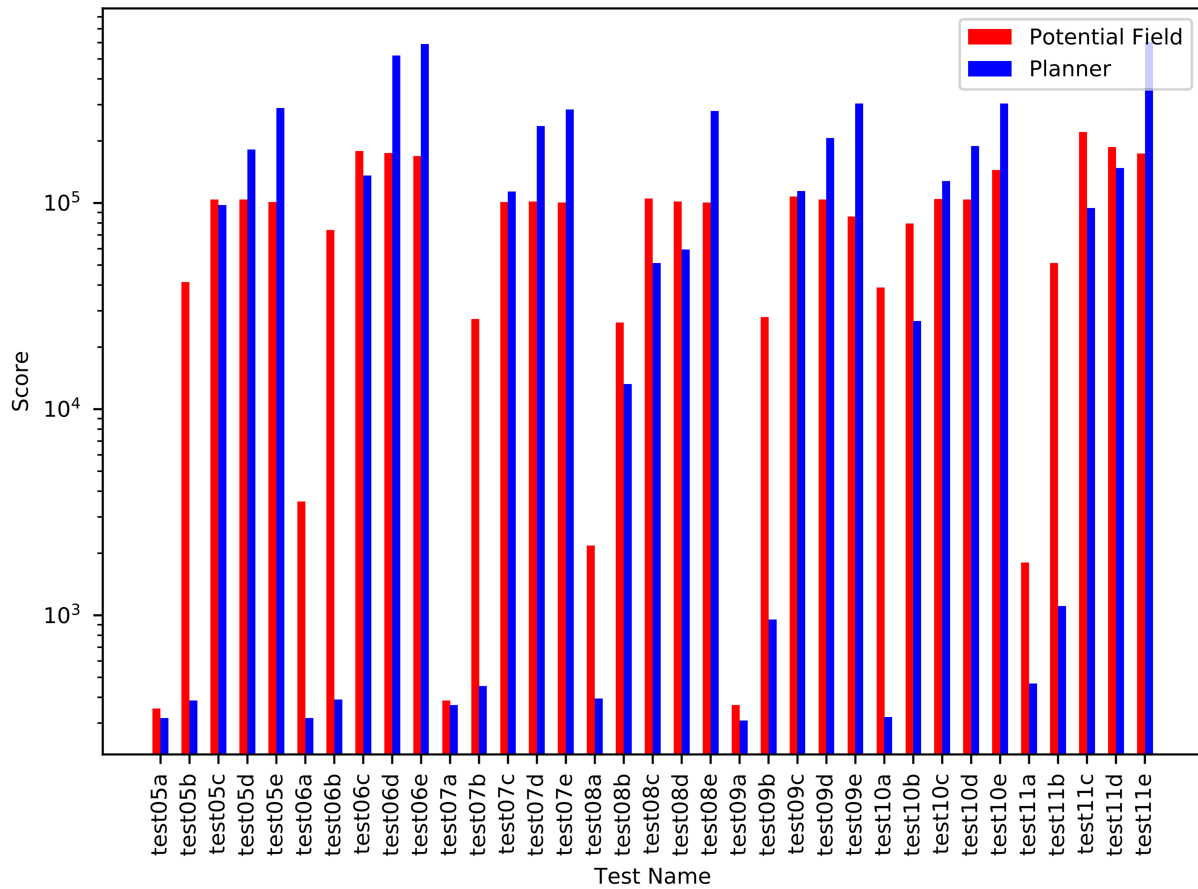


Figure 6-3: Plot showing score of the potential field planner and of RBPC in more realistic ocean surveying instances instances.

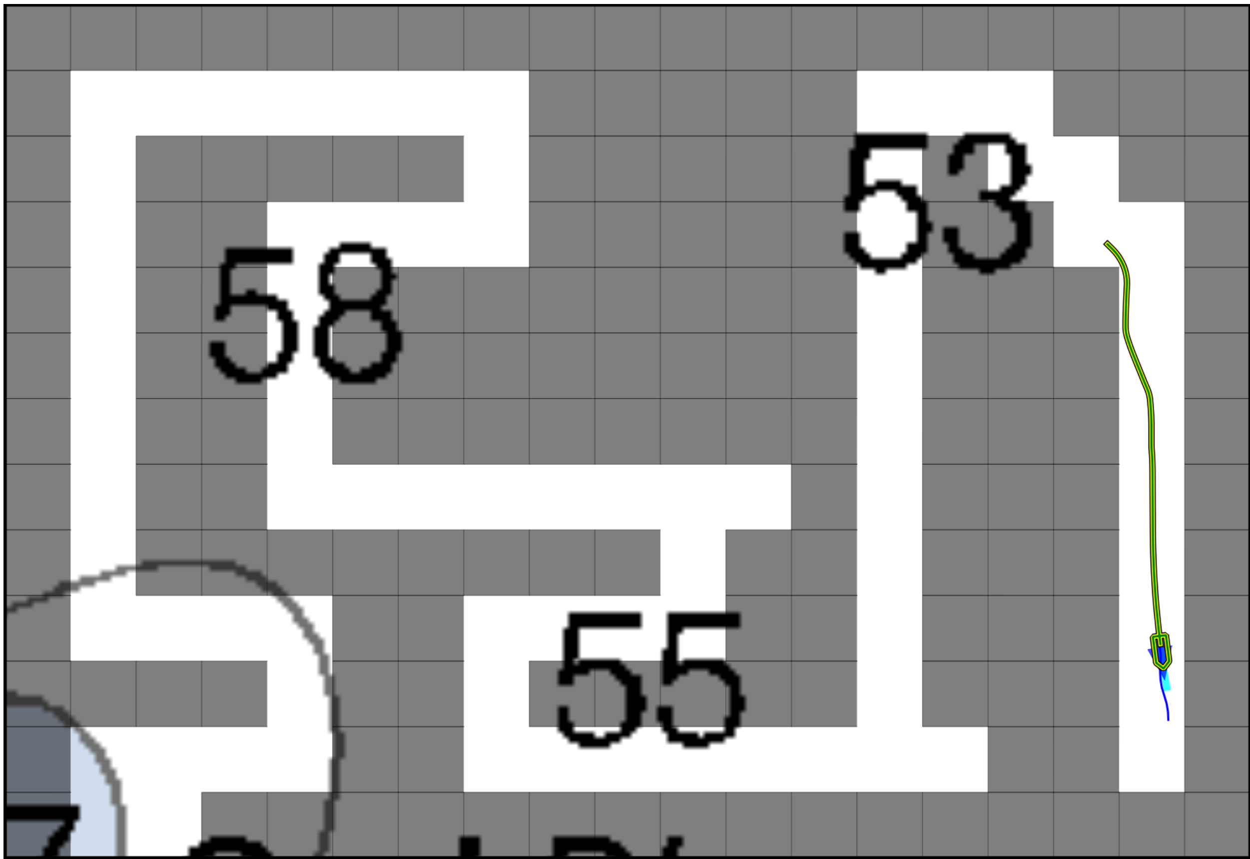


Figure 6-4: Maze map. Grey squares are blocked.

demonstrates RBPC's ability to discover feasible plans in a tight space. Figure 6-4 shows the vehicle completing this maze scenario in simulation. The path segment in this scenario is located in the far right corridor, vertically, where the vehicle's past trajectory can be seen. The robot starts in the bottom left. The potential field planner is unable to navigate for more than a few seconds in this scenario before colliding with the static obstacles. It plans to go to the right, towards the path segment, then turns abruptly upwards as the force from the wall pushes it back. The controller is not able to follow this plan well enough to avoid crashing into the wall. RBPC has a little trouble finding plans, but not enough to prevent it from completing the scenario. A video of RBPC in this scenario can be found at this link: `https://youtu.be/JDtHwSC_Eio`.

In addition to illustrating the advantage of dynamically feasible plans, this scenario demonstrates the algorithm's ability to find plans which obey a more complex cost function than the other scenarios require. Finding plans which avoid the numerous static obstacles requires a dense sampling of the space. While asymptotic optimality guarantees we will find optimal plans eventually, it is reassuring that RBPC can find such sophisticated plans in practice, and important to note for future extensions of this research.

The Pepperrell Cove scenario (Figure 6-6) requires the ASV to navigate through a crowded mooring field to survey the requisite path segments. RBPC is able to do this with little difficulty, scoring 655.25, as the ships are not so close as to prevent safe passage between them in most cases. A video of RBPC completing the scenario can be found at `https://youtu.be/XMzVnDoXGmY`. The potential field takes much more time to complete this scenario, scoring 845.95. It circumvents obstacles in a less efficient way, because it is reacting to their presence rather than planning around them. A video of the potential field completing the scenario can be found at `https://youtu.be/W_P8uRySxT0`.

6.3 Experimenting with Parameters

While the default configuration performs well, we explore some different options for some of the parameters in this section through direct comparison. The parameters to experiment were chosen because they intuitively might have a large effect on the performance of the system. As in Chapter

Figure 6-5: Satellite image of Pepperrell Cove. The moored boats in the harbor present a navigational challenge for an ASV.

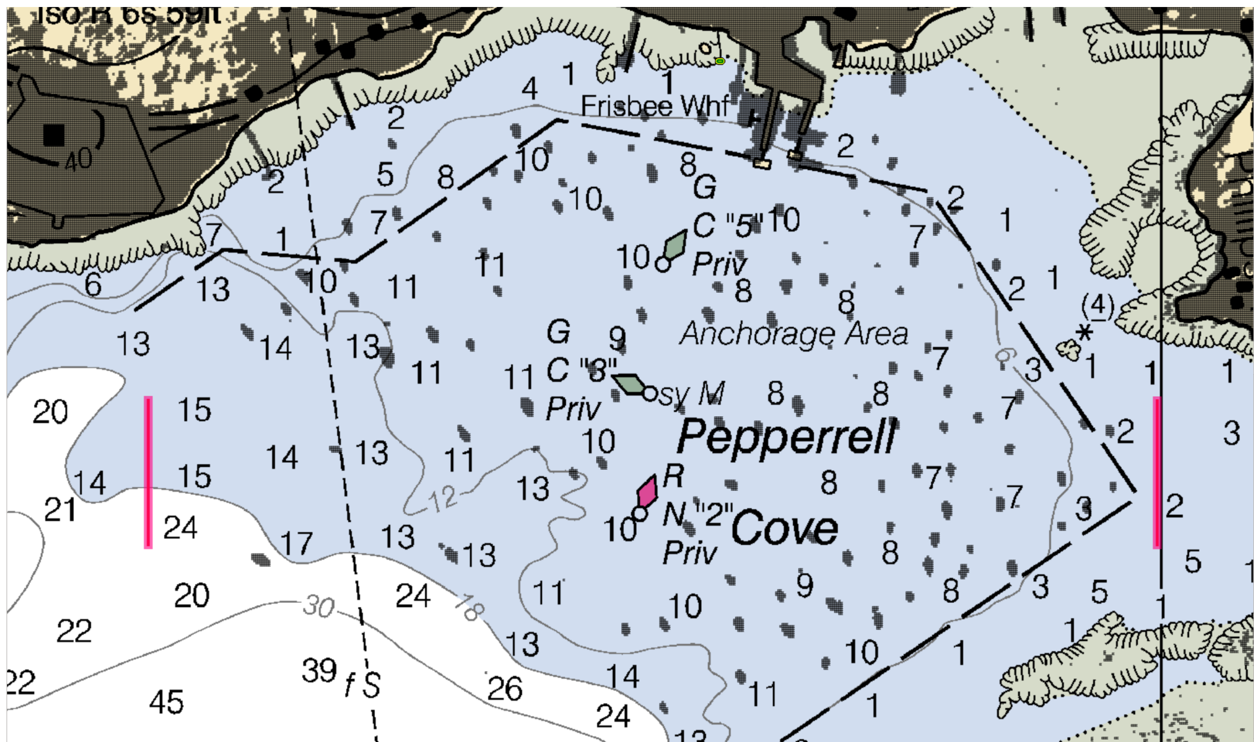


Figure 6-6: Pepperrell Cove scenario visualized in CAMP. Static obstacles, shown in grey, are taken from the satellite image to represent the coastline and moored boats in the harbor. Survey lines appear in the East and West, and the ASV starts the scenario next to Frisbee Wharf.

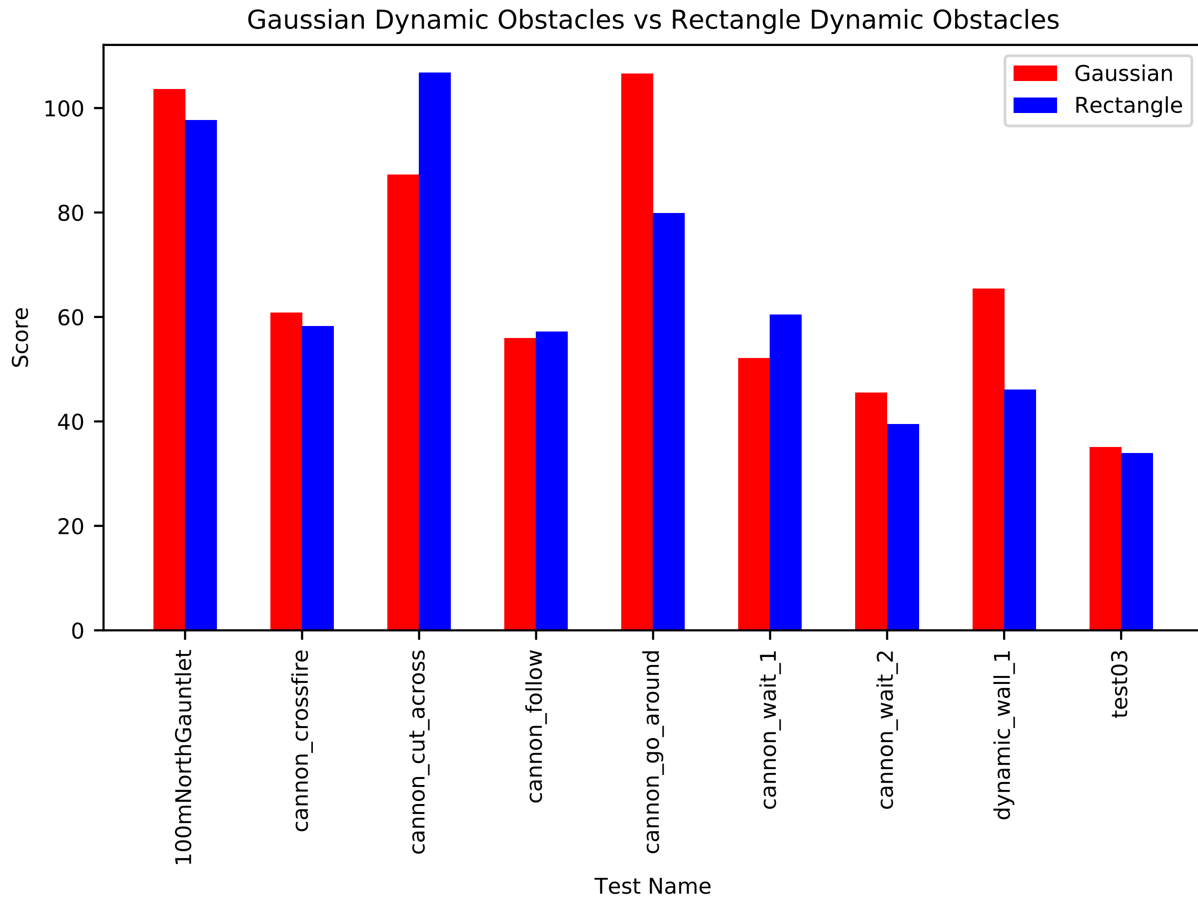


Figure 6-7: Plot showing scores with different dynamic obstacle representations in relevant scenarios.

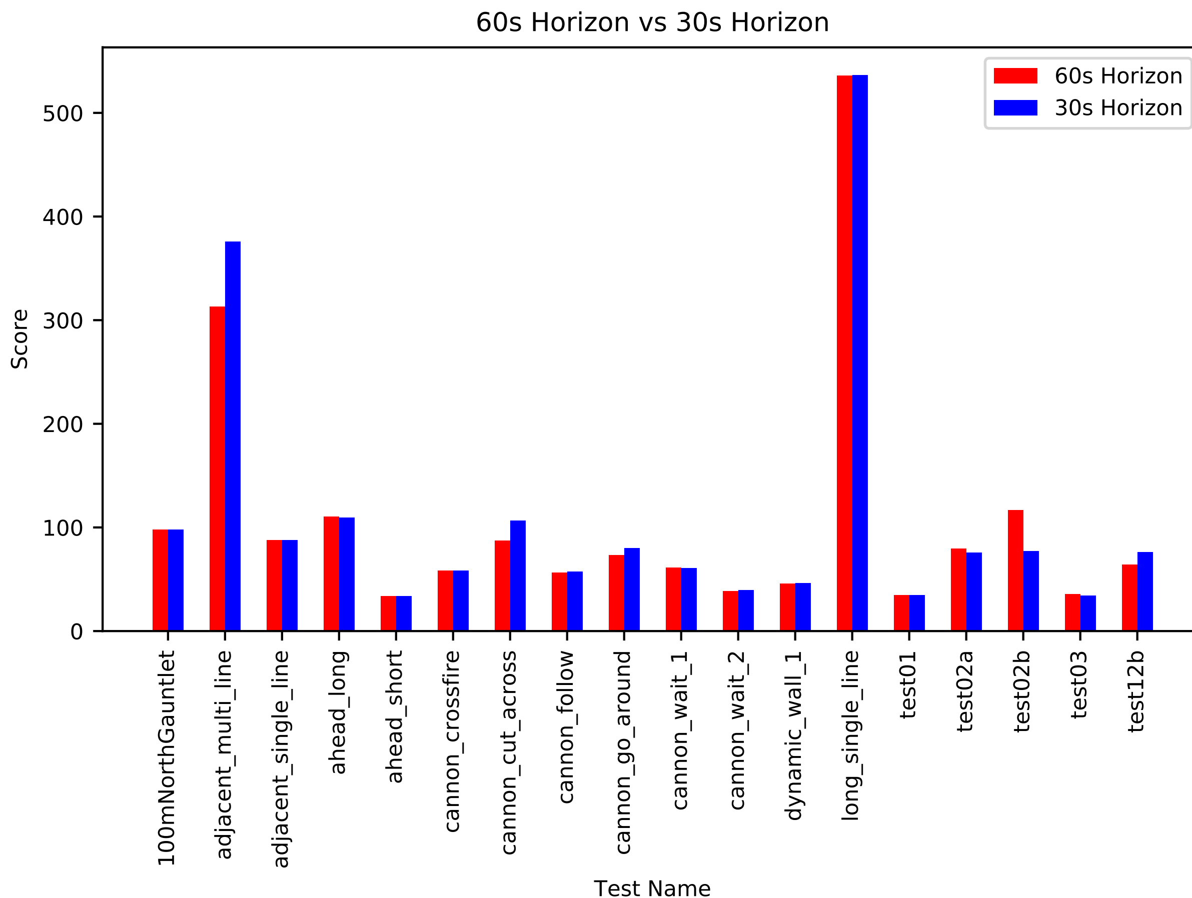


Figure 6-8: Plot showing score with different time horizons.

3, we discuss two different dynamic obstacle representations. Figure 6-7 shows cost incurred in various scenarios comparing the two representations: Rectangle, the rectangle fixed penalty model, and Gaussian, the probability density scaling penalty model. As seen in the figure, the rectangle representation scores slightly better for most of the scenarios, but is outscored in others. We note that the Gaussian representation is rather more difficult to tune, because the probability density function, and thus the collision penalty, depends on the covariance, which should vary in real world situations. Integrating this function into the objective function requires making decisions about how probability density should affect cost, and such decisions should be backed up by an expert observer judging behavior in simulation. We use the rectangle representation in all other scenarios, for simplicity and consistency.

The planning time horizon has a direct effect on the amount of computation required for each

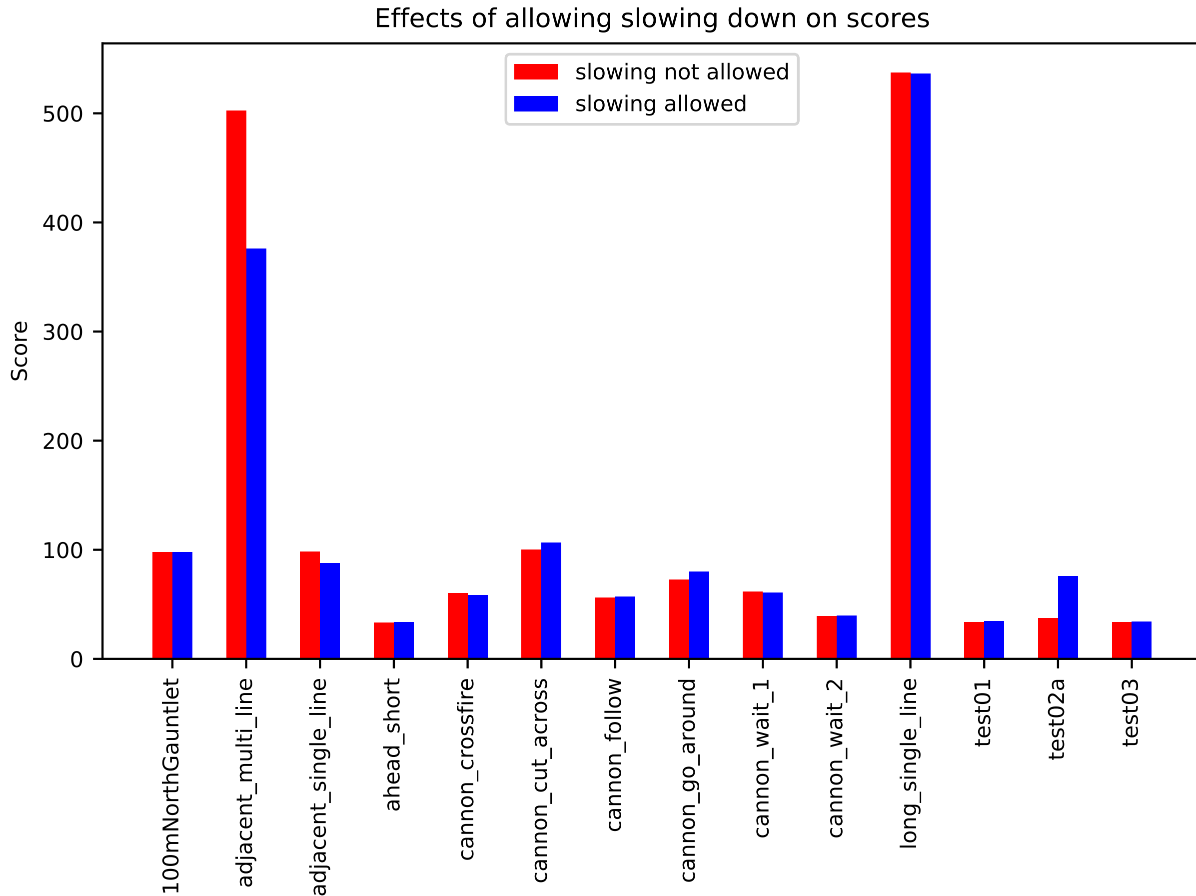


Figure 6-9: Plot showing score with slowing down allowed or not allowed.

sampling/search iteration. While a longer horizon should be beneficial in the limit of computation speed, on real hardware, extending the horizon limits the granularity of search which can be done under the time limit. Figure 6-8 shows a comparison of a 30s horizon (default) and a 60s horizon. While the longer horizon seems to maintain a slight advantage, they perform roughly the same on most scenarios. In *adjacent_multi_line* the performance margin is large, which indicates that a longer time horizon may be more helpful in multiple-survey-line scenarios. All other scenarios require a single line.

As mentioned in Chapter 3, the motion planner is allowed two different speeds, to help it avoid dynamic obstacles in a safe and predictable way. In Figure 6-9 we can see the effects of disallowing slowing down. While we would expect slowing down to help in the cannon_* scenarios in particular, the difference does not appear to be substantial or consistent. Scenario adjacent_multi_line has no

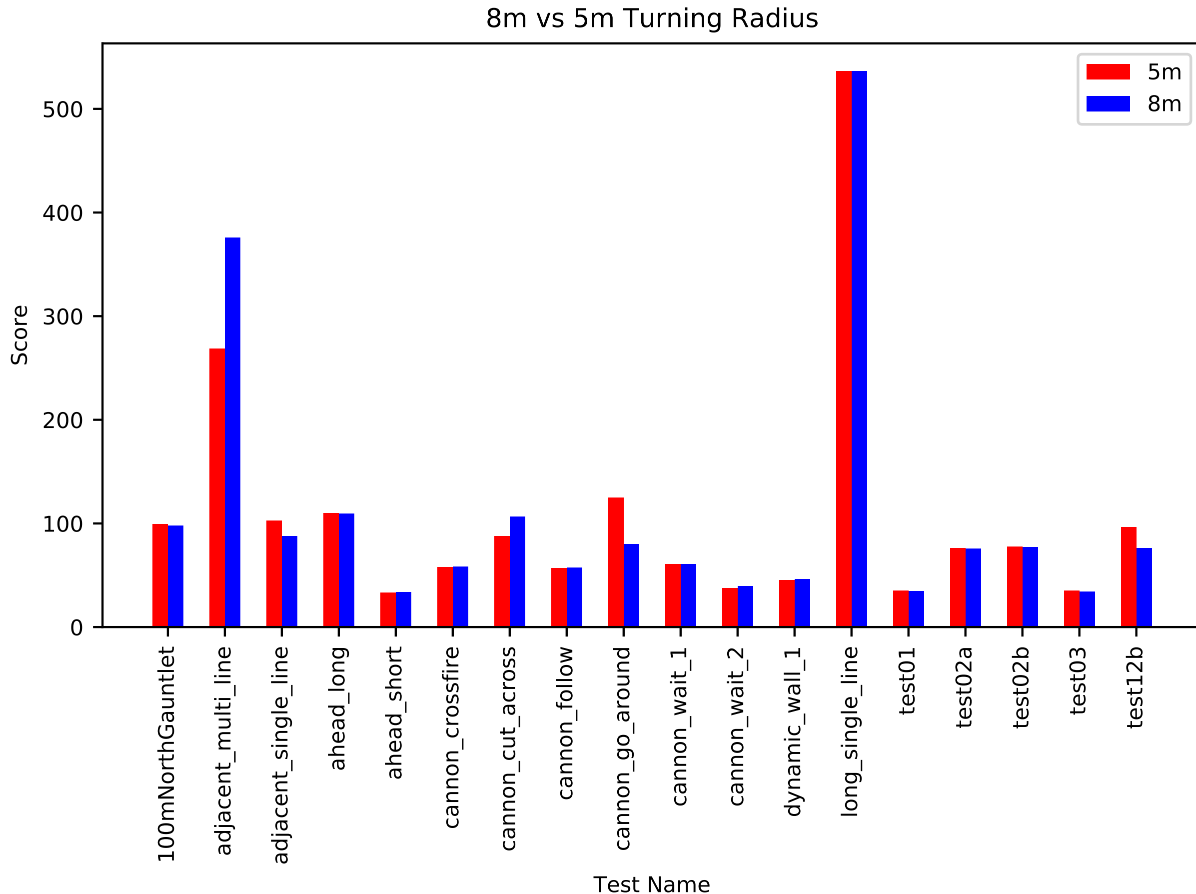


Figure 6-10: Plot showing score with two different turning radii.

dynamic obstacles, so slowing down should not affect the scores. This scenario seems to exhibit high variance in completion time.

While the default setting for the turning radius is 8m for the ASV used in simulation, as suggested by experts, we experiment with a smaller turning radius of 5m. Figure 6-10 shows a comparison of these two configurations. We once again see a large difference in adjacent_multi_line, and little difference elsewhere. Slightly more often, the larger 8m turning radius scores better, which indicates that 5m is not always dynamically feasible. This trend, although slight, can be seen in Figure 6-11, which shows, as a percentage, the number of achievable plans from each configuration.

In Figure 6-12 we compare the different heuristics described in Section 3.2. Sum computes the Euclidean distance to the nearest path segment endpoint and the sum lengths of the remaining paths. Euclidean TSP computes the optimal traveling salesman tour through the remaining path

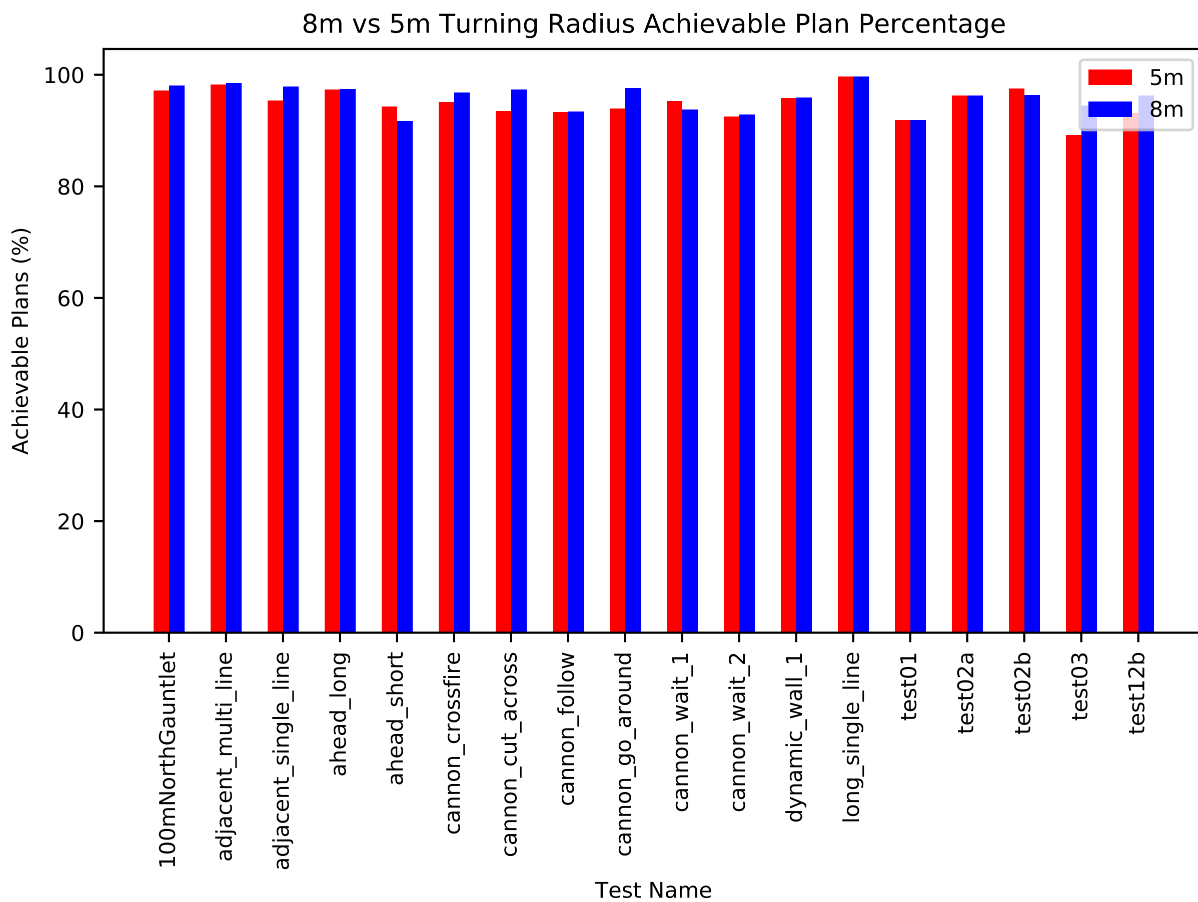


Figure 6-11: Plot showing percentages of achievable plans with two different turning radii.

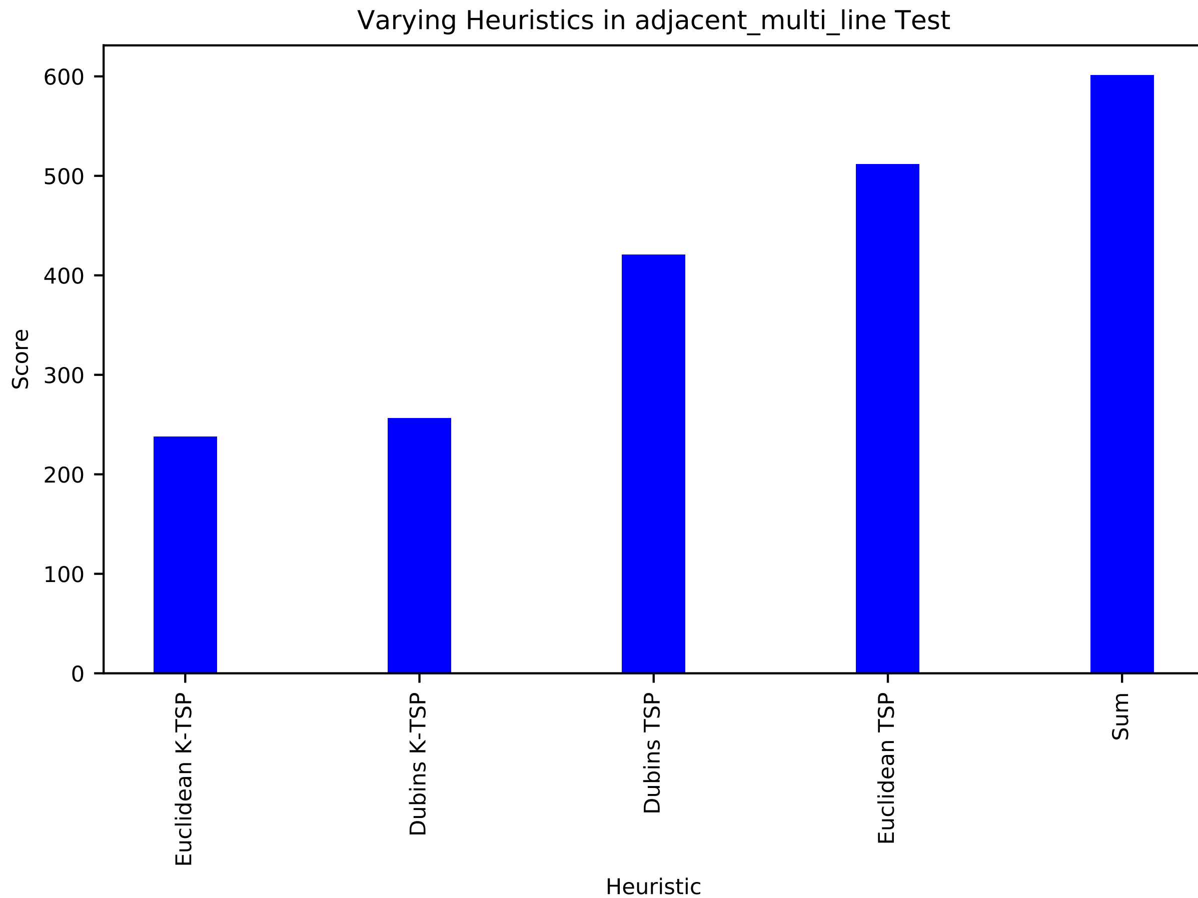


Figure 6-12: Plot showing scores in a single scenario with different heuristics

segments in Euclidean distance. Dubins TSP does the same in Dubins distance. Each K-TSP heuristic limits the branching factor to the two nearest path segments. While the K-TSP heuristics outscore their optimal counterparts, it should be noted that they are not admissible. Only Euclidean TSP and Sum are admissible.

CHAPTER 7

Conclusion

7.1 Discussion

The theoretical results presented in this thesis are important because, to the best of our knowledge, no prior work in motion planning claims to combine asymptotic local optimality, a real-time bound, and probabilistic completeness. While it is easy to extend an anytime motion planning algorithm to achieve asymptotic optimality by occasionally driving towards random poses, as in Kleinbort et al. (2019), algorithms which achieve this property often do well in practice. Real world applications of RBPC demand fast response times, because new and updated information about the robot and the environment can appear continuously, so it is important that we prove computation time can be bounded by a constant. While the heuristics with which we experiment cannot guarantee probabilistic completeness, by prescribing exactly the properties required, we bring it to the forefront of reachable extensions to this research.

There are four results from experimentation which are most compelling. RBPC is able to successfully escape local minima in its heuristic. This is an important property which requires foresight and reasoning, and which simpler approaches, including artificial potential fields, do not possess. Extending the time horizon in RBPC only extends this advantage to increasingly large local minima. Well-informed heuristics, as described in Chapter 3.4.3, allow RBPC to successfully navigate around any size obstacles. This result informs us that its performance will dominate that of approaches that avoid obstacles without optimizing for coverage: these approaches can be diverted indefinitely by an adversarial seawall.

By solving the maze scenario, we show RBPC is capable of finding complex paths in a confined space. While that scenario is not a very realistic one for the application of autonomous ocean surveying, finding collision-free paths in such a confined space suggests that the algorithm can

optimize a much more complicated objective than the ones which appear in the other scenarios. The suggestion is reinforced by our proof of asymptotic local optimality: we should eventually find solutions which best fit the objective function. This potential is particularly important when one considers extensions to the representations of dynamic obstacles, which may effect an arbitrarily shaped cost function. Solving a complex environment is very promising for the system's behavior in that context.

Dynamic feasibility is guaranteed in plans from RBPC under accurate parameters. Being able to quickly generate dynamically feasible trajectories for a nonholonomic robot which optimize a non-trivial cost function is not a simple task. This is a big advantage of RBPC over artificial potential fields, as can be seen in the previous chapter.

RBPC is also able avoid obstacles, instead of merely reacting to them, because it plans with lookahead, allowing the evolution of the environment to affect immediate decisions. This is extremely important because it helps the robot avoid situations where collision is inevitable, one of which occurred when the potential field planner attempted in cannon_wait_1. The time horizon is too small for the safe options to be visible to RBPC in most of the line following scenarios, but because the small-obstacle versions are solvable, with more lookahead, the larger ones should be too.

7.1.1 Limitations

Neither representation of dynamic obstacles perfectly characterizes the desired behavior for obstacle avoidance. The Rectangle representation implements a hard boundary for clearance, which is unrealistic. Penalty should scale smoothly with distance, at least for relatively far away obstacles. Also, obstacles are typically detected with some unceratinty, which the Rectangle representation is unable to capture. While the Gaussian representation scales penalty nicely with distance, and in some ways can capture uncertainty, it is not clear how to weigh probability density against time or distance in an objective. Such a weighting scheme should be agnostic to the covariance of an obstacle: the robot should still behave well regardless of the degree or shape of uncertainty. Conceptually the weighting scheme must decide which collision risks are acceptable and which are not.

One possible way to discover a weighting scheme is through automated learning in some scenarios with correct examples of behavior. The challenge of this approach is concocting a representative set of scenarios, so learned parameters perform well in new unseen cases.

The controller has parameters whose optimal values are also unclear. Scoring weights determine what good trajectories look like. For example, high weight in heading comparison means trajectories will tend to exactly match heading of the reference trajectory at the expense of distance or speed. The best behavior, and thus parameters, might depend on environmental conditions, so a well-informed policy might be necessary for good performance in a variety of conditions. Parameters for control granularity are also exposed, and their effects on behavior are even more opaque. As with dynamic obstacle parameters, machine learning on some representative examples of desired behavior could discover a very successful policy, but an optimal one is not known.

As discovered in the scenarios with dynamic obstacles with large clearance requirements, the time horizon is very limiting to realistic scenarios. 30-60 seconds, in a marine environment, is long enough to plan around obstacles in the immediate vicinity, but for large-scale obstacles it is insufficient, especially when the avoidance area of the obstacle is larger than the time horizon and the obstacle has greater speed than the robot is capable of. Expanding the horizon increases the computational time requirement, and in pilot experiments with horizons longer than 60s, RBPC could not reliably find plans. This topic is discussed further in Section 7.3.

7.2 Lessons Learned

In this section we concisely state and recap important observations made in the design and implementation process. Many of these can be found in context scattered throughout this thesis but we find it useful to bring them together in one section.

Lesson 1. Dubins paths are not robust to external disturbances.

When replanning regularly with external disturbances, Dubins curves similar to previous curves computed in the past are not always feasible. Thus, when computing sequential minimal length Dubins curves where the starting pose advances along the previous curve but is disturbed by a

small amount, the length of a subsequent curve can increase substantially, do to the new necessity of looping around. This is very undesirable.

Lesson 2. Dubins paths are not numerically stable.

Following the discovery of the previous lesson, the planning algorithm was allowed to assume it had not been disturbed from its previous plan if the controller deemed the plan achievable. This did not fully correct the undesirable looping behavior. Upon further investigation it was discovered that even when precisely calculating a pose directly on a Dubins curve one is not guaranteed to find the path suffix.

Lesson 3. Deciding where to consider the starting pose for a real-time motion planner is not trivial.

For traditional motion planning, one can assume that the robot is not moving while planning takes place, so the starting pose is completely known. For real-time algorithms this is not the case, and so the future pose must be estimated. In order to ensure the system does not oscillate between two plan families while a controller tries to achieve each one in turn, the estimation must take the most recent plan into account. One way to do this is for the controller to estimate this starting pose upon receipt of a new reference trajectory. Unfortunately, this slows planning time by one controller cycle, but it guarantees the estimated pose will be reasonable with respect to the most recent plan.

Lesson 4. It is often hard to specify even the simplest of objectives.

This lesson has come up in several ways throughout the research for this thesis. Described elsewhere are the difficulty of assigning weights to dynamic obstacles and to other parameters that define behavior. A third, less obvious way is the objective that the robot try to drive the paths straight on and straight through. It seems that this behavior would naturally emerge, but, because path segments can be covered while not directly on the center, RBPC proves that wrong: it consistently finds plans that enter the lines off-center at an angle and slowly correct. In some cases plans which short-cut the path in that manner are provably shorter than traversing head-on.

7.3 Further Research

RBPC is substantially limited by its time horizon. This can be clearly seen in Figure 6-3, where large dynamic obstacles cause it to incur extreme collision penalties. The existing algorithm cannot arbitrarily expand its time horizon, as collision checking is linear time in that dimension. An excellent extension to this work would be to generalize collision checking to an adaptable resolution, whereby the horizon could be scaled by local congestion, similar to Shah et al. (2016). This would enable the algorithm to have extremely far lookahead on the open ocean, but still find fine-grained collision-free trajectories when obstacles are nearby. Another approach could be to run a lower frequency planner to compute long-term plans, and use these plans to inform the faster, local search. This hierarchical approach would allow the robot to still be responsive to new information in the local search space, while making decisions that will keep it safe and efficient in the long term as well.

External disturbances in ocean surveying come in long and short term forms. The long-term forms, wind and current, are fairly consistent. The controller must estimate these to accurately follow motion plans, but the planner itself is completely unaware of these effects. Informing the steering function of such a consistent disturbance could help trajectories stay kinematically feasible in the face of even large disturbances.

Marine robots operating with complete autonomy need to behave in safe ways around civilian vessels, but they also need to behave predictably. The International Regulations for Preventing Collisions at Sea (COLREGs) are an important set of rules which define behavior in potential collision scenarios. Following these rules, and reasoning about how others will follow them or not, are essential extensions of this research. See Shah (2016) and the references therein for possible methods to achieve COLREGs compliance.

The proof of probabilistic completeness in Section 3.4 assumes a heuristic which performs dynamic programming-like learning over a partitioning of the space of poses, but no such heuristic is implemented in this work. Designing heuristics which update in this manner would be an important extension of this research. Learning may not be necessary for completeness if one can guarantee certain properties of the heuristic function initially. Implementing a learning heuristic

or determining conditions for which one is not necessary is left to future work.

It might become important to extend the path coverage problem to allow for partially incomplete path segments. In this paradigm, path segments would have a cost associated with leaving them uncovered. When the robot deduces it is no longer in its best interest to continue attempting to cover paths it is allowed to cease. This represents the application of ocean surveying slightly better, because small amounts of missed path, which equate to “data holidays,” are often not worth doubling back for. Additionally, this would allow the robot to deduce when a problem is unsolvable and terminate after completing as much coverage as possible.

In some cases, multiple vessels with surveying capacity may collaborate to survey an area. Designing automated collaboration between surveying robots is important for efficient coverage in these situations.

RBPC’s structure draws heavily from BIT* (Gammell, Barfoot, and Srinivasa 2020), but it does not take the potential advantage offered by ordering edge processing on a best-first basis. This is a conscious design choice, because for an ASV, where collision checking is only in two dimensions, the additional implementation complexity outweighs the potential efficiency gain. It would be interesting to measure the performance difference between a version which lazily evaluated edges, as in BIT*, and RBPC as written.

RBPC could take substantial advantage from parallelization. By increasing the capability by a constant number of threads it could expand the best few vertices in parallel, without contention over anything other than the open list. In a massively parallelized computation environment, the algorithm could collision check every edge in parallel, making the actual search trivially fast. Either of these would be interesting to try in simulation, but might be impractical to run on the hardware available on a small ASV.

RBPC relies heavily on Dubins curves. As seen in Chapter 3, Dubins curves are not robust in the context of re-planning for a real robot. It would be interesting to apply other techniques to the path coverage problem that use a different model. For example, Likhachev and Ferguson (2009) develop a set of motion primitives for a car with which they construct a state lattice to explore. Such an approach may prove similarly effective in the autonomous ocean surveying setting.

7.4 Conclusions

In this thesis we present a new path coverage problem to model that of autonomous ocean surveying. We propose a novel real-time motion planning algorithm well-suited for solving this problem, and we analyze its theoretical properties. We implement an approximate model-predictive controller to accompany it, and discover intricacies of how their interactions affect emergent behavior. We also implement other architecture to facilitate autonomous ocean surveying in an existing simulation environment. We examine the results of several experiments to analyze the behavior of the system as a whole, both with varying parameters and as compared to a potential field approach. The results indicate that dynamic feasibility and forward-looking planning are important features for a motion planner optimizing a non-trivial objective, but that there is room for improvement with the current design in some realistic scenarios.

Bibliography

Arsenault, Roland (2020a). *ASV Sim*. URL: https://github.com/ccomjhc/asv_sim.

— (2020b). *Autonomous Mission Planner*. URL: https://github.com/ccomjhc/AutonomousMissionPlanner.

— (2020c). *Mission Manager*. URL: https://github.com/ccomjhc/mission_manager.

— (2020d). *Project 11*. URL: https://github.com/ccomjhc/project11.

Arsenault, Roland and Val Schmidt (Feb. 2020). *A Mapping Focused Open-Sourced Software Framework for Autonomous Surface Vehicles*. Canadian Hydrographic Conference. URL: https://hydrography.ca/wp-content/uploads/2020/04/14_Arsenault_2020-02-25_Arsenault_CHC2020.pdf.

Barraquand, J., B. Langlois, and J. . Latombe (1992). "Numerical potential field techniques for robot path planning". In: *IEEE Transactions on Systems, Man, and Cybernetics* 22.2, pp. 224–241. ISSN: 2168-2909. DOI: 10.1109/21.148426.

Bertsekas, Dimitri P. and John Tsitsiklis (1996). *Neuro-Dynamic Programming*. Athena Scientific.

Bircher, Andreas et al. (2018). "Receding horizon path planning for 3D exploration and surface inspection". In: *Autonomous Robots* 42.2, pp. 291–306. ISSN: 1573-7527. DOI: 10.1007/s10514-016-9610-0. URL: https://doi.org/10.1007/s10514-016-9610-0.

Choset, Howie et al. (2005). *Principles of Robot Motion*. The MIT Press.

Choudhury, S. et al. (2016). "Regionally accelerated batch informed trees (RABIT*): A framework to integrate local information into optimal path planning". In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 4207–4214. DOI: 10.1109/ICRA.2016.7487615.

Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs". In: *Numerische Mathematik*.

Dubins, L. E. (1957). "On the curves of minimal length on average curvature, and with prescribed initial and terminal positions and tangents". In: *American Journal of Mathematics* 79.3, pp. 497–516.

Fickert, Maximilian et al. (2020). "Beliefs We Can Believe In: Replacing Assumptions with Data in Real-Time Search". In: *Proceedings of the Thirty-fourth AAAI Conference on Artificial Intelligence (AAAI-20)*.

Frazzoli, Emilio, Munther A. Dahleh, and Eric Feron (2002). "Real-Time Motion Planning for Agile Autonomous Vehicles". In: *Journal of Guidance, Control, and Dynamics* 25.1, pp. 116–129. DOI: 10.2514/2.4856. eprint: https://doi.org/10.2514/2.4856. URL: https://doi.org/10.2514/2.4856.

Fridovich-Keil, D., J. F. Fisac, and C. J. Tomlin (2019). "Safely Probabilistically Complete Real-Time Planning and Exploration in Unknown Environments". In: *2019 International Conference on Robotics and Automation (ICRA)*, pp. 7470–7476. DOI: 10.1109/ICRA.2019.8793905.

Galceran, E. and M. Carreras (2012). "Efficient seabed coverage path planning for ASVs and AUVs". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 88–93. DOI: 10.1109/IROS.2012.6385553.

Galceran, Enric and Marc Carreras (2013). "A survey on coverage path planning for robotics". In: *Robotics and Autonomous Systems* 61.12, pp. 1258 –1276. ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2013.09.004. URL: http://www.sciencedirect.com/science/article/pii/S092188901300167X.

Gammell, J. D., S. S. Srinivasa, and T. D. Barfoot (2015). "Batch Informed Trees (BIT*): Sampling-based optimal planning via the heuristically guided search of implicit random geometric graphs". In: *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3067–3074. DOI: 10.1109/ICRA.2015.7139620.

Gammell, Jonathan D, Timothy D Barfoot, and Siddhartha S Srinivasa (2020). "Batch Informed Trees (BIT*): Informed asymptotically optimal anytime search". In: *The International Journal of Robotics Research* 39.5, pp. 543–567. DOI: 10.1177/0278364919890396. eprint:

https://doi.org/10.1177/0278364919890396. URL: https://doi.org/10.1177/0278364919890396.

Hart, P. E., N. J. Nilsson, and B. Raphael (1968). "A formal basis for the heuristic determination of minimum cost paths". In: *TSSC* 4.2, pp. 100–107.

Hsu, David et al. (2002). "Randomized Kinodynamic Motion Planning with Moving Obstacles". In: *The International Journal of Robotics Research* 21.3, pp. 233–255. DOI: 10.1177/027836402320556421. eprint: https://doi.org/10.1177/027836402320556421. URL: https://doi.org/10.1177/027836402320556421.

Karaman, Sertac et al. (2011). "Anytime Motion Planning using the RRT*". In: *2011 IEEE International Conference on Robotics and Automation (ICRA).*

Kleinbort, Michal et al. (Sept. 12, 2019). "Refined Analysis of Asymptotically-Optimal Kinodynamic Planning in the State-Cost Space". In: arXiv: 1909.05569v3 [cs.RO].

Koenig, S. and M. Likhachev (2005). "Fast replanning for navigation in unknown terrain". In: *IEEE Transactions on Robotics* 21.3, pp. 354–363. ISSN: 1552-3098. DOI: 10.1109/TRO.2004.838026.

Koenig, Sven, Maxim Likhachev, and David Furcy (2004). "Lifelong Planning A". In: *Artificial Intelligence* 155.1, pp. 93 –146. ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2003.12.001. URL: http://www.sciencedirect.com/science/article/pii/S000437020300225X.

Koenig, Sven and Xiaoxun Sun (June 2009). "Comparing real-time and incremental heuristic search for real-time situated agents". In: *Autonomous Agents and Multi-Agent Systems* 18.3, pp. 313–341. ISSN: 1573-7454. URL: https://doi.org/10.1007/s10458-008-9061-x.

Korf, Richard E. (1990). "Real-time heuristic search". In: *Artificial Intelligence* 42.2, pp. 189 –211. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(90)90054-4. URL: http://www.sciencedirect.com/science/article/pii/0004370290900544.

Kuffner, J. J. and S. M. LaValle (2000). "RRT-connect: An efficient approach to single-query path planning". In: *Proceedings 2000 ICRA. Millennium Conference. IEEE International*

Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065). Vol. 2, 995–1001 vol.2. DOI: 10.1109/ROBOT.2000.844730.

LaValle, S. M. and J. J. Kuffner (1999). “Randomized kinodynamic planning”. In: *Proc. IEEE Int. Conf. on Robotics and Automation*, pp. 473–479.

LaValle, Seven M. (2006). *Planning Algorithms*. Cambridge University Press. URL: http://lavalle.pl/planning/bookbig.pdf.

Likhachev, Maxim and Dave Ferguson (2009). “Planning Long Dynamically Feasible Maneuvers for Autonomous Vehicles”. In: *The International Journal of Robotics Research* 28.8, pp. 933–945. DOI: 10.1177/0278364909340445. eprint: https://doi.org/10.1177/0278364909340445. URL: https://doi.org/10.1177/0278364909340445.

Mayer, Larry et al. (2018). “The Nippon FoundationGEBCO Seabed 2030 Project: The Quest to See the Worlds Oceans Completely Mapped by 2030”. In: *Geosciences* 8.2. ISSN: 2076-3263. DOI: 10.3390/geosciences8020063. URL: https://www.mdpi.com/2076-3263/8/2/63.

Otte, Michael and Emilio Frazzoli (2016). “RRTX: Asymptotically optimal single-query sampling-based motion planning with quick replanning”. In: *The International Journal of Robotics Research* 35.7, pp. 797–822. DOI: 10.1177/0278364915594679. eprint: https://doi.org/10.1177/0278364915594679. URL: https://doi.org/10.1177/0278364915594679.

Reed, Sam and Val E. Schmidt (2016). “Providing Nautical Chart Awareness to Autonomous Surface Vessel Operations”. In: *MTS/IEEE OCEANS Monterey*.

Shah, B. C. et al. (2014). “Trajectory planning with adaptive control primitives for autonomous surface vehicles operating in congested civilian traffic”. In: *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2312–2318. DOI: 10.1109/IROS.2014.6942875.

Shah, Brual (2016). “Planning for Autonomous Operation of Unmanned Surface Vehicles”. PhD thesis. URL: https://drum.lib.umd.edu/bitstream/handle/1903/19040/Shah_umd_0117E_17582.pdf?sequence=1&isAllowed=y.

Shah, Brual C. et al. (Oct. 2016). "Resolution-adaptive risk-aware trajectory planning for surface vehicles operating in congested civilian traffic". In: *Autonomous Robots* 40.7, pp. 1139–1163. ISSN: 1573-7527. URL: https://doi.org/10.1007/s10514-015-9529-x.

Song, Rui, Yuanchang Liu, and Richard Bucknall (2017). "A multi-layered fast marching method for unmanned surface vehicle path planning in a time-variant maritime environment". In: *Ocean Engineering* 129, pp. 301 –317. ISSN: 0029-8018. DOI: https://doi.org/10.1016/j.oceaneng.2016.11.009. URL: http://www.sciencedirect.com/science/article/pii/S0029801816305182.

Stentz, Anthony (1995). "The Focussed D* Algorithm for Real-Time Replanning". In: *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*. IJCAI95. Montreal, Quebec, Canada: Morgan Kaufmann Publishers Inc., 16521659. ISBN: 1558603638.

Strub, Marlin P. and Jonathan D. Gammell (Feb. 16, 2020a). "Adaptively Informed Trees (AIT*): Fast Asymptotically Optimal Path Planning through Adaptive Heuristics". In: arXiv: 2002.06599v2 [cs.RO].

— (Feb. 16, 2020b). "Advanced BIT* (ABIT*): Sampling-Based Planning with Advanced Graph-Search Techniques". In: arXiv: 2002.06589v1 [cs.RO].

Techy, Laszlo and Craig A. Woolsey (2009). "Minimum-Time Path Planning for Unmanned Aerial Vehicles in Steady Uniform Winds". In: *Journal of Guidance, Control, and Dynamics* 32.6, pp. 1736–1746. DOI: 10.2514/1.44580. eprint: https://doi.org/10.2514/1.44580. URL: https://doi.org/10.2514/1.44580.

Appendices

Appendix A

Discussion of Parameters

A.1 Planner Parameters

Planner frequency

The motion planning algorithm RBPC runs at a fixed frequency of 1 Hz. This frequency is high enough that new information about dynamic obstacles will inform planning early enough to take action to avoid them, but long enough that RBPC reliably finds plans at the 30s time horizon. The real-time guarantee allows us to change this frequency if desired. We expect that lowering this frequency would allow RBPC to find plans at longer time horizons, but increase the difficulty of accurate pose estimation to start planning: estimating a pose for a longer planning time may have larger error.

Turning radii

The non-coverage turning radius should reflect what the ASV is capable of achieving given reasonable external disturbances. The smaller this is, the tighter turns in planned trajectories will be. Based on the simulated model used in experiments, 8m is a good value for this parameter.

The coverage turning radius reflects the maximal turning rate for which coverage is allowed. It should reflect the capabilities of sonar equipment and a tradeoff between data quality and efficiency. If set equal to the non-coverage turning radius it will be ignored. 100m is used in the experiments.

Speeds

The max speed parameter reflects how fast the ASV is capable of going given reasonable external disturbances. It is assumed that this speed is acceptable for coverage. 2m/s is used in the experiments.

The slow speed is an alternative speed, intended for avoiding collisions. Setting it too low makes it difficult for the controller to follow plans while fighting moderate currents, but setting it high decreases the effectiveness for collision avoidance. If it is zero or equal to max speed this parameter is ignored. It is not feasible for the controller to maintain a constant pose for any length of time, which is why zero speed is not allowed. 1m/s is used in the experiments.

Line width

Line width determines the across-track error allowed during coverage. This potentially reflects a tradeoff between data quality and efficiency, as wider lines are easier to maintain, but if set too high the planner will discover all sorts of shortcuts which tend to be undesirable. This parameter also determines the minimum path segment length, which is equal to twice this value. 2m is used in the experiments.

Branching factor

The branching factor parameter is the number of nearby poses to which RBPC connects during expansion. Increasing this parameter increases the difficulty of search, but may provide better solutions quicker. The true branching factor is actually increased by a factor of up to four: one for each combination of the two speeds and two turning radii. A value of 9 is used in the experiments.

Time horizon

The time horizon determines how far in time motion planning goes. Increasing this increases the amount of work needed each iteration of search, but allows the system to react sooner to large obstacles, or plan its way out of larger heuristic local minima. 30s is a reasonable value for this parameter because it allows for enough lookahead to avoid small, local dynamic obstacles. Experimentation suggests that in uncluttered environments, higher values are possible. 60s seems to work reasonably well in the scenarios evaluated here. Higher values such as 90s or 120s sometimes work, but other times prevent RBPC from finding even an initial solution. In very cluttered environments, such as the maze (Figure 6-4), RBPC occasionally fails to find a plan with a 30s

horizon, but not often enough to prevent the system from completing the scenario. Unless otherwise specified 30s is used in the experiments.

Minimum trajectory duration

The minimum trajectory duration puts a lower bound on plan length to ensure safety for the near future and sufficient length of reference trajectory for the controller, but is almost completely ignored by RBPC. Since planning normally extends to the time horizon, which is always larger than or equal to this parameter, by default the restriction it imposes is satisfied. When a plan completes coverage, RBPC does not require that it extends to the time horizon, but only that it extends this minimum duration past the end of coverage. 5s is used for this parameter in the experiments.

Collision checking increment

The collision checking increment specifies the distance increment at which collision and coverage checking is performed. This parameter has an effect on the difficulty of search: decreasing it increases computation time per unit trajectory length. Due to the details of the implementation, increasing it can have undesirable effects. For very similar trajectories where the increment lines up differently, collision penalties and coverage can be substantially different. This effect is mitigated by forcing collision checking onto a discretization in time starting from a fixed point, but extending the algorithm to make at least coverage checking agnostic to this parameter is preferred. 0.05m is used in the experiments.

Initial samples

The initial samples parameter specifies the number of sampled poses generated before the first round search in RBPC. This has an impact on the quality of solution found in the first search, as well as the time required. Since the first search is usually the longest, with no incumbent solution to bound the motion tree, it is good to keep this value low, but not so low that an initial solution cannot be found most of the time. 100 is used, but in pilot experiments similar performance was often achieved with values of 32 or 1000.

Heuristic

The heuristic parameter determines which heuristic function guides search. Most of the scenarios examined in the experiments use a single path segment only, in which case all of the heuristics (described in Section 3.2) are very similar. While we see better performance from the K-TSP and Dubins TSP heuristics in Figure 6-12, Euclidean TSP is the best admissible heuristic, so it is used for the rest of the experiments.

Dynamic obstacles representation

The dynamic obstacles parameter determines which of the two representations is used. By default the Rectangle representation is used, because it more reliably produces desirable behavior from the system. Additional representations are an important matter of future work.

A.2 Controller Parameters

Controller frequency

The controller is run at a fixed frequency of 10 Hz. This is high enough to react to short-term disturbances such as actuation noise and waves, but long enough that the controller can reliably simulate 3-4 timesteps deep to determine controls each cycle. Decreasing this frequency would limit the controller's ability to react in the short term, but increase its lookahead. With a dynamically feasible motion planning algorithm such as RBPC, additional lookahead is not likely to be very beneficial, as the motion plans should not contrain extreme maneuvers that need to be anticipated far in advance.

Simulation timestep

The simulation timestep, *timestep* in Alg. 5, determines how long each control is simulated in the depth first search of trajectories. It thus has an effect on the length of lookahead the controller can achieve: longer simulation times mean further lookahead but less branching. 1s is used for this parameter.

Control granularity

There are two control granularity parameters: rudder and throttle granularity. Each determines the respective distance in control space for neighboring controls, as described in Chapter 4. A rudder granularity of 0.0625 and a throttle granularity of 0.125 are used. Adjusting these numbers has interesting effects on the controller's behavior. For example, with a higher rudder granularity (lower parameter value), turning away from straight has a very slight effect, and is often ignored as an option in favor of slowly drifting off-course and eventually correcting with a harder turn. Setting a low granularity (high parameter value) may make optimal values for going straight impossible to find, causing the controller to oscillate between values on either side, wiggling. The interplay between these values and the scoring function is not well understood.

Scoring weights

The scoring function has three weights: distance, heading, and speed (w_d, w_h, and w_s in Equation 4.1). These can be seen as conversion factors from the native units (distance in meters, heading in radians, speed in m/s) to score units. These weights determine what is prioritized when scoring trajectories. 1 is used for a distance weight, 20 for a heading weight, and 5 for a speed weight in the experiments. In pilot experiments lower heading weights allow the robot to oscillate around a straight line, whereas heading weight of 20 generally smooth its passage. The impact of different speed weights is still unclear. Good values for these parameters might depend on environmental conditions, and recommend more intensive further study, ideally with data from the physical robot.

Achievable threshold

The achievable threshold is in score units, mentioned above, and controls the controller's level of confidence. 15 is used for this parameter in the experiments. Higher values tend to work well often, allowing the planner to be almost completely separated from reality, as the controller is quite good at following its trajectories. Lower values make it harder for the planner to cover path segments, as its starting pose is more likely to be interrupted by real pose information, and the Dubins steering function is not robust to these disturbances.

A.3 Simulation Parameters

Current

The simulator exposes current speed and direction parameters. A current of 0.5m/s due East is used in the experiments. In pilot experiments the controller is robust to different current directions. Increasing the current speed makes it more difficult for the controller to achieve the speeds required by the planner, so when this parameter is changed, the planner's max speed parameter should also be changed to reflect a speed always dynamically feasible.

Noise

The controller exposes actuation noise and current noise parameters. In pilot experiments the controller was able to perform acceptably in extremely noisy environments. These parameters reflect variance in a Gaussian distribution with mean zero for the actuation noise parameters, and the current speed and direction for the applicable noise parameters. Thrust variance of 0.1, rudder variance of 0.25, drag variance of 0.1, current speed variance of 0.1, and current direction variance of 0.25 are used in the experiments. Current direction is in degrees.

Appendix B

Descriptions of Scenarios

Videos of many of the scenarios can be found at this link: `https://www.youtube.com/playlist?list=PLNcViETgkSdceqn0dB9kG2_fTC_lEgDra`.

B.1 Specific Behavior Scenarios

In this section we present the first set of scenarios mentioned in Chapter 6. These scenarios test for specific, different behavior, whereas the scenarios described in section B.2 are all similar.

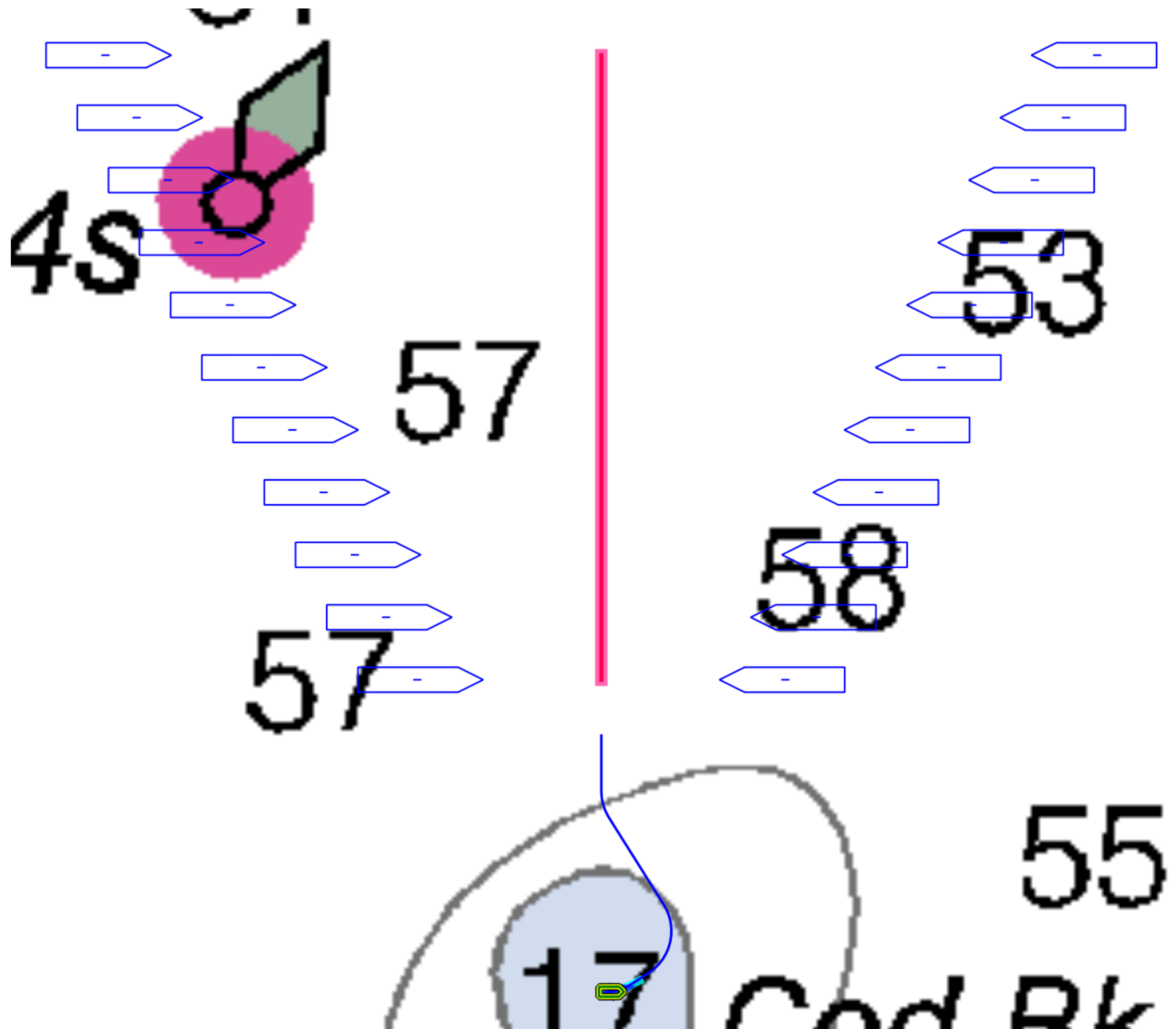


Figure B-1: 100mNorthGauntlet: ASV is positioned below a survey line with many dynamic obstacles approaching from both sides of the line. It must slow to avoid collision. This scenario tests slowing down to avoid obstacles, avoiding obstacles in general, and how several obstacles affect behavior (more obstacles implies more computational effort).

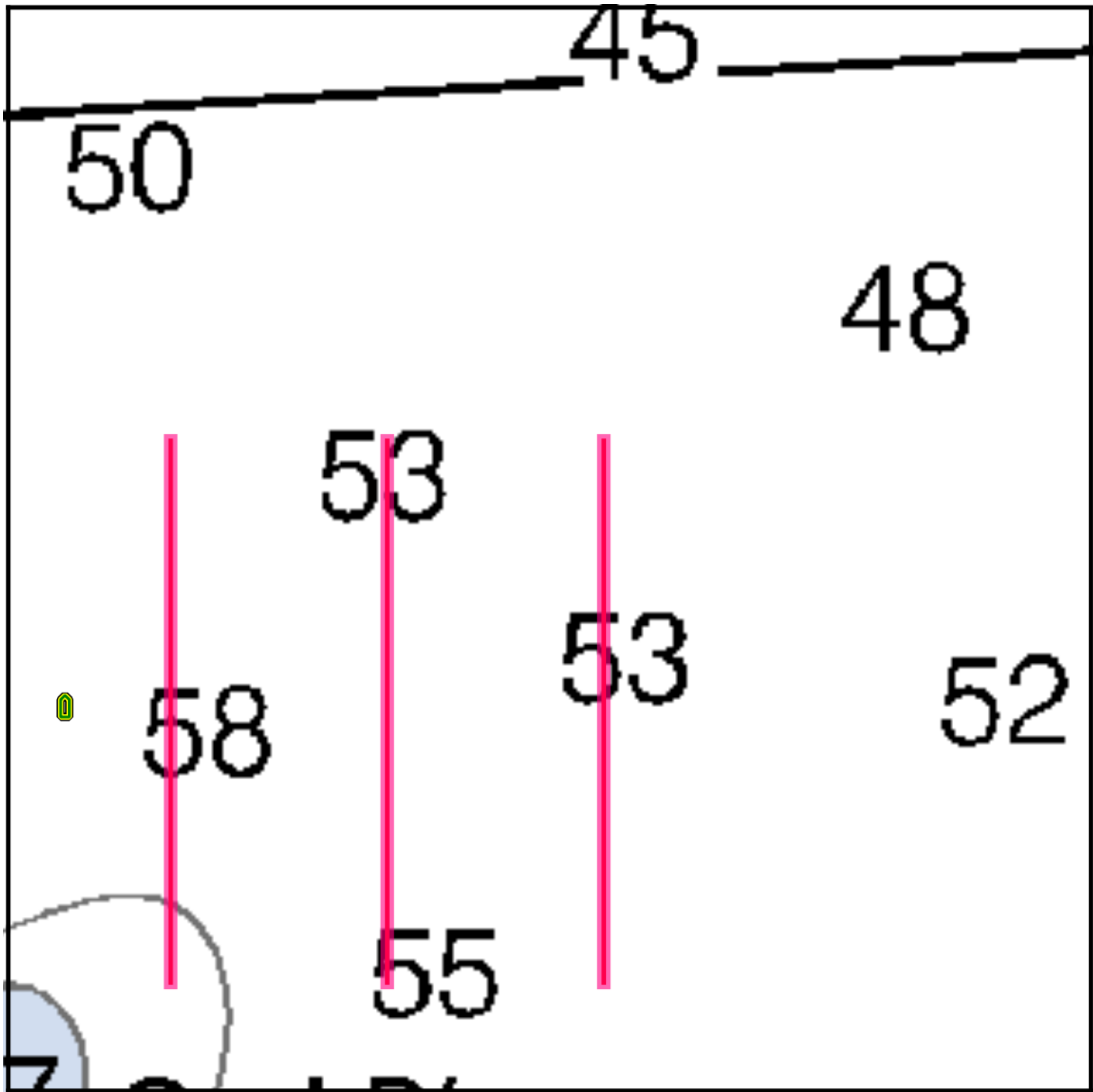


Figure B-2: adjacent_multi_line: ASV is positioned to the left of three parallel survey lines. This scenario tests handling of multiple lines.

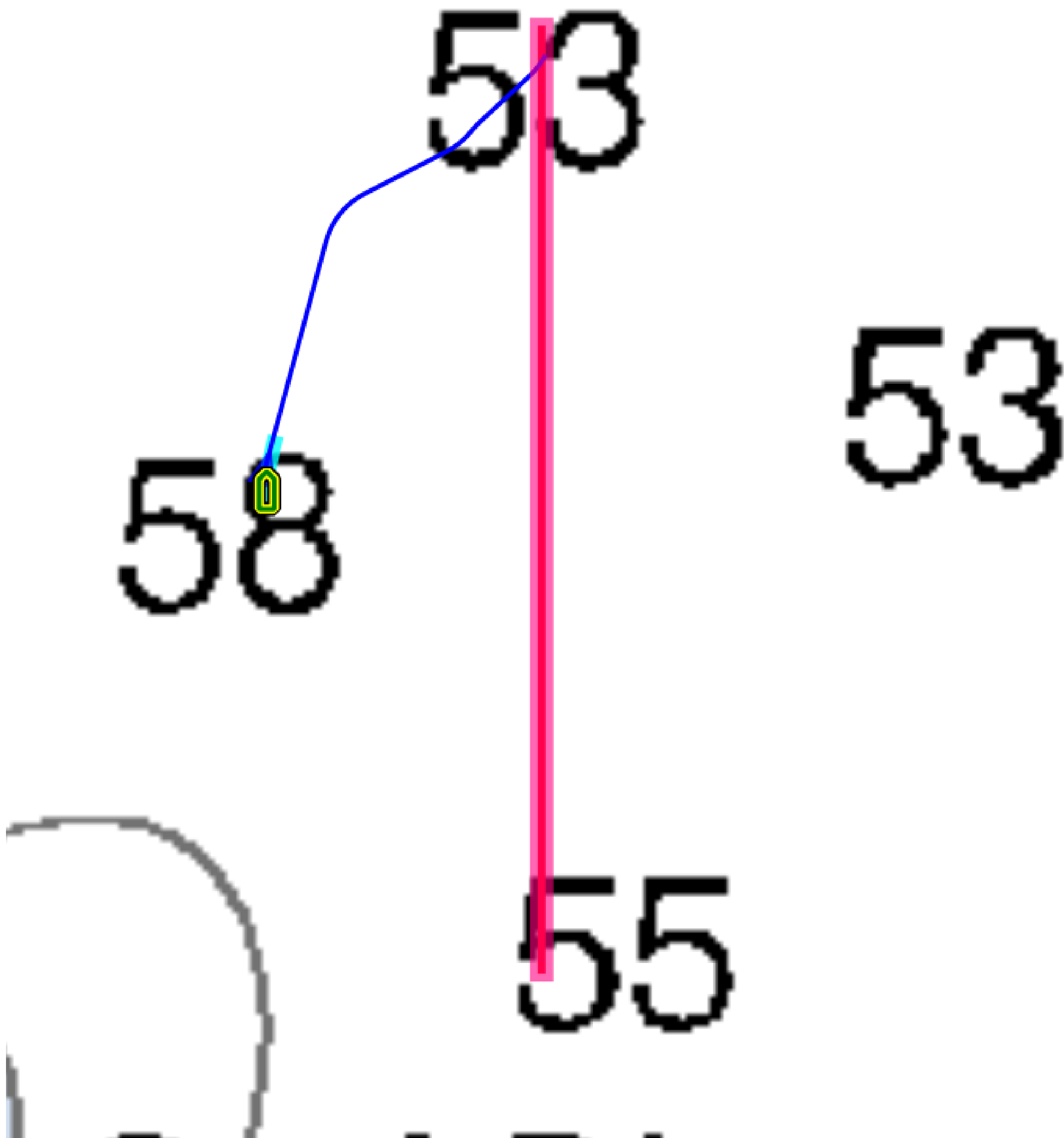


Figure B-3: adjacent_single_line: ASV is positioned to the left of a single survey line, partway through it. This scenario tests how the planner decides to begin to cover the line.

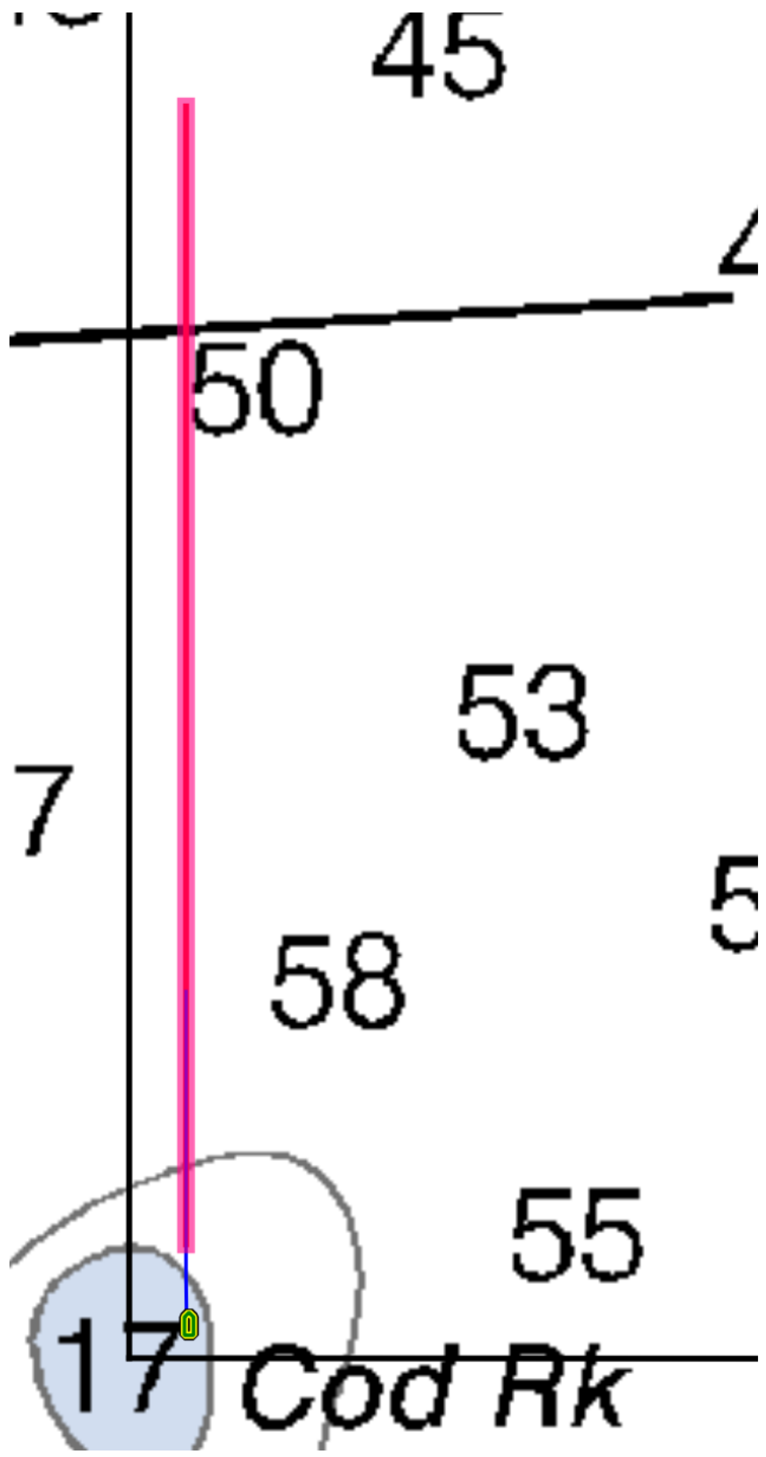


Figure B-4: ahead_long: ASV is positioned below a survey line. This scenario tests staying on a line that was easy to start.

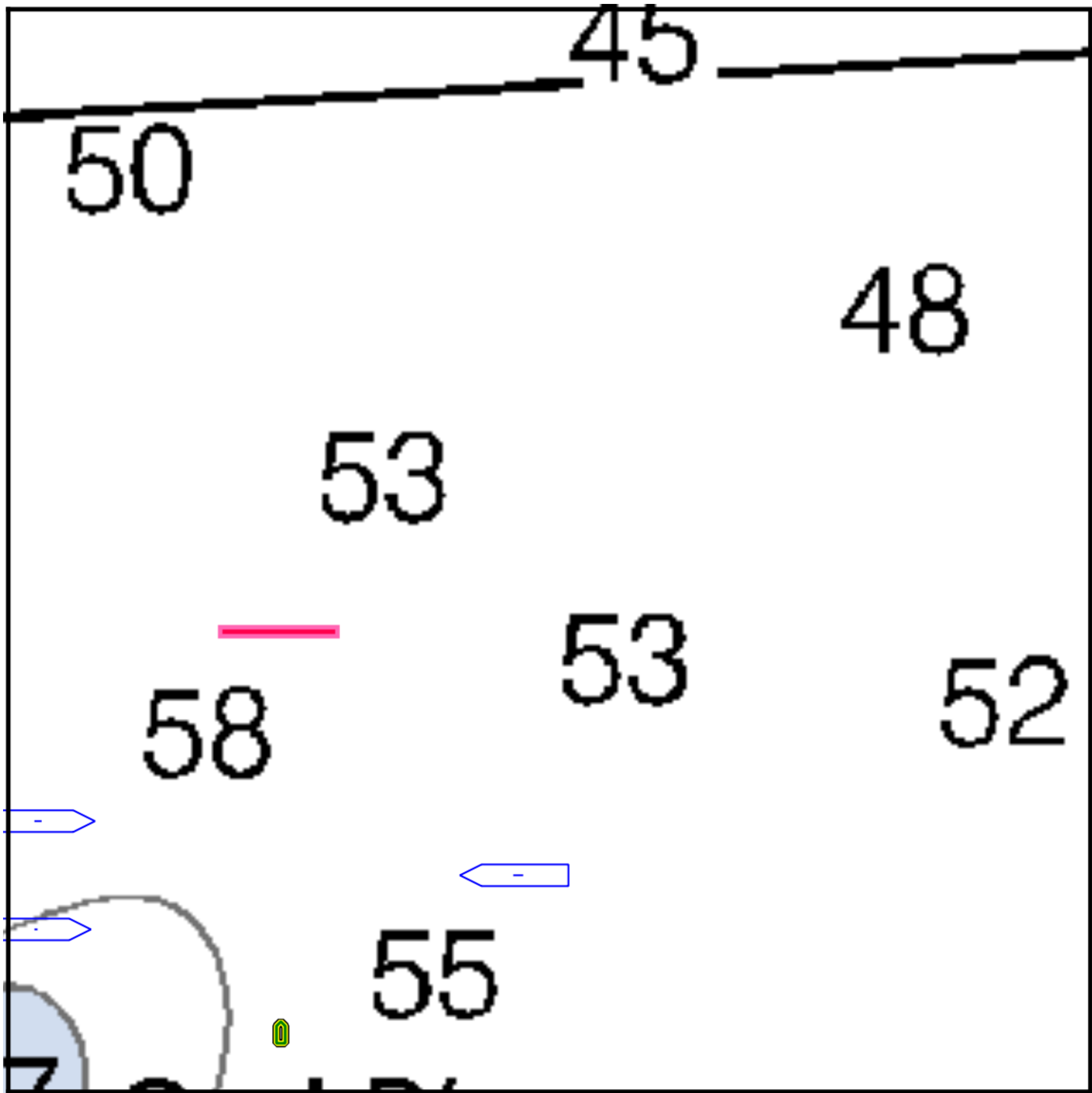


Figure B-5: cannon_crossfire: ASV starts below a horizontal survey line with three dynamic obstacles of differing speeds crossing its path. This scenario tests handling of multiple obstacles cutting across the ASV's path, and slowing to avoid obstacles.

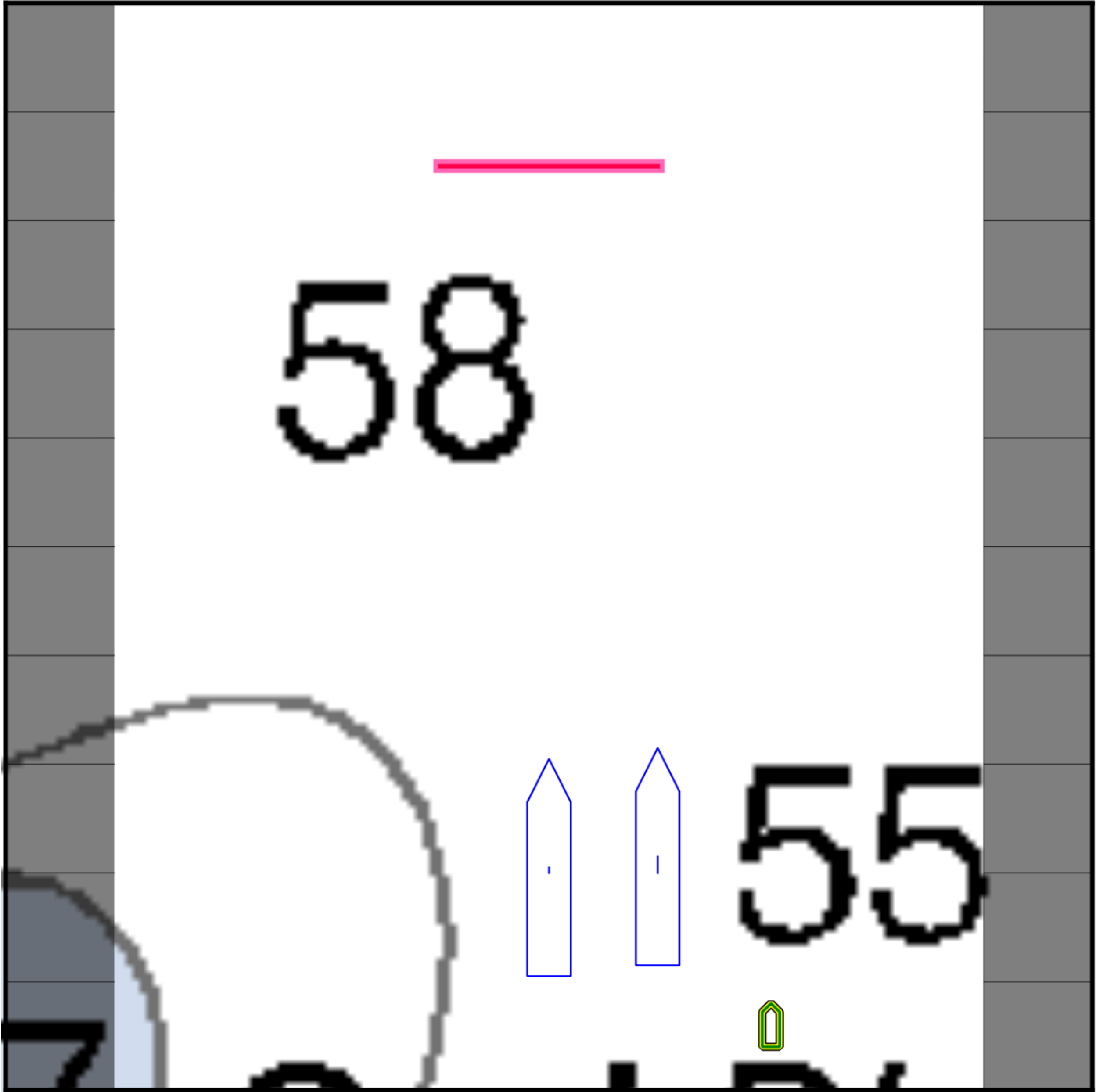


Figure B-6: cannon_cut_across: ASV starts below a horizontal survey line with two dynamic obstacles headed across the line as well. This scenario tests handling multiple obstacles interrupting a survey line, and slowing to avoid obstacles.

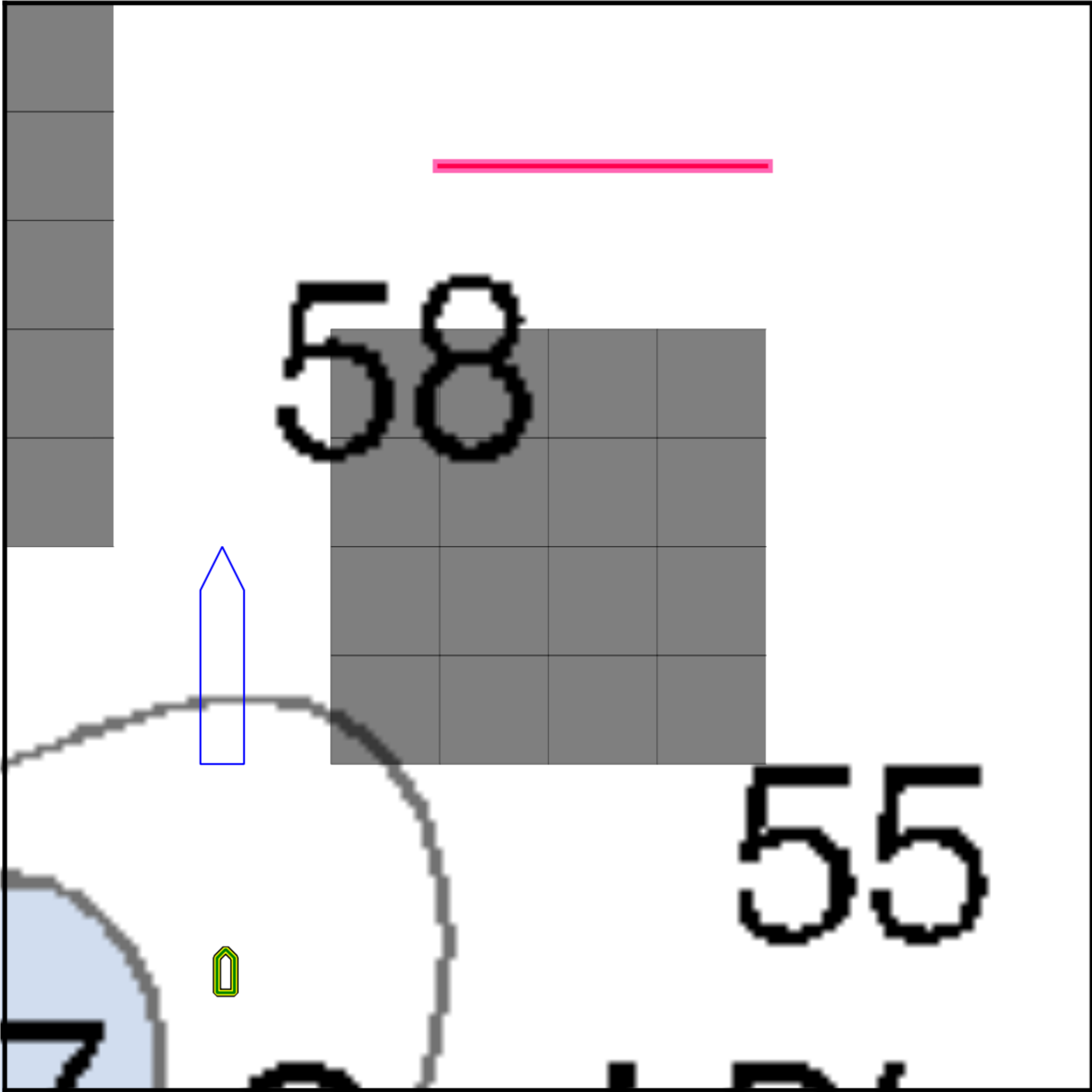


Figure B-7: cannon_follow: ASV starts below a dynamic obstacle with a horizontal survey line also above. The static map blocks a large portion of the middle of the space. This scenario tests following a dynamic obstacle at a safe distance.

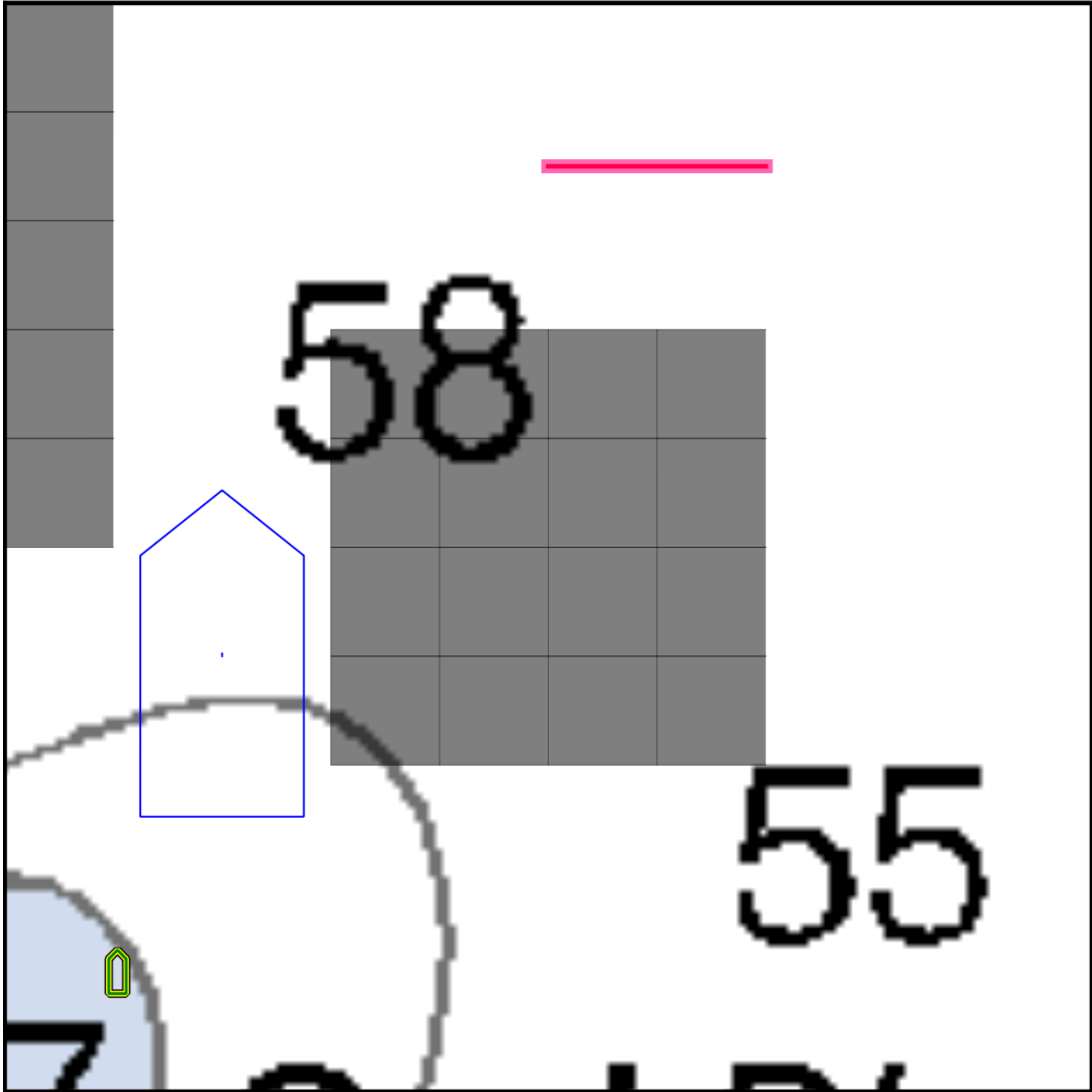


Figure B-8: cannon_go_around: ASV starts below a large, slow dynamic obstacle with a horizontal survey line also above. The static map blocks a large portion of the midle of the space. This scenario tests the ability to discover a faster route going around the island in the middle of the map to reach the survey line, as opposed to following the dynamic obstacle.

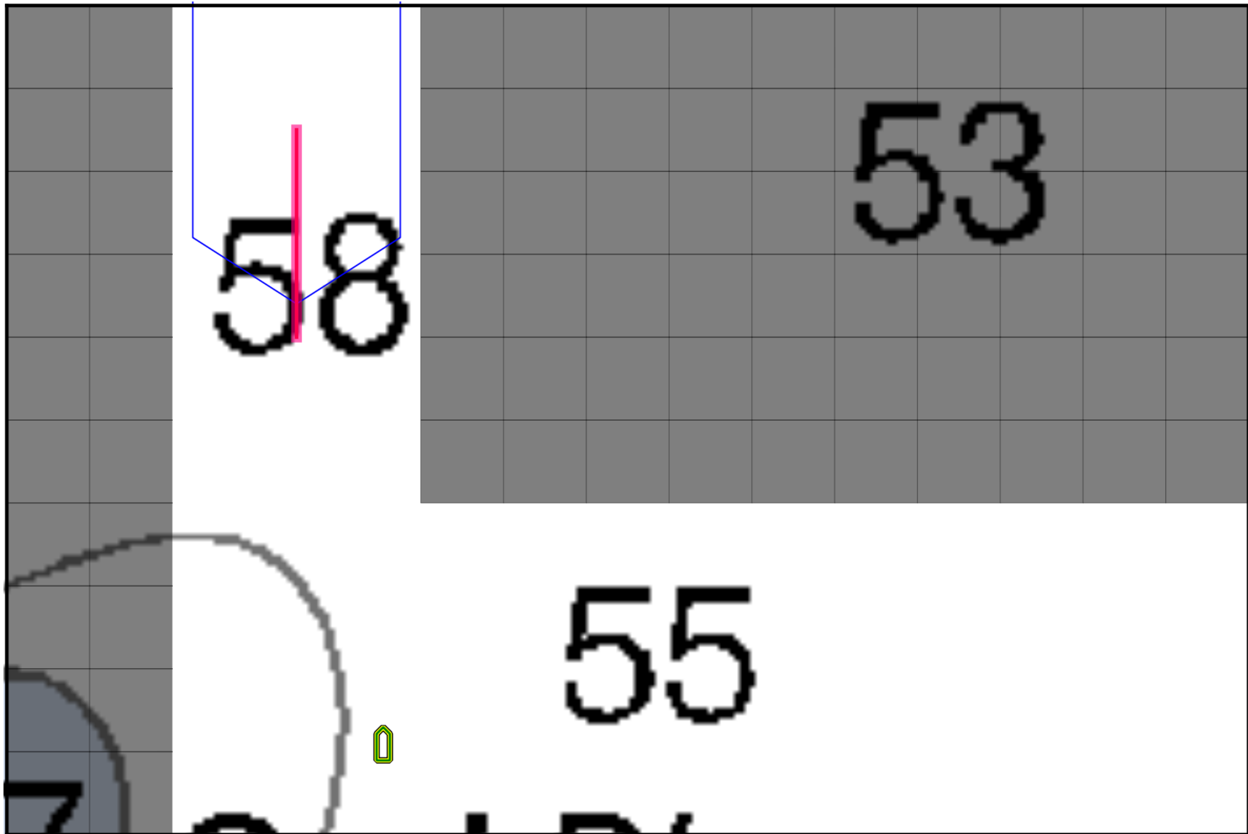


Figure B-9: cannon_wait_1: ASV is positioned below a survey line blocked by a large dynamic obstacle, moving slowly towards the ASV. The static map allows two corridors: one upwards from the ASV, containing both the survey line and the dynamic obstacle, and a second to the right of the ASV. This scenario tests the ability to wait in the open corridor for the dynamic obstacle to safely pass by.

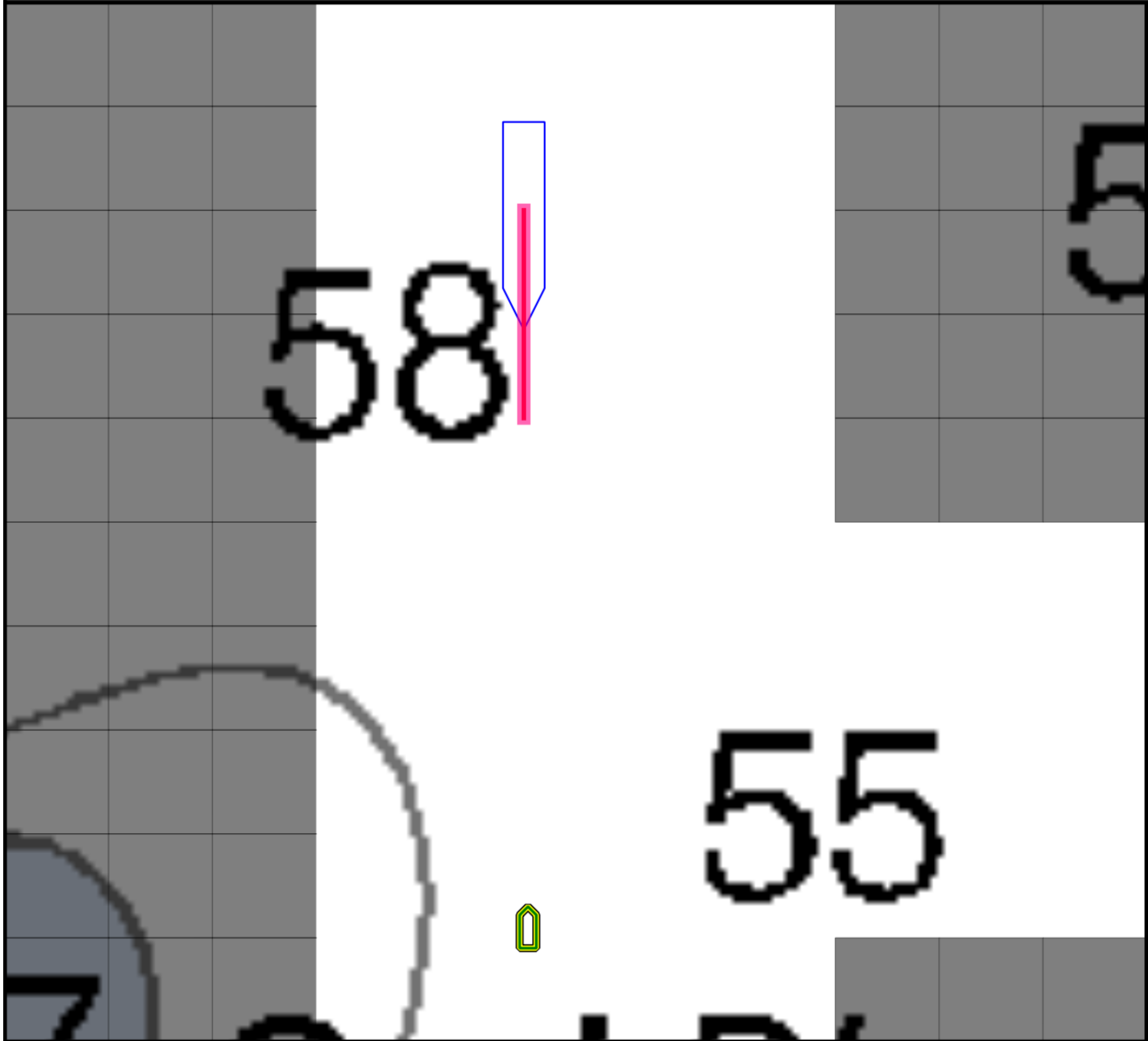


Figure B-10: cannon_wait_2: ASV is positioned below a survey line blocked by a small dynamic obstacle, moving towards the ASV. The static map allows two large corridors, one above the ASV, containing the survey line and the dynamic obstacle, and a second short one to the right of the ASV. This scenario tests the ability to avoid the small obstacle on the way to the survey line.

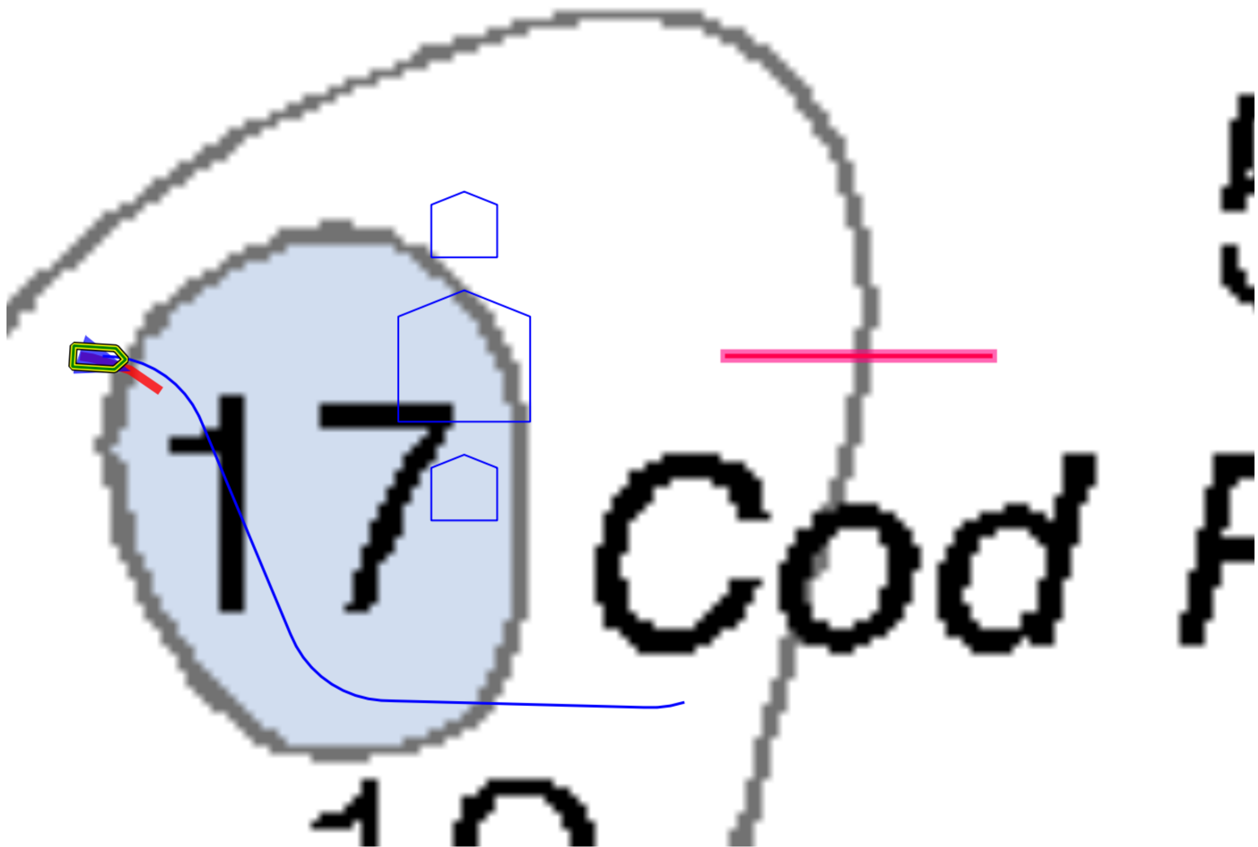


Figure B-11: dynamic_wall_1: ASV is positioned to the left of a horizontal survey line. The path between them is blocked by three unmoving dynamic obstacles. This scenario tests the ability to avoid unmoving dynamic obstacles.

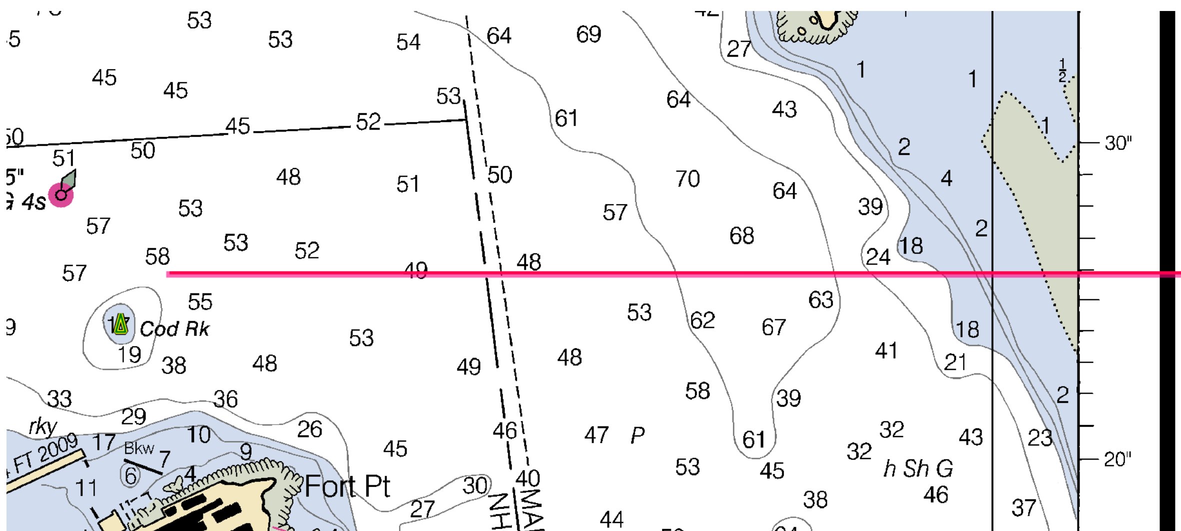


Figure B-12: long_single_line: ASV starts to the left of a long survey line. This scenario tests the ability to follow a line over a longer period of time.

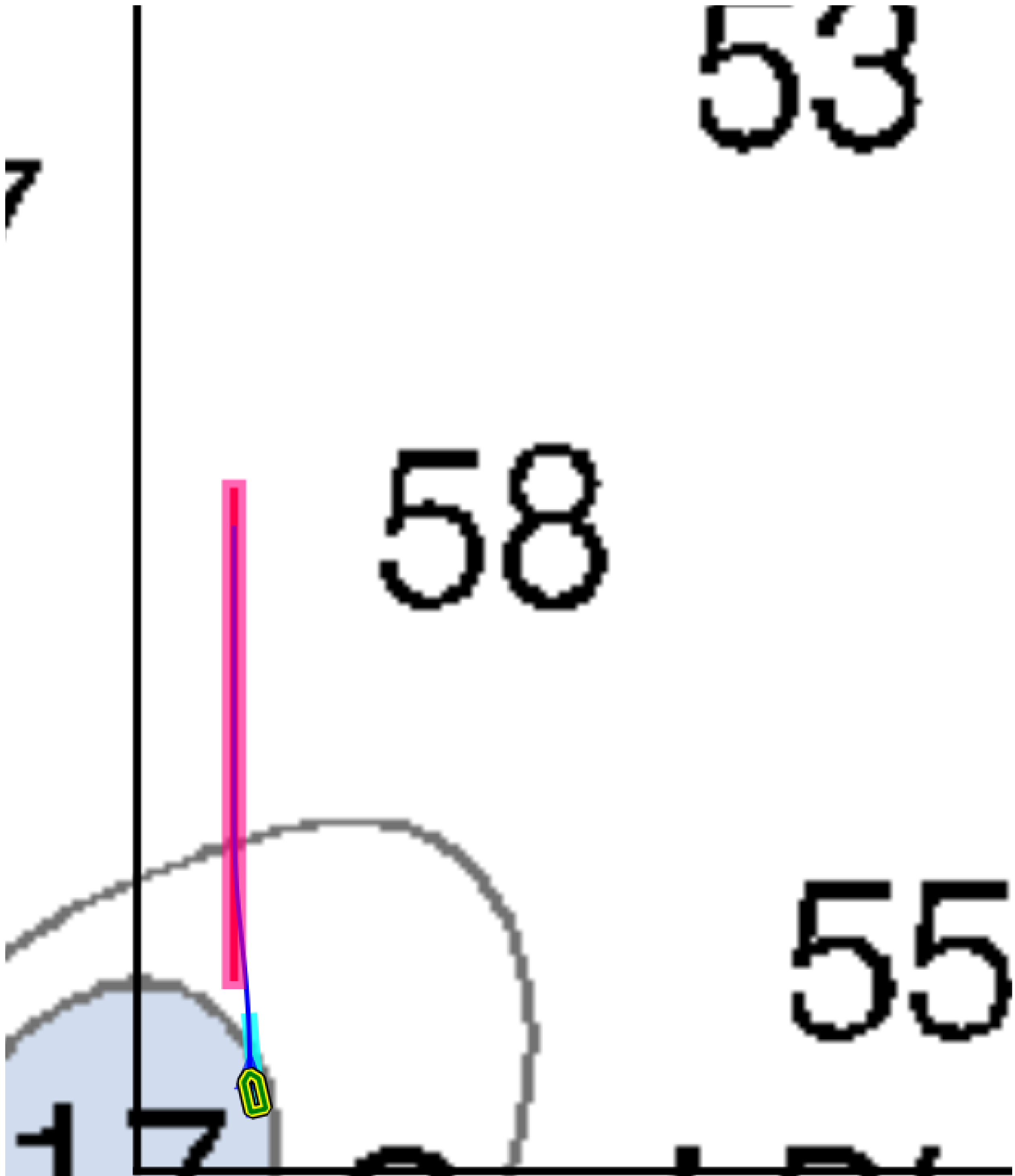


Figure B-13: Test01: ASV is positioned below a short survey line. This scenario tests basic short-term line following.

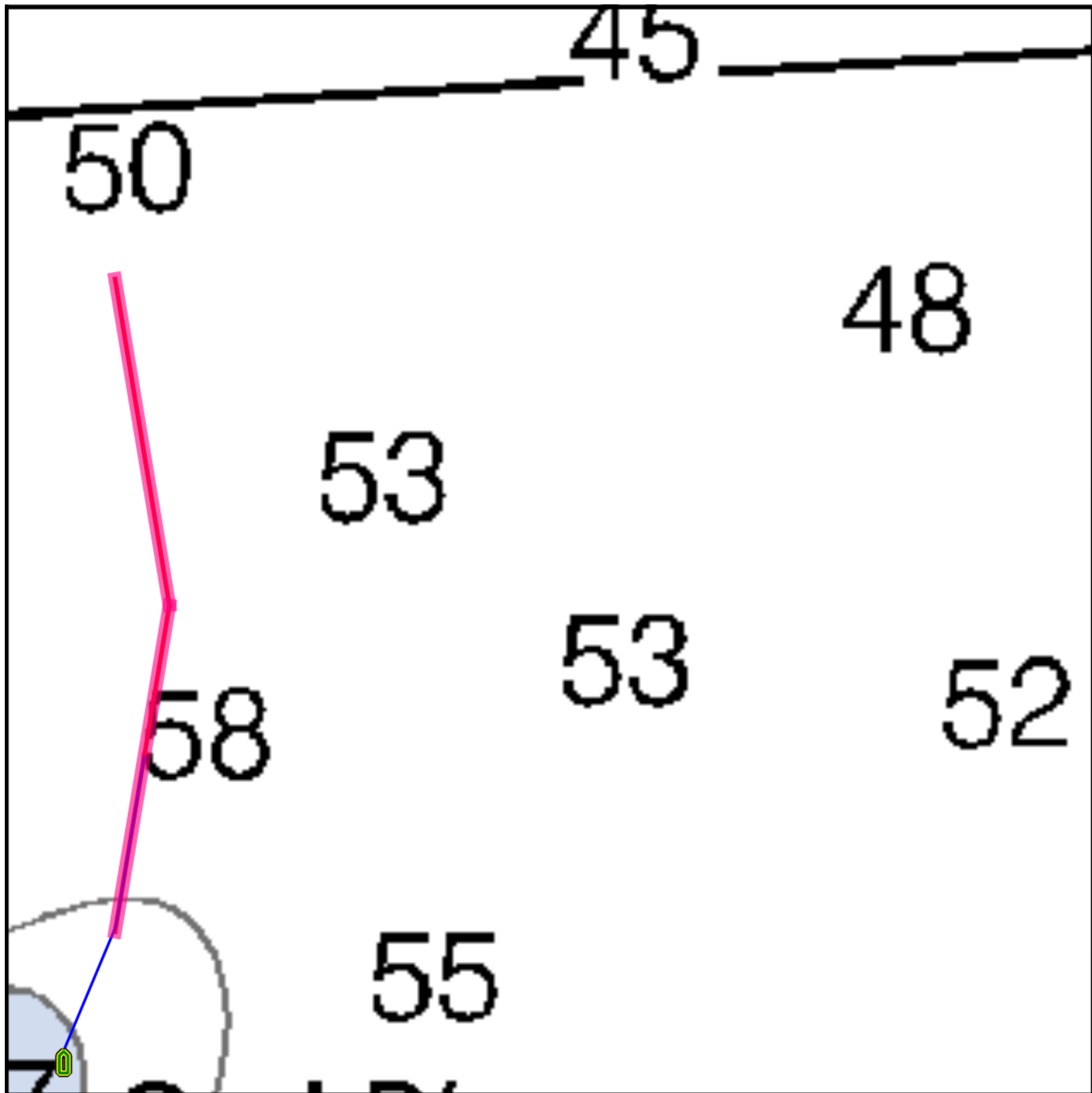


Figure B-14: Test02a: ASV starts below two connected survey lines with a wide angle between them. This scenario tests the ability to plan around a slight corner in a survey.

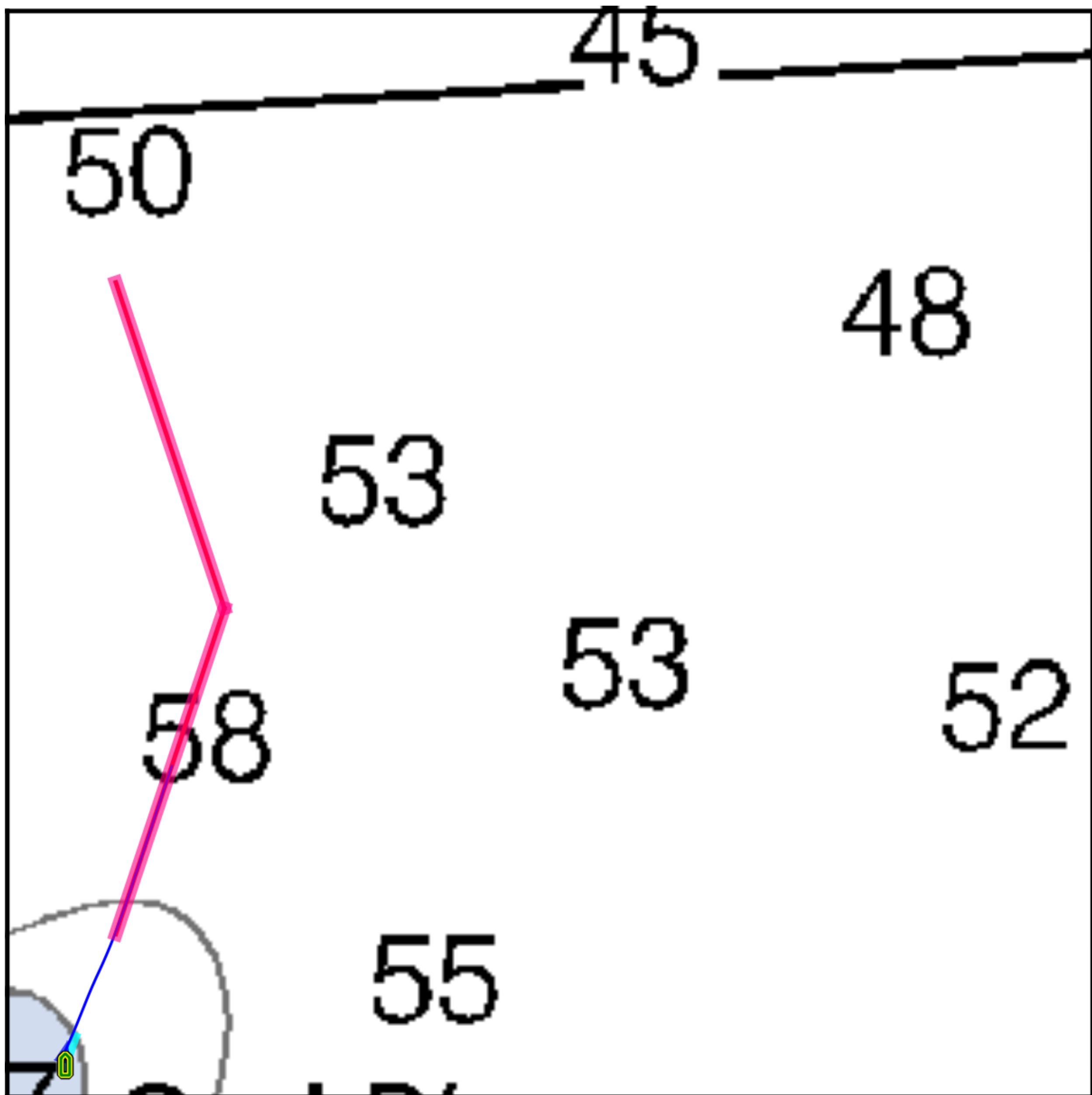


Figure B-15: Test02b: ASV starts below two connected survey lines with a narrower angle between them than in Test02a (Figure B-14). This scenario tests the ability to plan around a moderate corner in a survey.

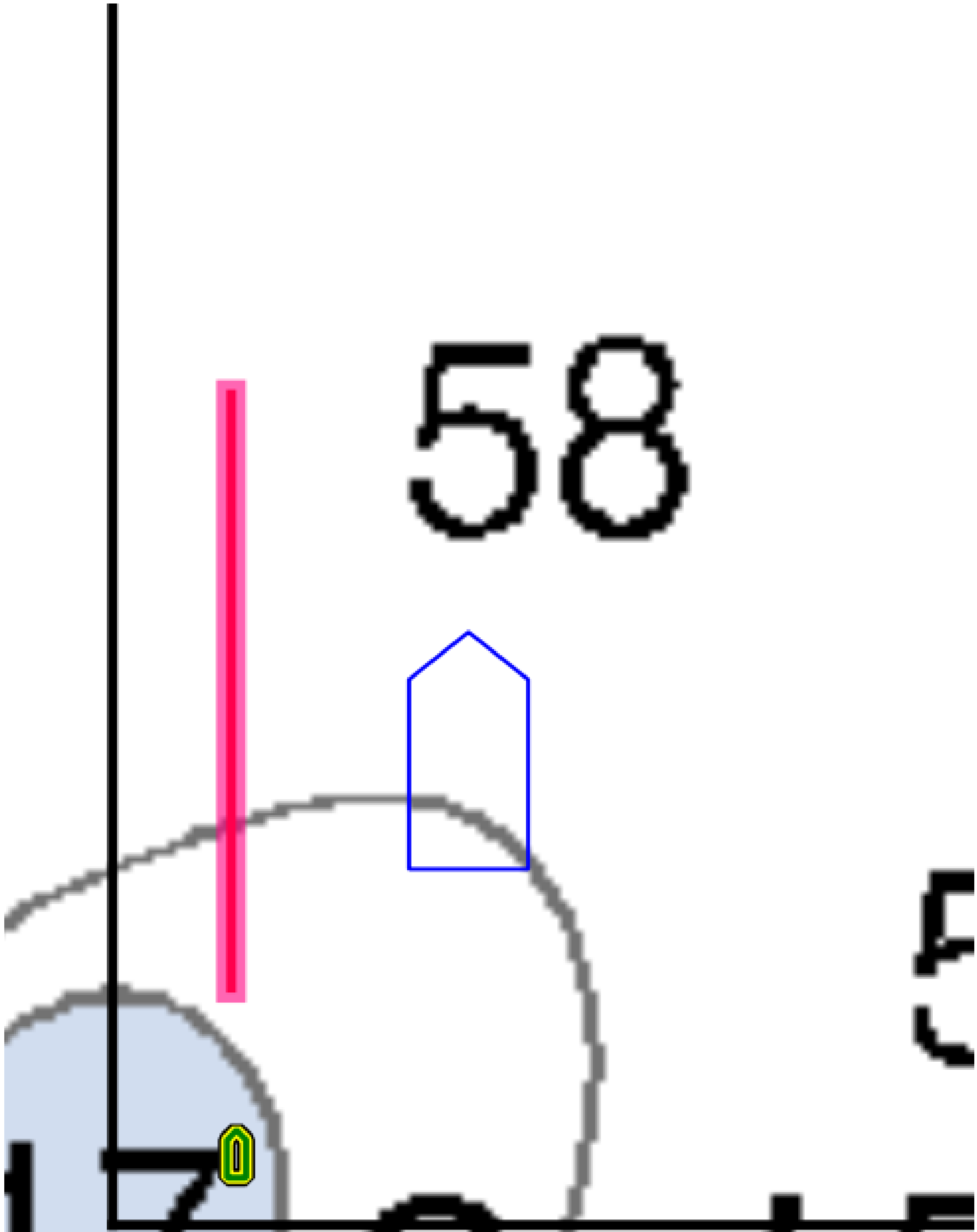


Figure B-16: Test03: ASV starts below a short survey line with a single stationary dynamic obstacle to the right of the line. This scenario tests the ability to ignore the obstacle, which is a safe distance away for its size and direction, and cover the line.

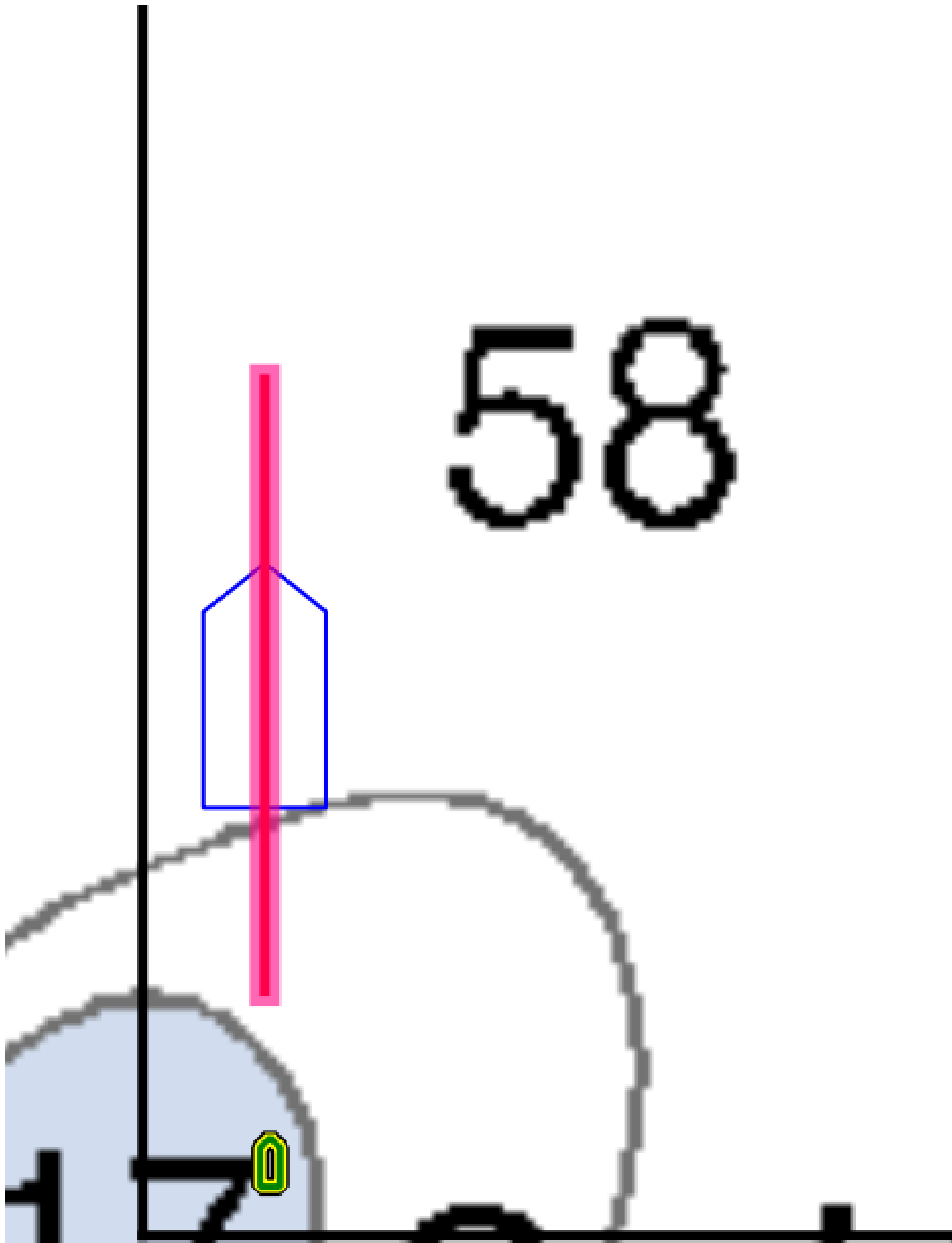


Figure B-17: Test04: ASV is positioned below a short survey line with a stationary dynamic obstacle in the middle of the line. This scenario tests the ability to trade off the mission goals for safety, as the survey cannot be completed safely.

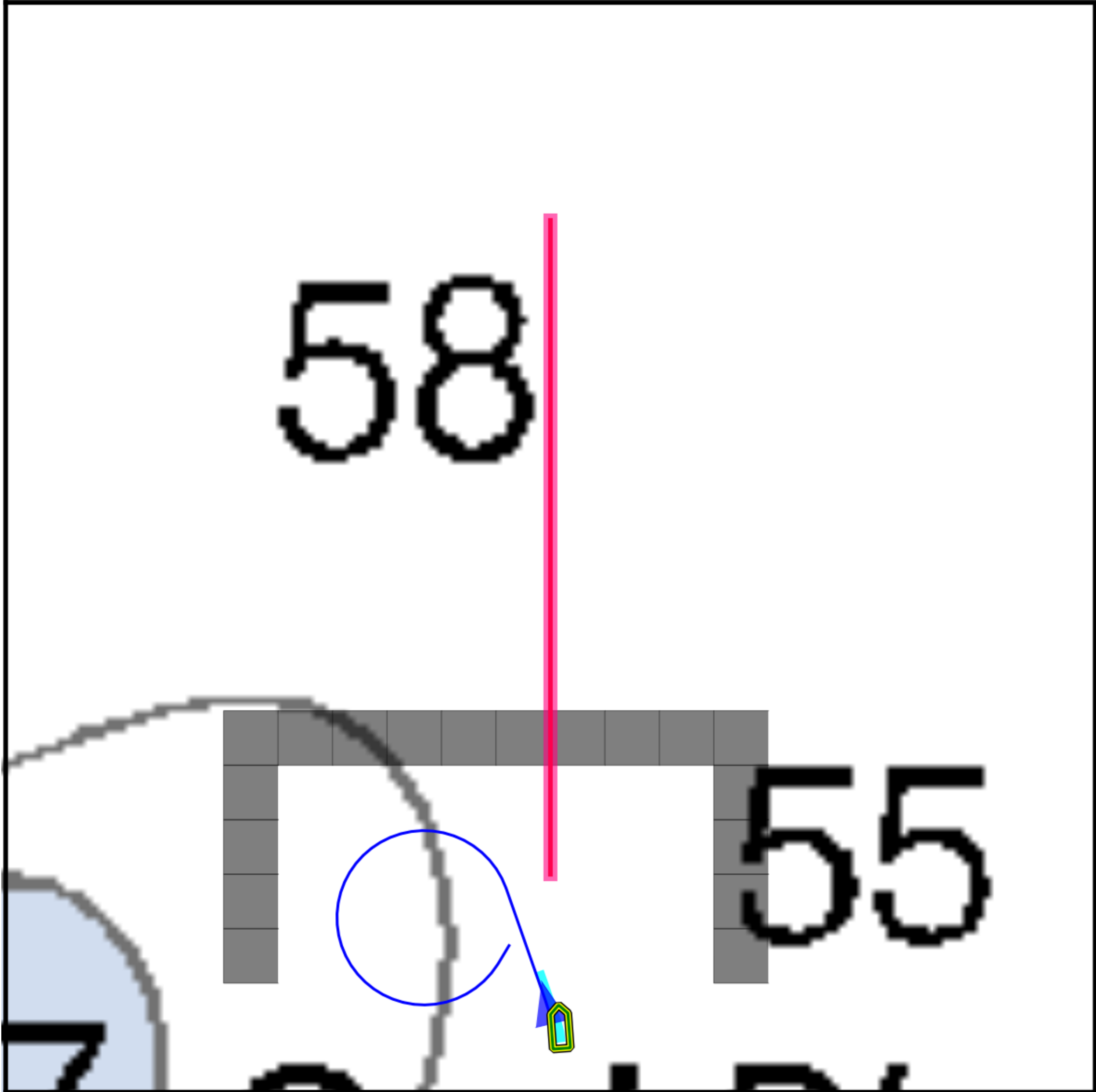


Figure B-18: Test12a: ASV is positioned below a survey line passing through a static obstacle. This scenario tests the ability to navigate around the static obstacle to complete as much of the survey as possible.

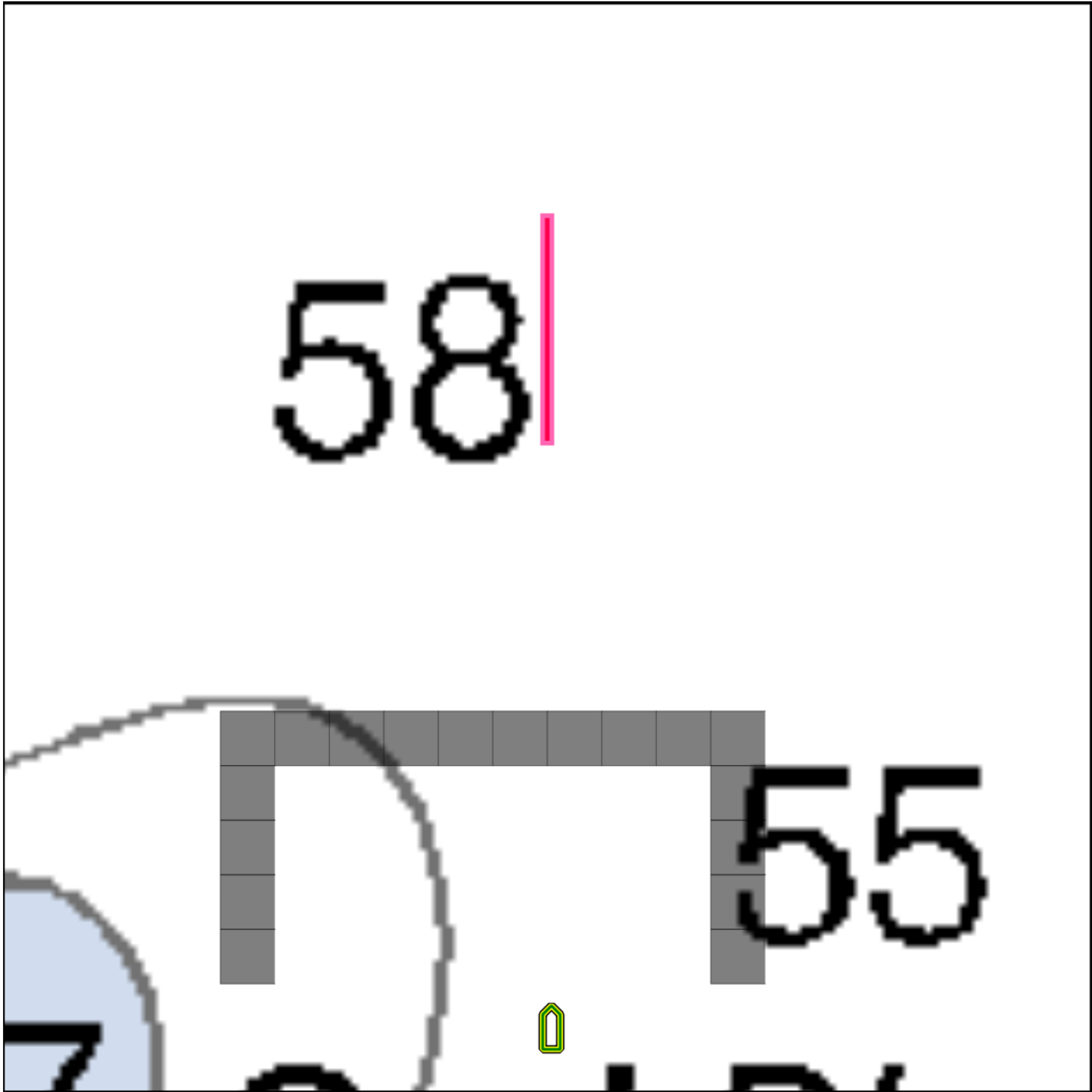


Figure B-19: Test12b: ASV is positioned below a static obstacle forming a local minimum in all implemented heuristics. This scenario tests the ability to navigate outside of such a minimum, and is possible using a reasonable time horizon.

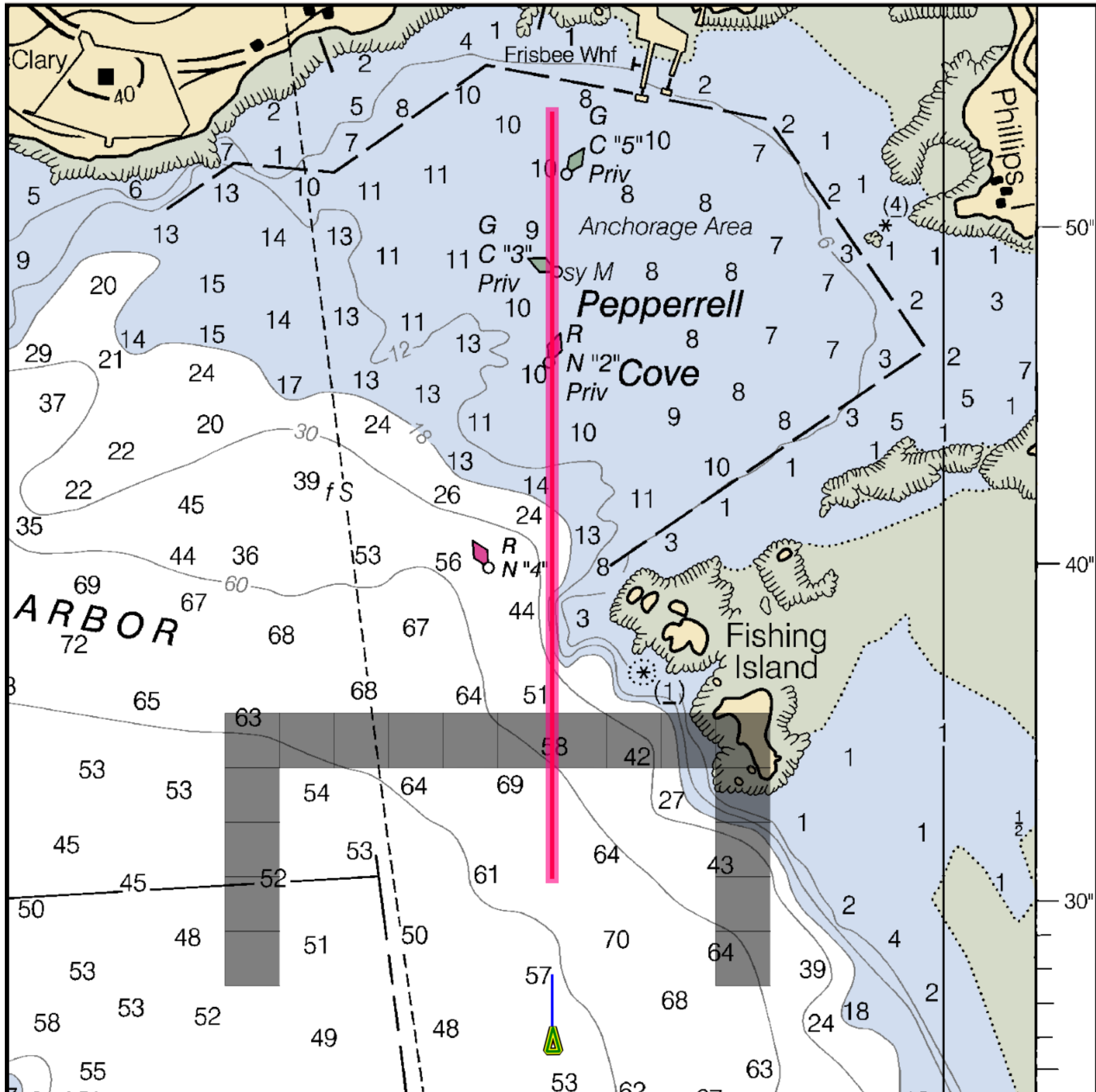


Figure B-20: Test12c: ASV is positioned below a survey line passing through a large static obstacle. This scenario tests the ability to navigate around the static obstacle to complete as much of the survey as possible.

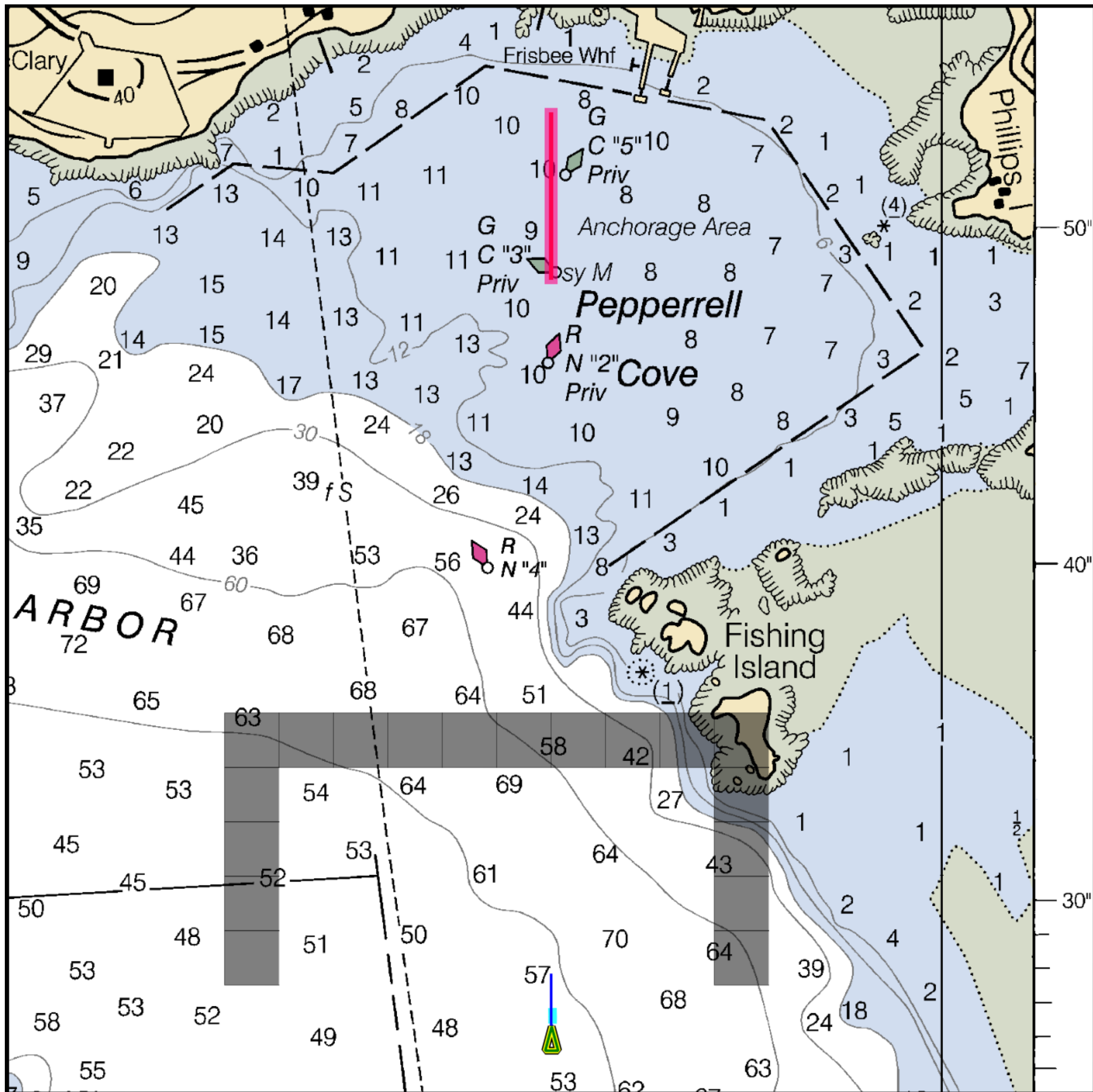


Figure B-21: Test12d: ASV is positioned below a large static obstacle forming a local minimum in all implemented heuristics. This scenario tests the ability to navigate outside of such a minimum, but is too large to be possible using any time horizon feasible for RBPC as presented in this thesis.

Figure B-22: narrow_wide: ASV must pass through a narrow corridor before navigating around a seawall to a short survey line.

B.2 Line Following Scenarios

In this section we present several variations on basic line following in the presence of dynamic obstacles. They are designed to be longer to observe interaction with dynamic obstacles whose clearance requirements are much larger than those in the earlier tests. Each scenario family in this

section equates to four scenarios in the experiments. For example, Test05 here describes scenarios test05a, test05b, test05c, test05d, and test05e.

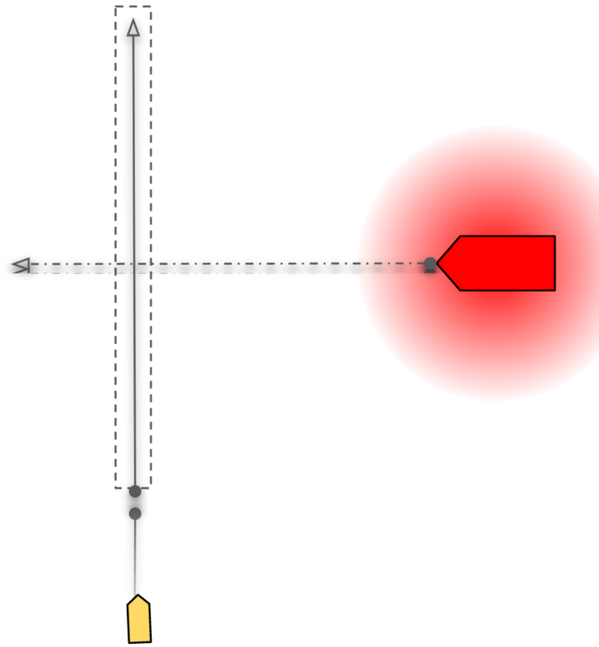


Figure B-23: Test05. ASV must navigate along a survey line with a moving obstacle from the right which requires 50m (a), 100m (b), 300m (c), 500m (d), and 2000m (e) avoidance. A video of RBPC running in the Test05 scenario can be found here: `https://youtu.be/hioDkJkH2vM`

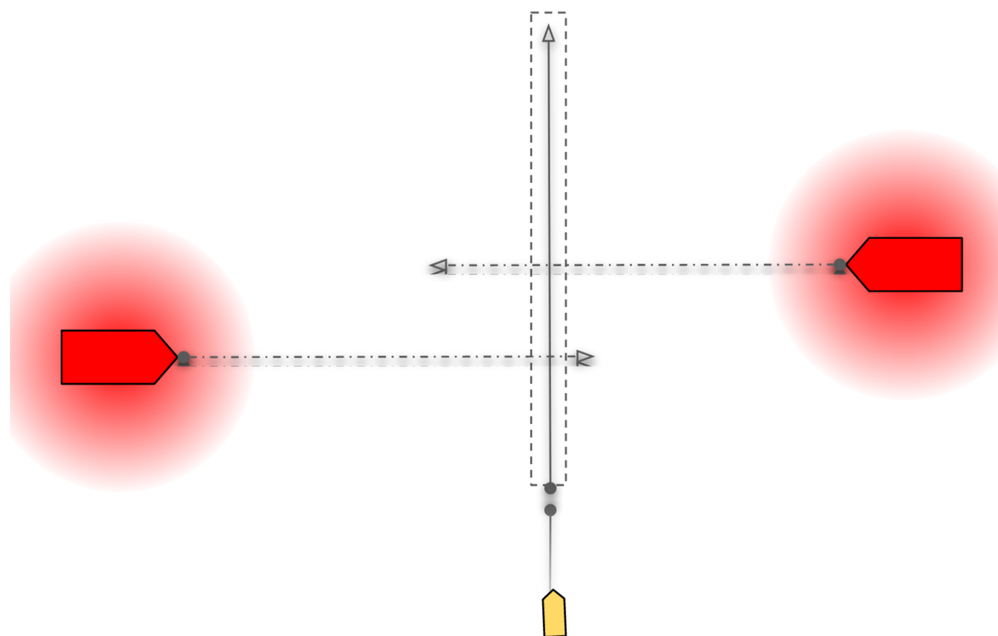


Figure B-24: Test06. ASV must navigate along a survey line with a moving obstacle from the right and left which require 50m (a), 100m (b), 300m (c), 500m (d), and 2000m (e) avoidance. A video of RBPC running in the Test06 scenario can be found here: `https://youtu.be/fhztkAFL7V4`

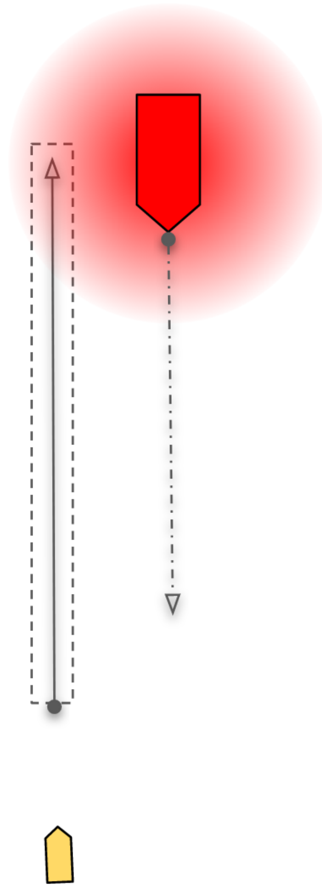


Figure B-25: Test07. ASV must navigate along a survey line with an oncoming obstacle to the right of the line which requires 50m (a), 100m (b), 300m (c), 500m (d), and 2000m (e) avoidance. A video of RBPC running in the Test07 scenario can be found here: `https://youtu.be/RHPIoS2MSgM`

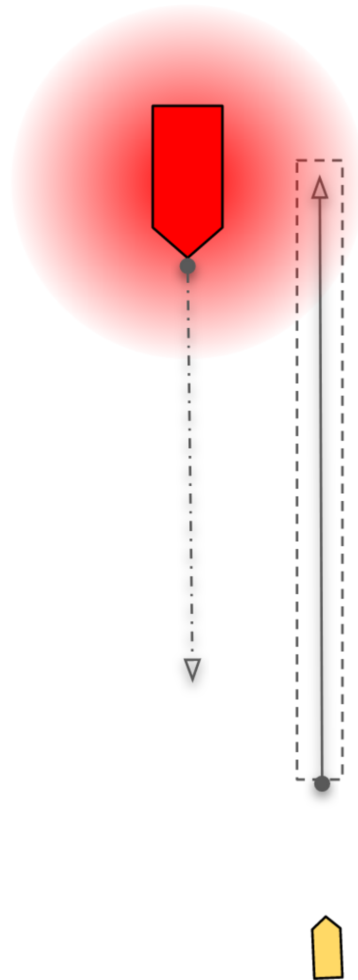


Figure B-26: Test08. ASV must navigate along a survey line with an oncoming obstacle to the left of the line which requires 50m (a), 100m (b), 300m (c), 500m (d), and 2000m (e) avoidance. A video of RBPC running in the Test08 scenario can be found here: `https://youtu.be/ru4Pj5HE_4s`

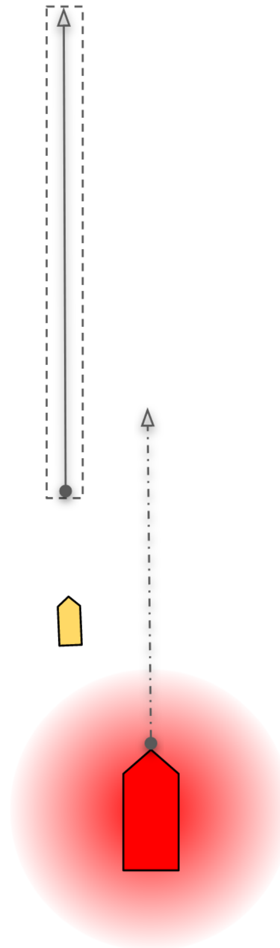


Figure B-27: Test09. ASV must navigate along a survey line with a faster obstacle from astern which requires 50m (a), 100m (b), 300m (c), 500m (d), and 2000m (e) avoidance. A video of RBPC running in the Test09 scenario can be found here: `https://youtu.be/SMx9qOK-iwE`

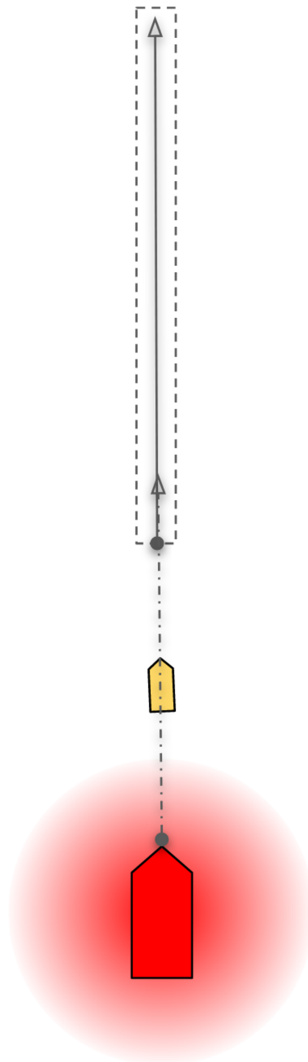


Figure B-28: Test10. ASV must navigate along a survey line with a faster obstacle from *directly* astern which requires 50m (a), 100m (b), 300m (c), 500m (d), and 2000m (e) avoidance. A video of RBPC running in the Test10 scenario can be found here: `https://youtu.be/xufyfoqKZSs`

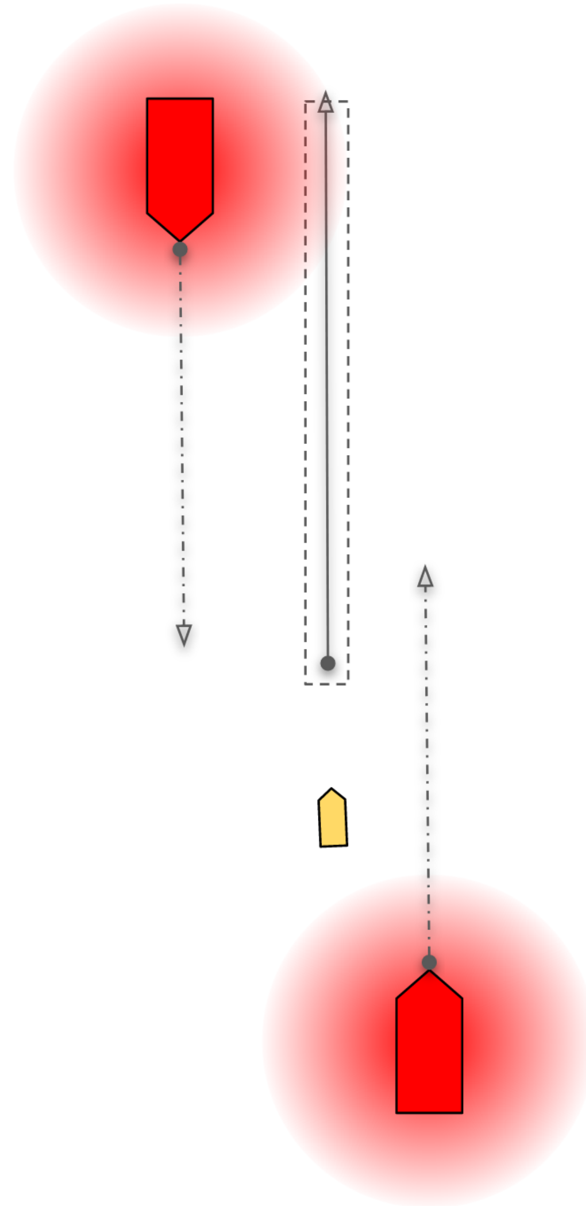


Figure B-29: Test11. ASV must navigate along a survey line with moving obstacles from ahead and astern which require 50m (a), 100m (b), 300m (c), 500m (d), and 2000m (e) avoidance. A video of RBPC running in the Test11 scenario can be found here: `https://youtu.be/w0Enj0JaS1Y`