7-2011

Parallel Algorithms for Real-time Motion Planning

Matthew McNaughton
Carnegie Mellon University, mmcnaugh@ri.cmu.edu

Parallel Algorithms for Real-time Motion Planning

Matthew McNaughton
mmcnaugh@ri.cmu.edu
CMU-RI-TR-11-30

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Robotics

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

July 2011

Thesis Committee:
Chris Urmson, Chair
Anthony Stentz
Guy Blelloch, Computer Science Department
Dmitri Dolgov, Google Inc.

This research was supported through the General Motors/Carnegie Mellon University Autonomous Driving Collaborative Research Laboratory.

Abstract

For decades, humans have dreamed of making cars that could drive themselves, so that travel would be less taxing, and the roads safer for everyone. Toward this goal, we have made strides in motion planning algorithms for autonomous cars, using a powerful new computing tool, the parallel graphics processing unit (GPU).

We propose a novel five-dimensional search space formulation that includes both spatial and temporal dimensions, and respects the kinematic and dynamic constraints on a typical automobile. With this formulation, the search space grows linearly with the length of the path, compared to the exponential growth of other methods. We also propose a parallel search algorithm, using the GPU to tackle the curse of dimensionality directly and increase the number of plans that can be evaluated by an order of magnitude compared to a CPU implementation. With this larger capacity, we can evaluate a dense sampling of plans combining lateral swerves and accelerations that represent a range of effective responses to more on-road driving scenarios than have previously been addressed in the literature.

We contribute a cost function that evaluates many aspects of each candidate plan, ranking them all, and allowing the behavior of the vehicle to be fine-tuned by changing the ranking. We show that the cost function can be changed on-line by a behavioral planning layer to express preferred vehicle behavior without the brittleness induced by top-down planning architectures. Our method is particularly effective at generating robust merging behaviors, which have traditionally required a delicate and failure-prone coordination between multiple planning layers. Finally, we demonstrate our proposed planner in a variety of on-road driving scenarios in both simulation and on an autonomous SUV, and make a detailed comparison with prior work.

Acknowledgements

I had a lot of help with this work. While I have been learning how to program robot cars, my long-suffering wife, Jenney, has become an expert at talking down overwrought graduate students. She has made many sacrifices to be with me through this time. She deserves more than I could hope to repay. I love this woman.

Many thanks go to my advisor, Chris Urmson, for recruiting me to the Tartan Racing Team. It was an unparalleled privilege to work with so many incredibly competent, hard-working people. For that matter, thanks to the team for being such, and writing the Tartan Racing code base that I have used to support my work. Extra thanks to Jarrod Snider for keeping Boss in shape for the last four years, and Young-Woo Seo for the camera equipment loan. Thanks also to Chris for the opportunities he has given me since the big race, for keeping me on track to finish this dissertation, and gently but firmly pushing me out of the nest. There's more waiting out there to be done.

I thank GM for supporting my work, and gamely going along with my approach to the problem. John Dolan deserves my undying gratitude for expertly managing our relationship with GM and for his sharp-eyed reading of many, many words on short notice.

Thanks go again to Jenney, and to my dear friends Marek Michalowski, Maxim Makatchev, Karen Hecht, and Seth Ciotti, for driving and filming many test laps around the track with me, Boss, my Subaru, and an old GMC Sierra.

A long, long time ago, at the University of Alberta, several professors got me off to a good start. I wouldn't have gotten here without their advice, encouragement, guidance, financial support, personal time, and other good things. Thanks to Jim Hoover, Piotr Rudnicki, Jonathan Schaeffer, Duane Szafron, and Hong Zhang. Thanks also to the Government of Canada and NSERC for a generous award that allowed me to focus on this task.

Thanks finally to my committee, Tony Stentz, Guy Blelloch, and Dmitri Dolgov for lending their time and their minds to assessing my work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

I must commandeer your vehicle.

Robocop

For decades, humans have dreamed of making cars that could drive themselves, so that travel would be less taxing, and the roads safer for everyone. This thesis brings us one step closer to realizing that dream. We have made strides in motion planning algorithms for autonomous cars, using a powerful new computing tool, the massively parallel graphics processing unit (GPU).

Autonomous on-road vehicles sophisticated enough to drive safely and reliably in any traffic condition would be of great benefit to society. They would decrease traffic accidents, lower costs in the transportation industry, decrease congestion and commute times, and grant independence to millions of people who are currently unable to drive because of age or infirmity.

An autonomous on-road vehicle requires a robust motion planning system. The motion planning system generates a trajectory specifying the movement of the vehicle over the next several seconds, using information provided by the perception system about the vehicle's current state, the shape of the road ahead, and the locations of other vehicles and objects on the road. As conditions on the road change, the planner must be able to provide updated plans quickly. It is a challenging problem to generate a trajectory that manifests good driving behavior in complex traffic scenarios within the tens of milliseconds available. Existing motion planning algorithms do not provide the levels of robustness and reliability required of a practical system. It is evident that both new ideas and increased computational resources must be brought to bear on this problem.

In recent years, graphics processing units have evolved from specialized chips dedicated to rendering three-dimensional graphics for video games and CAD models, into powerful general-purpose parallel processors that can run some parallel algorithms much faster than a normal CPU can run their sequential counterparts. In order to solve a problem using parallel algorithms, we need to think about it in new ways. Not only can the motion planning problem for on-road vehicles be solved using parallel methods, as we will show, but the additional power made available by the GPU allows us to make qualitative improvements in system robustness and flexibility.

In the rest of this document we describe the new advances we have made in motion planning for autonomous on-road vehicles using the GPU. In the following section we motivate our selection of the motion planning problem as a worthwhile application domain, and argue that parallel computation in general and GPUs in particular will become increasingly important in the future.

1.1 Motivation

The field of motion planning encompasses applications as diverse as computing protein interactions, creating assembly plans for complex manufactured products, deriving humanoid walking gaits, plotting dynamic aircraft trajectories, and driving ground vehicles through rough terrain. A broad range of solution methods are needed to address these disparate problem domains. For this thesis we narrow our scope to parallel motion planning algorithms for autonomous on-road vehicles, an application of considerable importance.

1.1.1 Why Autonomous Vehicles?

According to the National Highway Traffic Safety Administration (NHTSA), approximately two million vehicle crashes occurred in the US in 2007[1]. The vast majority are attributable to driver error. For example, 20% of crashes involved inadequate surveillance of the roadway by the driver, and 10% involved a distraction within the vehicle. The automotive industry has been developing increasingly capable active safety systems to prevent crashes or mitigate their consequences. Numerous auto makers offer a lane departure warning system that warns the driver when the vehicle drifts out of the current lane without first signalling. Mercedes and Infiniti offer vehicles that can actively brake the wheels on the opposite side in order to slow or stall the drift. Mercedes and Infiniti offer a blind spot intervention system that detects when the driver is about to collide with another vehicle while changing lanes and similarly nudges the vehicle back. The Lexus LS 460 can actively apply torque to the steering wheel in order to keep within the lane

Typical event in crash scenario class	Cost	Fatalities
Left turn at junction, hit opposing traffic	$10.8B	1200
Left or right turn at junction, hit lateral traffic	$7B	1200
Pass a vehicle on a highway, hit an oncoming car	$0.9B	520
Turn at intersection, hit a pedestrian	$0.8B	280
Start a left-hand turn from right-hand lane, hit vehicle traveling in the same direction	$2.8B	190
Total	$32.3B	3690

Table 1.1: Crash scenarios that are beyond the ability of modern active safety systems to mitigate; data are for the USA from the year 2004[73]

on curving roads. Crash-imminent braking systems are able to detect imminent collisions to which the driver has not yet responded and begin braking in order to lessen the impact of the collision. All of these systems are relatively new and available on expensive luxury vehicles. We are not aware of studies that estimate the number of crashes prevented and lives saved by these technologies, but they are no doubt effective. However, we contend that there are several significant types of crashes which neither gentle braking, nor minor steering corrections, nor panic braking can prevent. Table 1.1 lists a few of these types of crashes, with the estimated economic costs and loss of life they cause. In the case of a poorly-judged left hand turn, for example, panic braking in the middle of an intersection would not necessarily be helpful. To prevent this and other types of crash would require the vehicle to decide when to initiate the turning maneuver. Unless the vehicle and driver were to trade control frequently back and forth, this implies that the vehicle should drive autonomously all of the time, necessitating a complete competence over all driving tasks. We expect the cutting-edge active safety systems available in the luxury vehicle market to save more and more lives as they trickle down to less expensive vehicles. However, thousands of lives and tens of billions of dollars per year will continue to be lost until fully autonomous vehicles are able to do all of the driving.

In addition to the lives that would be saved as a direct result of the crashes prevented by autonomous vehicles, hundreds of thousands more lives would be

improved by the enhanced mobility they could offer. As people age, their ability to drive safely diminishes. The American Medical Association states that "motor vehicle crash rates per mile begin to increase at age 65"[6]. Drivers age 65 and above are four times more likely to be struck while making a left-hand turn than adult drivers aged 18 to 64[2]. The average person spends the last 8 years of his or her life without the functional ability to drive[6], significantly decreasing their independence and quality of life. Autonomous vehicles could grant independence to millions of older people who are able to move around on their own but are unable to drive safely.

Finally, it is estimated that 3 trillion vehicle-miles were driven in the United States in 2010[76], with 2 trillion of these in urban areas. Freeing humans from the burden of attending to the driving task would have an enormous impact, allowing tens or hundreds of billions of commuting hours to be spent more productively and affecting nearly everyone's life for the better.

1.1.2 Importance of Motion Planning

A competent motion planner is a necessary part of a capable autonomous on-road vehicle. As mentioned at the beginning of this chapter, the motion planner must generate a precise trajectory for the vehicle to follow in a limited amount of time. Current autonomy systems include motion planners that are able to handle some, but not all, scenarios required for an autonomy system to be competent at all driving tasks. Motion planners in the literature exist to change lanes[58, 105], turn at intersections[28] and avoid some obstacles[43], but no single one can perform all necessary actions of normal driving, let alone handle emergency evasive maneuvers in complex environments. Motion planning on-road is a complex task that requires large amounts of computing power. So far the existing algorithms and the traditional sequential computers on which they are designed to run have not been able to solve the problem. We must invent new ideas and find more computational resources to solve this problem. In the next section we will show that GPUs are a promising potential resource.

1.1.3 Parallel Computing Approach

Performance increases in traditional sequential computers are expected to remain slight for the foreseeable future. Past increases have come from architectural improvements made possible by increases in transistor count and clock speed. Although companies like Intel have been able to keep increasing transistor counts, additional architectural improvements for sequential computing have become harder to develop, and clock speeds have stopped increasing altogether. The result is that future performance increases will come mainly in the form of increasing parallelism.

Figure 1.1 shows the clock speeds of CPUs sold by Intel[20, 19, 72] since 1993. In that year the Pentium® processor was introduced at 60 MHz. Processor clock speeds doubled every two years for the next twelve years after that, culminating in 2005 with a Pentium 4® processor running at 3.8 GHz. The last six years, by contrast, have not seen a significant increase in the maximum clock speeds of processors sold by Intel. Further increases in clock speed became difficult to achieve due to the increasing power demands and the heat generated at higher clock speeds.

Though its processor clock speeds have plateaued, Intel has continued to increase transistor density. Gordon Moore, a co-founder of Intel, predicted in 1975 that transistor density on computer chips would double approximately every 2 years[18]. This prediction has held true and become known as Moore's Law. According to Intel researchers[41], Moore's Law is expected to hold true for at least the next decade, resulting in computer chips with many billions of transistors and tens or hundreds of cores. Figure 1.2 shows the number of transistors actually put onto each processor sold by Intel. The wide variation is explained by the fact that the number of transistors on a chip is a significant factor in its cost.

Figure 1.3 shows that Nvidia Corporation, a major maker of GPUs, has also increased the number of transistors on their chips, as well as the number of basic parallel computing elements that they compose. The compute cores of an Nvidia processor are individually much smaller than an Intel processor core, but

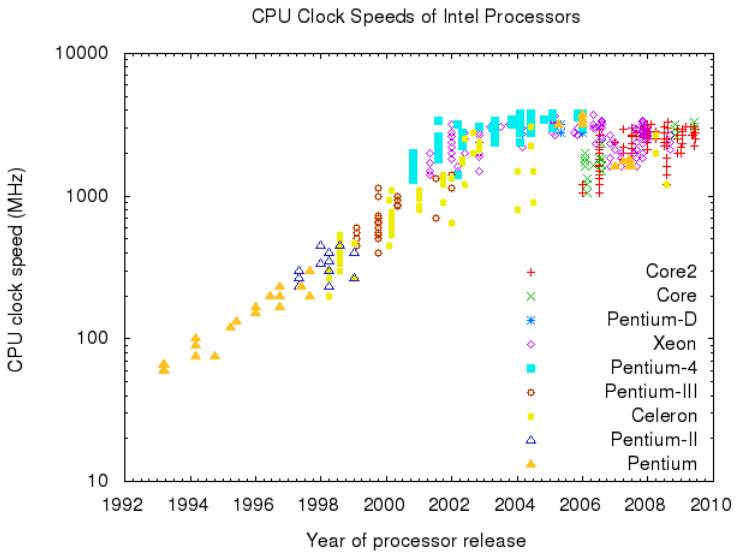


Figure 1.1: Log plot of CPU clock speeds of desktop and server processors sold by Intel from 1993–2008[20, 19, 72].

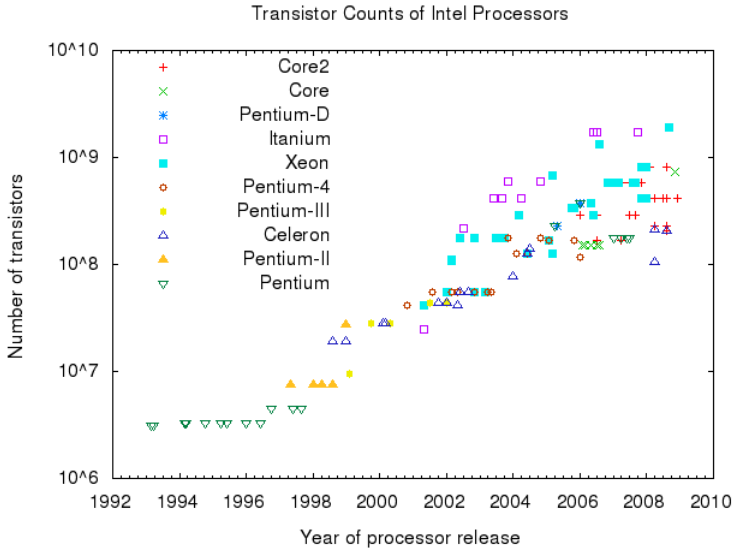


Figure 1.2: Log plot of numbers of transistors in desktop and server processors sold by Intel from 1993–2008[72, 19, 72].

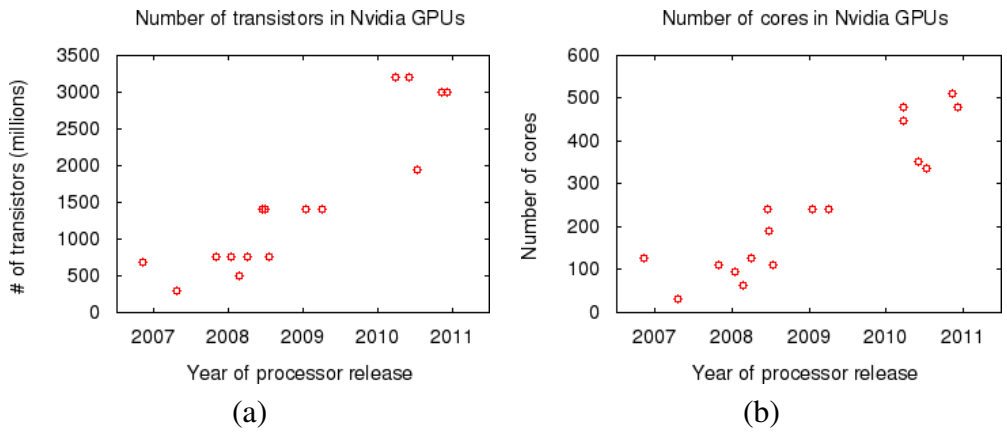


Figure 1.3: Nvidia GPUs designed for general-purpose parallel computing have shown rapid increases in the numbers of (a) transistors and (b) parallel computing cores they comprise.

the continued growth in their numbers is striking. We will explore the differences between styles of parallelism in Intel CPUs and in Nvidia's GPUs in the next chapter. Briefly, while Intel processors are designed primarily to execute sequential tasks quickly, with small amounts of parallelism added in opportunistically, the GPU is designed from scratch to run massively parallel computing tasks.

The key observation is that processor clock speeds are no longer increasing, while transistor counts are still increasing. The implication is that faster processors will deliver increased performance mainly through parallelism, which will require new algorithmic approaches to motion planning. In the following sections we specify our thesis and give an overview of our approach.

1.2 Problem Statement

Autonomous on-road driving systems require more robust and capable low-level trajectory planners before they are able to handle all of the eventualities that may occur when driving on public roads. New parallel computers offer a way to increase the amount of computational resources available to the trajectory planner.

This thesis presents new planning ideas necessary to take advantage of these new resources.

1.3 Thesis Statement

Our aim is to create a motion planner that can generate effective high-speed maneuvers for robots in complex driving scenarios. This thesis shows that:

> *Parallel algorithms can improve the speed and quality of motion planning, enabling otherwise unattainable levels of performance for real-time applications.*

By the *speed* of motion planning we mean:

- The latency of the motion planner from reception of input to completion of its output.

By *performance* we mean:

- The planner's ability to plan complex maneuvers through complex situations, and at highway driving speeds.

By *complex* we mean:

- A driving scenario requiring multiple lateral motions and multiple phases of acceleration and deceleration.

By *real-time* we mean:

- The planner must be guaranteed to produce a plan within a deadline. A real-time planner is not necessary for low-speed applications where a panic stop is always a viable and safe response should the planner fail to generate a plan in time to avoid an obstacle. However, Urmson showed[103] that above a certain speed that depends on the vehicle dynamics and sensor horizon, a swerve response may be necessary. Such a swerve response must

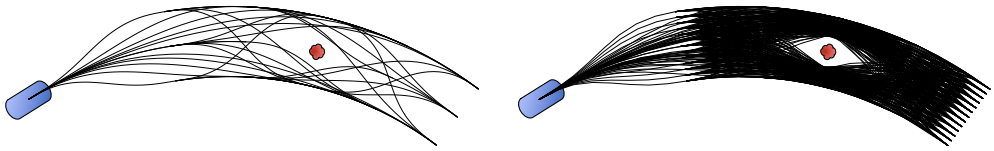


Figure 1.4: The GPU enables candidate plans to be sampled at increased density.

be calculated at least within the time that would have been needed to stop instead, setting an upper bound on the minimum latency allowed for the planner. Hence, on-road driving is a real-time application.

By *quality of plan* we mean:

- The correspondence of the plan to the actual abilities of the robot.
- The expressiveness of the plan; i.e., can we express a complex desire and expect the motion planner to return a solution optimized to satisfy it?

1.4 Approach

A contemporary graphics processing unit (GPU) from Nvidia Corporation contains a general-purpose parallel computer that can run some workloads at a significant speedup over typical CPUs. We have found that the trajectory planning problem can be significantly accelerated on a GPU. The additional cycles allowed us to take several new approaches, increasing the density of candidate plans our planner analyzes, as illustrated in Figure 1.4. The net benefit is to increase the overall robustness and flexibility of the planner. That is, a quantitative increase in computing performance allows us to make a qualitative leap in planner performance by several methods:

Search Space Decomposition Autonomous driving, like any interesting planning domain, presents a space of possible actions too large to search naïvely. We must decide how to ignore large portions of the search space *a priori* and decompose the remainder into smaller, tractable subproblems which can

be solved independently and combined into a solution to the full problem. We propose a sampling method that samples the search space sufficiently sparsely to render the problem feasible, yet with great enough coverage to find reasonable plans in a range of driving situations.

Cost Scheduling A common approach to decomposing the search space is to factor the planning system into a hierarchy, where each planner in the hierarchy operates on a simpler model of the world than the one below, but makes decisions on a broader scope. Planners receive commands from above and turn them into more detailed plans to send downwards. This rigid command structure can lead to undefined behavior when commands turn out to be infeasible upon closer examination. We contribute a mode of communication between layers that eliminates the rigidity in the chain of command: upper layers modify the cost function used by lower layers to determine which plan to follow, but do not issue direct commands. Our other contributions enable this more robust method of communicating, which requires the lower layers to take responsibility for examining many more plans.

Resolution-Equivalent Grid In a nonholonomic system such as a car traveling on a public road, temporal and spatial planning dimensions are coupled, meaning it is impossible to factor the problem along these dimensions into independent subproblems. This constrains the options available for *Search Space Decomposition*. Our planner searches in a unified space including spatial and temporal dimensions. To resolve nonholonomic constraints without increasing the grid resolution, we adapt the locations of grid points on the fly, similar to the concept of resolution-equivalence enunciated in the Incremental Search Engine[99], used in Hybrid A*[25], and the method of Barraquand and Latombe[11].

1.5 The Rest of This Document

In Chapter 2 we survey related work in motion planning for autonomous on-road vehicles, both parallel and sequential approaches, and show how the work of this thesis fills an important gap in the literature. We also present a brief overview of the GPU architecture, and review the other types of parallel computers. In Chapter 3 we describe our planning algorithm for the on-road vehicle motion planning problem. We show how to use a boundary-value problem solver to connect sampled points in the search space with high-fidelity kinematic paths, turning the exponential growth of complexity with path depth into a tractable linear growth. In Chapter 4 we look at how to set up the planning space described in the previous chapter to achieve specific driving behaviors. We show the effect of tuning cost function parameters on the vehicle's behavior, and how the behavioral layer can effect interesting and useful driving behaviors by making changes to the cost function while the vehicle is in motion. In Chapter 5 we discuss implementation issues for our planning algorithm: how we integrated it into the existing Tartan Racing infrastructure, and how we implemented our algorithm on the GPU to achieve substantial acceleration compared to a sequential version on the CPU. In Chapter 6 we evaluate the performance and capabilities of our planning algorithm compared to previous works, showing that our planner advances the state of the art in emergency evasive maneuvers. We conclude in Chapter 7, summarizing our contributions and discussing future research directions.

Chapter 2

Related Work

In this chapter we provide background on three areas related to this thesis. First, we discuss the related work in motion planning for on-road vehicles, and work on parallel motion planning in general. Second, since we have chosen the GPU parallel computing architecture to address the motion planning problem, we give the reader an introduction to the major concepts of the GPU computing architecture, which will be useful in understanding the planning algorithms presented in later chapters. Third, to justify our choice of the GPU, we review other types of parallel computing architectures available and the typical applications for which they are suited.

2.1 Planning Approaches

In this section we look at related work in motion planning, including algorithms and applications specific to vehicles, algorithms and planning ideas that are not specific to vehicles but which are used in or related to our algorithm, and general-purpose parallel planning algorithms presented for the GPU.

2.1.1 Sampling Approaches for Off-Road Planners

In this section we look at some early progress in planning algorithms for vehicles in off-road situations. We observe a trend towards increasing fidelity of the vehicle model to reality, and an attempt to generate more complex plans.

Krogh and Thorpe[52] divide the planning algorithm for a mobile robot in a cluttered environment into two parts: a global planner which uses a simple grid to plot a rough path to the goal through the environment, and a local planner which follows a potential-field formed by nearby obstacles and the path. They remarked (in 1986) that "Computational complexity precludes the use of optimal control or dynamic programming for real-time steering control." Later authors were able to break this barrier as available computational power increased, a trend that we sustain through this thesis.

Kelly and Stentz[50] explicitly abandon planning methods based on geometrical analysis of robot configuration space (C-space) such as visibility graphs through polygonal obstacles[64], and pursue a sampling approach, or *generate-and-test*. Their method samples constant-steering command trajectories, simulating each one's effect on a high-fidelity model of the system dynamics and environment. Their algorithm scores these short-range, high-fidelity trajectories based on criteria such as proximity to obstacles, roll angle, and heading towards goal, then arbitrates between candidate trajectories based on a weighting function.

Lacaze et al. propose ego-graphs[54], an approach to generating more complex motions than [50], while using a kinematic vehicle model sufficiently realistic to navigate around obstacles. They precompute a tree of steering and acceleration actions that cover the space in front of the vehicle. The size of the tree is exponential in its depth. A weakness of this work is that nodes with similar states are not coalesced. For example, if a left swerve followed by a right and a right swerve followed by a left both come to the same place, then each of their children will be evaluated, duplicating effort. As we will see, our planner coalesces similar states and is polynomial, rather than exponential, in the depth of the search graph. Bacha et al.[7] report their use of an ego-graph approach at the 2007 DUC, employing an A* search to find a sequence of steering actions in a precomputed table.

A common approach to planning for robotic vehicles, especially in earlier work, is to use a crude model of the vehicle kinematics and its environment to create a roughly optimal planned path. The path may then be followed in simulation by a local planning algorithm using a higher-fidelity model including more accurate kinematics and dynamics, and tested for safety. For example, Leedy et al.[59] use a crude model of a vehicle to generate a rough path for a vehicle traveling along a dirt road using A* search. They then smooth the path using a pure-pursuit path tracker[21] with a higher-fidelity model. A final safety check is performed on the resulting path before it is executed on the vehicle. Two problems with this approach became evident: first, if the vehicle approximation is an underestimate of the vehicle's true capabilities, then the global path can be much

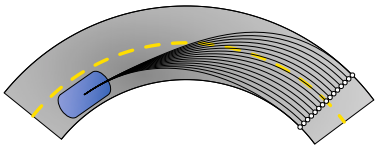


Figure 2.1: The Tartan Racing motion planner for structured driving in lanes[29]

more expensive than the true optimal path. Second, if the planner overestimates rather than underestimates the vehicle's abilities, then the local planner will be unable to find a way to follow the global path at all, leading to a stalled vehicle, or worse.

Alternatives to the pure-pursuit path tracker[21] are reviewed by Snider[96]. Even high-fidelity plans may benefit from tracking using closed-loop control rather than using model-predictive control. We will discuss the control architecture used in our implementation in Chapter 5.

2.1.2 Swerve-Sampling On-Road Planners

The literature contains several approaches to on-road vehicle planning that sample a variety of swerves, and a variety of acceleration profiles, evaluate each according to some criteria, and pick the best.

Ferguson et al.[29] present a "local planner" which drives along a lane using a reference center line describing the road shape. It samples candidate endpoints spaced evenly across the road at some distance ahead, and for each endpoint optimizes a velocity-independent cubic polynomial that steers the vehicle directly onto the point with an orientation parallel to the road, illustrated in Figure 2.1. Over each such path it samples several acceleration profiles to find the best overall trajectory. This planner was demonstrated in the 2007 DARPA Urban Challenge(DUC)[23], where their robotic vehicle drove in a suburban environment with other vehicles. The overall system is described by Urmson et al. in [28].

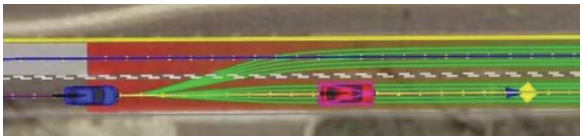


Figure 2.2: Lateral offsets sampled from a reference trajectory by Stanford's Junior[68]

Similar to [29], Montemerlo et al.[68] propose a planner that samples paths as swerves, offset from a reference path defined by the center of the lane, as shown in Figure 2.2. This approach was sufficient for simple traffic interactions involving, for example, changing lanes to pass a single slow-moving vehicle. This planner was also fielded in the 2007 DUC, within a broader system described in [69].

Werling et al.[105] propose a planner that samples a set of lateral motions away from or towards the road center line, another set of motion profiles along the road center line, and evaluates all combinations of the two sets. They use quintic polynomials for both types of motion, and show that these polynomials are optimal solutions for paths that minimize jerk. By also using a jerk-minimizing cost function to select plans, their method minimizes transients introduced by replan cycles. Their goal is a low-level planning layer that can carry out simple strategies mandated by a higher-level behavioral planning layer while reacting quickly and flexibly to unexpected situations.

While the preceding planners represent paths as essentially a single swerve action, Kuwata et al.[53] use a rapidly-exploring random tree[57] to generate on-road plans containing longer sequences of motions. Their RRT-based motion planner samples in the space of target points for the steering controller(Figure 2.3) and longitudinal accelerations. The set of sampled plans is represented as a tree of potential vehicle trajectories. The tree is extended by forward-simulating the entire vehicle system with the steering controller's reference point set to a newly sampled point. Planner efficiency is increased by heuristically biasing the sample distribution. Some sample points are designated to accelerate the vehicle down to zero velocity. The planner runs until it finds an acceptable path that ends with the

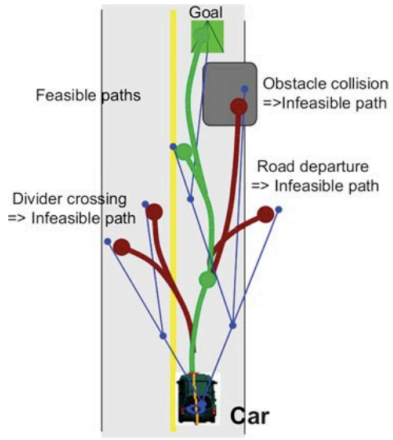


Figure 2.3: RRT-based motion planner for MIT's Talos, from[60]

vehicle at a stop. It then selects the best of these paths, which the vehicle follows to the ending stop unless a new plan can be generated in time. This planner cannot guarantee that it will react quickly to sudden changes in the environment. It was demonstrated on MIT's Talos vehicle[60] at the 2007 DUC, and we discuss it further in Chapter 6 where we compare it to our planner.

2.1.3 Lane Change Maneuvers

In this thesis we seek a general-purpose motion planner to generate trajectories realizing a variety of behaviors. Several researchers have examined the generation of an optimal lane change maneuver, and safety bounds required for such maneuvers, outside of a general-purpose motion planning context.

Kanaris et al.[47] determined minimal vehicle spacing requirements before a lane change could be initiated in order to ensure that an accident could be avoided should the leading or merging vehicle suddenly brake, taking into account the varying performance of the tire in braking when undergoing lateral motion. They include the possibility that the merging vehicle must change speeds in order to complete the merge, and they examine the potential behavior of the vehicles involved given six different degrees of communication between the vehicles in an

Automated Highway System context. Jula et al.[46] perform similar work, though limited to a kinematic mode of the vehicle.

Papadimitrious and Tomizuka[79] analyze lane changing in the presence of multiple obstacles. They claim that a search-based method is not desirable because of its lower computational efficiency. They propose instead a geometric formula to generate the path for the maneuver, devised to efficiently perform collision checking. They show a solution for the emergency "three-vehicles problem", where a vehicle traveling in a lane with a static obstacle ahead must merge into a short gap between two vehicles in a neighboring lane before hitting the obstacle.

Godbole et al.[38] propose a trajectory generation method for the "three-vehicles problem" with lane changing/merging. Their plan involves several stages, where the gap is first selected, then the vehicle aligns with it, and finally moves into it.

Lee and Litkouhi[58] propose a lane change maneuver using a quintic polynomial that accounts for lateral forces on the tire. They sample several candidate solutions to find one that satisfies lateral acceleration bounds, but they do not address obstacle avoidance.

The planners reviewed in this section and the preceding one work by generating a tree of sampled swerve actions. With most of them, the forms of plan considered are single, simple swerve actions – a single level in the tree. The ego-graphs[54] of the previous section and the RRT[53] just mentioned are exceptions. In those cases, a longer sequence of actions can be considered, but we note that the computational complexity is exponential in the length of the sequence. We look next at attempts to reduce that complexity.

2.1.4 State Lattice

As an attempt to bridge the fidelity gap between local and global planning layers mentioned previously[52, 59], Kelly and Pivtoraiko[83] introduced the state lattice, a means of embedding a discrete graph into the continuous robot configuration space, where vertices in the graph represent robot states, and edges joining vertices represent trajectories that satisfy applicable constraints on the robot's mo-

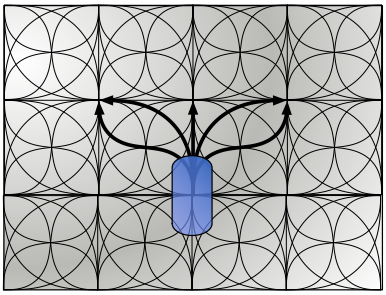
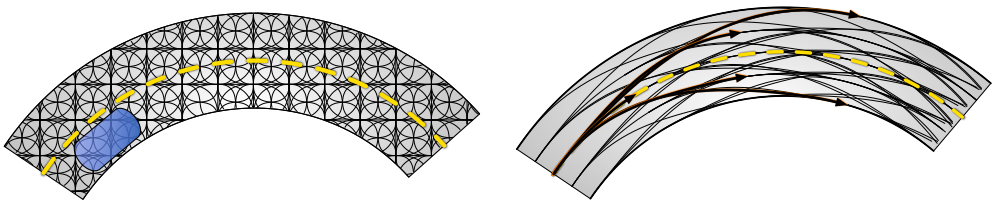


Figure 2.4: A simple state lattice.

tion. Figure 2.4 illustrates a simple state lattice. The benefit of the state lattice is that plans generated via a search through a state lattice are closer to the robot's actual abilities than prior approaches, though it typically requires more computational resources to generate and search the lattice. An enabling technology to build the state lattice is a boundary value problem solver for the robot that can find a path between pairs of vehicle states it is desired to connect in the lattice. A fast solver and accompanying polynomial spiral path representation is given by Kelly and Nagy[49].

The state lattice approach has been shown to be effective for planetary rover navigation applications[83] and a wheeled indoor assistant robot[91]. It was also used at at the 2007 DARPA Urban Challenge(DUC)[23], where the Tartan Racing team[28] used it for driving situations such as parking lots and error recovery[62, 31, 29].

The benefit of the state lattice approach to plan quality and performance is that all paths within the lattice use a relatively high-fidelity kinematic model of the vehicle, and these paths may split and re-join, turning a tree into a graph, and potentially reducing the exponential amount of work done in the length of the path to polynomial.



(a) (b)

Figure 2.5: (a) Naive application of the state lattice to a road. Many edges in the lattice are not useful. (b) The state lattice conformed to a road.

2.1.5 On-road Lattice Planners

The lattice planner concept has been applied to on-road vehicles driving in traffic, although the naive approach of laying the structure depicted in Figure 2.4 on top of a road is not workable. Figure 2.5(a) shows how an attempt at this might look. The state lattice as formulated by Pivtoraiko et al.[84] is suitable for unstructured environments where any robot heading could reasonably be in the final solution. However, on a road the selection of feasible headings is highly constrained. At each point along the road, only a small range of headings close to the orientation of the road should appear in a reasonable plan. Therefore, the state lattice must be adapted to the environment such that it includes only states that are a priori likely to be in the optimal path. Figure 2.5(b) illustrates the idea.

Rufli and Siegwart[90] assume that using the iterative method of [49] to construct the paths in a lattice like Figure 2.5(b) is too time-consuming. Defining a road line as a C^1 path, they propose a closed-form calculation for constructing a lattice warped onto the road. Their method generates the lattice vertices and edges given some constraints on the sampling strategy of the steering and throttle input variables, limited to robots with kinematics including typical vehicles. They present resulting paths through the lattice on a road with static obstacles, but do not report the number of lattice edges that can be generated per second using their formula.

Ziegler and Stiller[106] propose an augmentation of a conformal state lattice with time and velocity dimensions in order to plan with moving traffic. They

sample lattice points in a space warped along the road center line, and use quintic polynomials defined as a function of time over the longitudinal and lateral motions along the road to join the lattice points. Their approach requires the trajectories connecting lattice points to start and end at one of a fixed set of times T and velocities V. The use of fixed values in T and V makes it difficult to plan to merge between two vehicles traveling with velocities not contained in V, and requires a finer discretization in order to use low accelerations. They point out that it is difficult to compose A* search heuristics for the energy cost terms useful for tuning vehicle behavior, such as squared jerk of the path. They therefore propose an exhaustive, rather than heuristic search in order to find the optimal path.

These adaptations of the state lattice concept to an on-road context don't deal satisfactorily with the time and velocity dimensions that we must consider for higher-speed driving applications, but which could be neglected for the low-speed rover and parking lot applications mentioned in the previous section. For low-speed applications, actions can be devised to modify the (x, y, θ) configuration variables independently, allowing the search to proceed on a simple grid. At higher speeds, time and velocity must be added as state variables to enable navigation among moving obstacles. Dynamic constraints induce a coupling that makes it impossible to modify the variables independently of one another. A simple grid structure is untenable in this case since it assumes that state variables may be changed independently.

In the next section we survey attempts to overcome this coupling in high-dimensional search spaces.

2.1.6 Resolution-Equivalent Grid

Many planning methods factor the space of possible system states into a discrete grid and search for a plan as a sequence of moves from the start state, through a sequence of grid points, to a goal state. Such methods have been proposed both for driving problems and more general classes of search. An important class of grid we call the *resolution-equivalent grid*, following a name suggested by

Tompkins et al.[98] for its use in constructing a resolution-complete planner. In this type of grid, each cell has one distinguished interior point, and the plan is a sequence of moves through these movable interior points, rather than the corners. The distinguished point is determined in the progress of the search. This method is often used for nonholonomic systems, including car-like vehicles, where state variables are coupled. Tompkins[99] proposed the Incremental Search Engine (ISE), which uses the resolution-equivalent grid concept to search for long-term mission plans for autonomous mobile robots. ISE divides the state variables into two sets: independent and dependent. The independent variables are manipulated directly by the action set, and can be modified independently of one another, hence the name. For example, a holonomic robot moving in the plane can change its x and y coordinates independently to land on a grid point in the (x, y) space. However, it cannot change its time t and velocity v coordinates independently of these to land at desired grid points along the t and v axes. If acceleration were infinite, then $x_2 = x_1 + vt$ and we could choose one of t and v to join x as an independent variable, but in the realistic case of bounded acceleration it is practical to treat both variables as "dependent". Dependent variables typically change as a side effect of the independent variables. Therefore they are identified by the grid cells they fall into, as opposed to the grid corner points that identify the independent variables. Two states in the space that have the same values for their independent variables, and whose dependent variables fall into the same grid cells, are considered equivalent.

Barraquand and Latombe[11] use a resolution-equivalent grid in a planner for a nonholonomic car-like mobile robot with a trailer. Their planner builds a tree of states by iteratively selecting a state already in the tree and extending it outwards using some control trajectory. The first trajectory endpoint state to land in a cell becomes the distinguished state, and later trajectories whose endpoints land in the same cell are pruned away.

Dolgov et al.[25] propose Hybrid A*, an adaptation of the classic A* search algorithm[74] to finding trajectories for a car-like vehicle using the resolution-

equivalent grid concept. Edges in the Hybrid A* search graph are represented as vehicle trajectories, and vertices in the graph as grid cells in the vehicle state space. In contrast to the algorithm of Barraquand and Latombe[11], where the first trajectory endpoint to reach a cell becomes its distinguished point, with Hybrid A* the distinguished point of a cell may be reassigned if the A* search portion of the algorithm finds a lower-cost way to reach any other point in the cell.

In this section we looked at methods to search for plans within a grid without being constrained to the grid points, which can be problematic when there are dependencies between state variables. The works we surveyed did not deal simultaneously with spatial and temporal dimensions for a robot with nonholonomic kinematics. Our aim in this thesis is to take this step by combining the resolution-equivalent grid with a state lattice conformed to the road. In the next section we look at related work in solving the boundary value problem needed to connect edges in the lattice.

2.1.7 Boundary-Value Solvers

As mentioned in the previous section on state lattices, a boundary-value problem solver is necessary to connect states in the lattice. The solver must be fast enough to generate many edge paths, and the paths must be suitable for driving.

Dubins[26] showed how to generate minimal-length continuous paths between two points given that the paths have a bounded curvature R^{-1}. This is a good approximation to a vehicle driving forward with a limited turning radius. The resulting paths consist of straight lines and circular arcs of minimum radius R. These are not suitable for real vehicles, since to follow the paths precisely would require the vehicle to turn the steering wheel at an infinite velocity at transitions between the path elements.

Reeds and Sheep[87] extend the Dubins model to allow paths to reverse direction. Again, the shortest paths are composed of straight lines and arcs of minimum radius.

Shin and Singh[94] propose that a sequence of clothoids can interpolate de-

sired waypoints for the vehicle to travel while maintaining continuous curvature and with a piecewise constant steering rate. Scheuer and Fraichard[92] propose a "bi-elementary" path - a concatentation of two clothoids which can connect any two configuration endpoints for a car-like robot. They use this to construct a complete planner. Real vehicles have finite steering acceleration, but at low speeds, we can assume that infinite accelerations will induce negligible error. At higher speeds, paths with finite acceleration may be necessary.

Kelly and Nagy[49] propose that a cubic polynomial spiral, that is, a curve defined by a third-order polynomial function of curvature with respect to arc length, can neatly connect pairs of vehicle states specified by position (x, y), heading θ, and curvature κ. They present a fast root-finding method to solve for the parameters of a cubic polynomial spiral joining two states of the form (x, y, θ, κ), and also for higher-order polynomials, also called *generalized Cornu spirals*, to enforce continuity in higher-order derivatives of κ. In this thesis we use the cubic and higher-order polynomial described in this paper to generate paths with continuous curvature.

In their swerve-sampling approach to on-road motion planning, Werling et al.[105] propose that at higher speeds, the dynamic constraints on the vehicle are more restrictive than the nonholomic kinematic constraints. Therefore they use quintic polynomials with a simple closed-form solution to solve independently for endpoint constraints in a two-dimensional space conformed to the road. At lower speeds the planner must use a different trajectory representation to avoid violating constraints, and a translation step is required to express the dynamic path in terms of the vehicle kinematics.

2.1.8 Trajectory Deformation

Paths are expensive to compute from scratch, but once obtained they may be cheaper to keep updated as obstacles move. Trajectory deformation approaches attempt to continuously deform a path or trajectory in order to improve its utility.

Brock and Khatib propose the Elastic Strips method[14], which deforms a path

according to virtual forces, representing the robot dynamics as internal forces, and repulsive fields formed by obstacles as external forces. Their approach does not take moving obstacles into account, and must begin with a valid path obtained by some other method.

Fraichard and Delsart[35] describe `Teddy`, a trajectory deformation algorithm that extends the potential fields formed by obstacles along the time axis. Their method samples fixed points on the robot and uses the Jacobian of the point location with respect to the configuration variables to allow an articulated robot to avoid moving obstacles. Like Elastic Strips, `Teddy` requires an independent global motion planner to provide a valid path as input.

Ratliff et al. present CHOMP[85], similar in spirit to `Teddy`, but able to take an invalid path as input, and better able to induce control changes at points earlier in the path in order to satisfy constraints on points later in the path, resulting in smoother paths. CHOMP does not handle moving obstacles.

In this thesis we are concerned with global path generation for on-road vehicles. Trajectory deformation approaches in the literature either do not handle moving obstacles or require a valid global path in the first place. We note that they may be useful for improving paths once they are found by the global planner, though this is outside the scope of this thesis.

2.1.9 Obstacle Representations and Heuristics

In this section we survey research on the general problem of planning in dynamic environments, and representations for moving obstacles. The on-road driving problem can be seen as a special case of this general problem.

Early work in planning amongst dynamic obstacles for real-time applications tended towards reactive techniques, due mainly to the high computational demands compared to the available computing power. Fox et al.[33] propose the Dynamic Window approach, which searches in the space of velocities for a holonomic robot given dynamic constraints, moving obstacles, and a fixed time interval. Minguez and Montano propose Nearness Diagrams[67], which perform a

geometric analysis of nearby obstacles in order to produce a reactive action which avoids getting the robot stuck in dead ends, a persistent drawback of reactive methods.

Fiorini and Shiller[32] propose a search method for dynamic environments based on the idea of *velocity obstacles*. Their method transforms the planning domain into velocity space, and searches for avoidance maneuvers for the robot in that space. They consider disk obstacles and consider constraints on accelerations, which they call *dynamic constraints*. They avoid the complexity of nonholonomic kinematic constraints by observing that at high speeds the dynamic constraints are typically more restrictive. They demonstrate their method in a simulated high-speed driving problem.

Fraichard and Asama defined an Inevitable Collision State(ICS)[34] as a state s of a robotic system which is guaranteed to result in a collision with an obstacle no matter what action is taken. The ICS notion is an extension to static collision checking, which simply tells whether a robot is *currently* in collision with an obstacle.

Erdmann and Lozano-Perez[27] introduced the idea of a configuration space-time in planning motions for multiple bodies considering constraints on velocity. They formed a discretization of the configuration spacetime and computed all valid poses for a robot over a time range. They then searched a graph connected by regions where valid configurations overlapped in consecutive spacetime slices. Their solution did not consider dynamic or kinematic constraints and could only search for feasible, but not optimal paths.

Compared to these earlier works, in this thesis we propose an algorithm that is focused specifically on the on-road driving problem, where it searches for a global optimum considering space and time dimensions, and upgrades the discrete binary collision test to a continuous cost function that can take many more factors into account.

2.1.10 Catalog Approaches

Autonomous vehicles will need to be extremely reliable in order to be viable in the market. Some researchers take the view that in order to achieve the required level of reliability they must be engineered by starting with a complete catalog of all entities that could appear on the road, and then constructing an automaton with a specified response to each one.

Furda and Vlacic [36] construct an automaton comprising every situation one could encounter while driving, and use it to select the driving behavior most appropriate to the situation. They state that they are concerned with generating behavior in real-time with a verifiable system taking explainable actions. They demonstrate preliminary results showing the overtaking of a stopped vehicle with no oncoming traffic, overtaking with one oncoming vehicle, and queueing at a stop sign.

Horst et al. have applied the 4D/RCS[3] design methodology for complex electro-mechanical systems (CEMS) to autonomous on-road driving. Their design philosophy is fleshed out into a detailed model which offers an interesting contrast to our approach.

In [10], the authors distinguish between deliberative, reactive, and behavioral architectures. A reactive system has no world model and reacts according to direct sensory-actuator mappings. An example of a behavioral architecture is Brooks-style subsumption[15], where a reactive system is augmented with more complex and time-extended procedures but still lacks a coherent world model. A deliberative system has a coherent world model and reasons explicitly about possible actions and their outcomes. 4D/RCS is conceived of as an architecture and design methodology for deliberative systems. The authors of [10] propose a layered architecture where higher layers are concerned with abstract representations and long-term goals, with each of them sending "commanded subgoals" down to the next-lower layer to implement, recursively. Layers send "status" messages back up to higher layers if required to re-evaluate their plans. For autonomous driving, they propose a hierarchy of decision tables containing an exhaustive specification of all entities one could encounter on the road and each distinct situation that could

condition the vehicle's actions.

In [43] the same authors propose a trajectory-generation layer similar in scope to the planner presented in this thesis. They eschew a search-based approach, assuming that a higher-level behavior planner will be able to cheaply determine a sequence of pose checkpoints such that the trajectory generator's task is simply to interpolate between the checkpoints and modulate the velocity to stay within relevant dynamic constraints. If a collision would be inevitable given the pose checkpoints, then the trajectory generator sends a failure status up the hierarchy, but it is not clear what they propose the upper levels would do to fix the situation.

Using a hard-coded set of decision tables requires one to embed the (implicit) cost function into the planner's search algorithm in an intimate way which may make it difficult to change. We believe that as long as the search space contains the required action, and the cost function is engineered to rank it first, it should be immaterial by what method the planner produces the desired action. Compared to the 4D/RCS approach, we propose an implementation of a trajectory planning layer (designated "ElementalManeuverSubsystem" in [9]) and suggest an alternative organization of the layers - rather than commands coming down from the behavior layer to the trajectory layer and "status" going back up to it, we propose that the behavior layer express preferences by modifying the cost function that ranks potential plans, and allow lower layers to examine an exhaustive set of plans in order to determine the best available plan given the cost function. The status returned is either a plan or a failure.

2.1.11 Motion Planning in Parallel

In this section we present related work on parallel planning algorithms. Some parallel planning algorithms in the literature are presented specifically for the GPU, although in some cases they could as well be used on other parallel platforms. Other planning algorithms are presented in a more general context.

Henrich[42] reviewed a variety of parallel approaches to motion planning in the literature in 1997. His review does not include any attempts at planning in

dynamic environments. A variety of methods are presented for parallel construction of potential fields; representation of C-space using sets of bitmaps; graph search algorithms including parallel Dijkstra and parallel randomized search; and Voronoi diagrams for cell decomposition. None of the methods in the literature is of particular use in driving tasks. Effective parallel algorithms are usually devised with a specific computing architecture in mind, and the various parallel computers available at the time do not resemble the contemporary GPU.

Martinez-Gomez and Fraichard[65] applied GPUs to the problem of checking inevitable collision states for a robot moving among dynamic obstacles in a planar domain. Their technique depends on the identification of a set of evasive and imitative maneuvers, such that it is reasonable to assume that if no maneuver in the set can avoid all obstacles, it is not possible to do so. The benefit of the ICS is that it can help to eliminate states early in a recursive search such as A*, thus preventing their descendant states from being considered.

Kider et al. propose R*GPU[51], a parallel extension of R*[63], which is itself a randomized modification of A*[74]. R* periodically stops searching towards the usual A* goal and searches instead towards randomly chosen subgoals in order to escape from local minima. The R*GPU performs multiple randomized subsearches in parallel. The authors report between 5 and 35 times as many successful search results on a planar 6-DOF arm planning problem within a fixed time window when running on a 128-core Nvidia GPU(cf. Figure 1.3), as compared to a sequential implementation of the algorithm running on a single core of a conventional CPU.

Pan et al. present a parallel extension[78] to Probabilistic Roadmaps(PRMs)[48] implemented on a GPU. They focus on the problem of a high-degree-of-freedom robot in a static environment, a typical application of PRMs. Their method samples the vertices for the PRM graph in parallel, performs collision detection for pose samples in parallel, and performs the local planning to build edges between vertices in parallel. In addition, they use parallel data structures implemented on the GPU such as a parallel k-nearest neighbor search[77] and a recursive obstacle

volume representation[55]. They also use a parallel breadth-first search to accelerate single queries to the PRM. They present results for a simulated high-dof robot arm in a static environment.

Gayle et al.[37] use a GPU to accelerate collision detection in a planner for a deformable robot in a complex stationary environment.

The distance transform takes a binary image as input and computes the distance from each unset grid cell in the image to the closest set cell. This is useful both to check for collisions of robot configurations with the environment, and building potential fields to define cost functions over robot trajectories. Rong and Tan[88] present an approximation algorithm to the distance transform for the GPU. Cao et al.[16] propose an exact distance transform(EDT) algorithm for the GPU which is faster than previous approximate solutions.

In summary, GPUs have been applied to a variety of algorithms related to motion planning. Collision checking is a particularly suitable application area for GPUs, since it is essentially an operation on images, a task for which GPUs were originally designed. GPU applications to the planning algorithms themselves have been limited to general-purpose solutions that assume expansive search spaces with massive, readily exploitable parallelism. Specialized problem domains such as on-road vehicles will require tailored solutions to benefit from the GPU.

2.1.12 Summary

We have reviewed a range of work in motion planning relevant to our aims for autonomous on-road vehicles. On-road driving can be seen as a problem of sampling swerve-like actions. We have taken inspiration from the state lattice approach for off-road driving as a means of composing sequences of such swerve actions while avoiding the exponential blow-up in states to which naïve sampling methodologies are prone. Motion planning at high speeds introduces coupling between state variables that must be incorporated into the search space. We have identified the resolution-equivalent grid method as a means to accomplish this while avoiding yet another potential source of exponential blow-up. We have se-

lected a boundary-value problem solver for polynomial spirals in order to generate feasible edges between states in the lattice graph.

2.2 Parallel Computing

In the following section we introduce the reader to the GPU, and provide the reader with a survey of alternative parallel programming models to the GPU, to compare their various strengths and weaknesses and show why the GPU is a suitable choice for on-road motion planning. We also give a brief introduction to the major concepts in complexity analysis of parallel algorithms.

2.2.1 Introduction to the GPU

In this section we describe the major architectural features of the GPU that are most salient when selecting and designing algorithms to maximize performance. The interested reader can refer to Appendix A for an in-depth introduction.

The GPU processor is composed of a variable number of *scalar cores*. Each scalar core can execute one instruction per clock cycle, though complex instructions such as division and built-in transcendentals may take longer. The scalar cores are gathered into groups of eight that share a small cache of programmer-addressable local memory, and each scalar core has a large number of registers to support the private storage needs of hundreds of threads. Each scalar core within a group shares a single instruction issue unit, so each of the eight scalar cores execute the same instruction at the same time, albeit on different data. Groups of scalar cores can only exchange data by storing to and reading from the global device memory. Higher-performance GPUs are obtained mainly by increasing the number of scalar cores on the chip.

Data-Parallel Operations

An operation that performs the same computation on several pieces of data at once is termed *data-parallel*. Data-parallelism stands in contrast to *task-parallelism* where multiple threads of control may simultaneously perform entirely different functions. In order to gain the benefits of the GPUs for motion planning, it is necessary to develop algorithms that can use data-parallel operations

Data-parallel computers typically organize data elements into fixed-size vectors. On the Nvidia GPU for example, data-parallel operations are performed on vectors of 32 elements. Conditionals such as an `if()` statement conditioned on the element values suffer reduced performance as operations are performed sequentially on the elements taking the "true" branch, and then those elements taking the "false" branch. This data-parallel computing model where the same instruction is run on multiple data elements is also called *single-instruction multiple-data* (SIMD).

Memory Architecture

In traditional CPU architectures, cache misses are extremely detrimental to performance. All execution stops until the desired information is retrieved from memory further away from the CPU. The most common way to mitigate this problem is by using larger caches. Another way is to build hardware that can switch instantly to another runnable thread when one thread is stalled.

GPUs use a comparatively small amount of cache memory, and rely on the programmer to write algorithms that use hundreds of threads. The GPU schedules threads in hardware at a fine grain, so that when a thread stalls waiting for a memory access to complete, another thread can be run immediately. If enough threads are available, then the GPU cores can be kept working even when threads must frequently wait for memory operations to complete.

Memory bandwidth is the aggregate amount of data that can be transported between main memory and the processor per unit time. GPUs have a considerably

higher memory bandwidth than CPUs. The Intel Core 2 Duo, for example, has a theoretical bandwidth of 8.5 GB/s, whereas the Nvidia GTX 260 has a memory bandwidth of 112 GB/s. The higher memory bandwidth of the GPU comes as a result of a wider data path, implying that peak bandwidth is only achieved when vector loads or stores access contiguous memory locations.

The GPU for Motion Planning

GPUs are a data-parallel SIMD architecture that hides memory access latency by using many threads rather than a cache. A high memory bandwidth allows the GPU to process large amounts of data, as long as memory accesses issued at the same time are contiguous. The GPU is an excellent fit for the on-road motion planning problem. Referring to Figure 2.5(b), we note that at each step along the road, the lattice contains many similar path structures that do not depend on one another, and so can be evaluated in parallel. We will argue in more detail later that the hard real-time requirements of on-road driving requires all trajectories edges to be evaluated in order to guarantee a feasible solution. The highly regular structure, and the requirement that all trajectories must be evaluated, allows the efficient use of the SIMD operations offered by the GPU. Lastly, the large numbers of independent trajectories in the lattice ensure that there will be enough work to occupy many threads, allowing the latency-hiding features of the GPU to come into play and maximize utilization of the scalar cores.

In the following section we review other parallel computing architectures and introduce some background in complexity analysis of parallel algorithms.

2.2.2 Alternatives to the GPU

We chose the Nvidia GPU with its proprietary CUDA development environment to demonstrate our planner. One alternative would have been GPUs from AMD, which use the industry standard OpenCL GPU programming framework that works on chips from Nvidia, AMD, IBM, and Intel. Intel recently cancelled a GPU in de-

velopment called Larrabee[93], which would have combined CPU elements onto the GPU, allowing tasks that could not be executed efficiently with data-parallel operations to be handled on the GPU, avoiding the slow transfer of data to the system's main CPU which is currently necessary with Nvidia GPUs. Intel has since begun to released graphics processors such as Sandy Bridge which are integrated onto the CPU die, but they are not powerful enough to be considered for this work.

We used an Nvidia GPU with CUDA for three reasons. First, OpenCL was not available when we started this project, high-performance Intel products supporting OpenCL acceleration are still not available, and IBM's Cell[81] processor is older and more difficult to program. Second, since we were writing experimental software rather than releasing a product, we had no need to develop portable software by switching to OpenCL. Third, by using CUDA we were able to fully exploit features of the GPU that might not be exposed by the OpenCL abstraction.

We believe that fundamental design constraints will ensure that the algorithms we have devised for the Nvidia GPU will be applicable to future parallel processors. Traditional CPUs are designed to execute a single sequential thread of control with low latency by allocating a large proportion of transistors to such optimizations as speculative execution, branch prediction, a provision of excess functional units, analysis of instruction-level parallelism, and code translation. Both Nvidia and AMD GPUs, as well as IBM's Cell and the cancelled Larrabee eschew these optimizations and allocate more transistors to processing units. They use data-parallel operations to run a large number of independent threads of control with a higher latency on the individual level, but with a higher aggregate throughput due to the greater number of processing units. We will expand on these principles of operation in Section 5.3.1.

With a future device that provides higher bandwidth and lower latency between GPU and CPU, for example by integrating the components onto the same chip, it may be possible to simplify or otherwise improve some of the GPU-specific algorithms we present in Section 5.3. The overall design, however, should remain stable.

2.2.3 Distributed Systems

While GPUs accelerate data-parallel operations, a more general class of parallel systems are distributed systems. Distributed systems use large clusters of computers to solve large problems that can be broken into smaller chunks. The largest parallel computer systems are distributed systems composed of many smaller computing nodes. For example, the smallest supercomputer sold by Cray, the CX1, resembles a network of normal servers. It uses Xeon processors, with the largest configuration offering 6 computer "blades" connected with Infiniband network adapters, and each with dual quad-core Xeon processors.

Google's MapReduce[24] application is a well-known example of a software framework for programming distributed systems computing the results of massively parallel problems.

Some robotics applications use ad-hoc distributed approaches. Robotics applications typically have a profusion of software processes, handling a variety of perception and planning tasks, as well as ancillary functions such as a user interface. In the DARPA Urban Challenge[23], many entrants[60, 80, 28, 86, 70, 13, 7, 66, 17] used distributed computing systems.

While distributed systems can increase throughput for large computations, just as the SIMD parallelism employed by GPUs can, the latency of communications between computing nodes can make meeting real-time requirements of on-line motion planning more challenging.

2.2.4 Exotic Multi-core Platforms

In this thesis we are concerned with exploiting parallelism using SIMD operations such as those used by GPUs. We are interested in GPUs because they serve a large niche market that we expect will continue to be a rich source of fast and cheap parallel computers. However, other niche markets are also served by processors that use multiple cores on a single chip, but without the use of SIMD instructions. For example, the Tilera Corporation TILE64 processor[12] has 64

general-purpose cores on a single chip with a high-bandwidth interconnection between them. One application of this style of many-core computer is for network switches that perform extensive analysis on streams of packets. While such processors likely could be used to accelerate motion planning, their present and likely future inaccessibility renders them unsuitable for motion planning at this time.

There are important applications where vector operations cannot be used, because the problem does not admit of any way to organize data so that they can be accessed together in vector form. Parallel sorting, operations on linked lists, random access into hash tables, and operations on large sparse matrices are a few examples. The Cray XMT[22] architecture uses processors that run 128 threads simultaneously. The 128 threads share a single set of functional units to hide memory latency. The processors are distributed, but share a global memory space, and are connected by a very fast network fabric. The XMT architecture is optimized to maximize throughput when random access to very large data sets is required. Motion planning problems typically do not deal with such large data sets, so this form of parallel computer is not suitable for solving motion planning problems.

2.2.5 Complexity Analysis in Parallel Programming

A full introduction to complexity analysis for parallel programming, that is, the theoretical comparison of running times for parallel algorithms, is beyond the scope of this document. However, we introduce a few important terms. The reader can consult [100] for a deeper introduction.

Sequential algorithms perform their task one step at a time, while parallel algorithms may execute multiple steps at once. The time taken by a sequential algorithm is simply proportional to the number of steps it must take. For sequential computers, the theoretical time taken is usually a good predictor of the time actually taken by an implementation of the algorithm. Since there are so many ways to organize a parallel computer, it can be difficult to choose an appropriate model to predict the performance of a parallel algorithm. The *work-depth* model focuses on the number of operations performed by an algorithm and the dependencies be-

tween these operations, without regard to a parallel machine model. The *work* W of a parallel algorithm is the total number of operations it executes. The *depth* D is the length of the longest sequence of dependencies among the operations. We define $\mathcal{P} = W/D$ as the *parallelism* of the algorithm. For a sequential algorithms $W = D$ so $\mathcal{P} = 1$. The greatest degree of parallelism is achieved when all operations are independent, that is $D = 1$. A parallel algorithm is called *work-efficient* if it performs at most a constant factor more operations than a sequential algorithm to solve any problem instance. For tasks where results must be computed within a short time frame, such as the real-time motion planning problem we are concerned with, a work-efficient algorithm is not necessarily preferred, if a work-inefficient algorithm with a smaller depth is available.

2.3 Summary of Related Work

Several threads run through the motion planning literature: a move from local, reactive planning to global, deliberative planning as available computational power increases, epitomized by the state lattice approach; the improvement of methods to join vehicle states by paths that vehicles can follow precisely, for example with polynomial spirals; the orderly sampling of states and actions within a constrained environment as with the conformal state lattice; and enhanced approaches to grid decomposition of planning state spaces to deal with dependencies between state variables, as with the resolution-equivalent grid approach. Table 2.1 illustrates where this thesis contributes to the literature. No work yet has addressed the area of our contribution: real-time deliberative planning with an accurate vehicle model using a scalable parallel algorithm.

While much has been done, it is evident that more work is necessary before the dream of a motion planner with capabilities equivalent to a competent human driver can be realized. This thesis ties these threads together to create a robust and effective planner for high speed on-road driving applications. We will use the increased computational power made available by GPUs to create a global planner

Representative work	Deliberative	High-dimensional search	High-fidelity motion	Parallel	On-road	Real-time
Werling et al.	✓		✓		✓	✓
Ziegler & Stiller	✓				✓	✓
Hybrid A*(Dolgov et al.)	✓		✓		✓	
4D/RCS	✓				✓	✓
Parallel PRM	✓	✓		✓		
This thesis	✓	✓	✓	✓	✓	✓

Table 2.1: Where this thesis fits into the literature.

that exhaustively analyzes all paths through a road-conformed state lattice with a resolution-equivalent structure to generate precise trajectories in real-time.

Chapter 3

Planning Algorithm

In this chapter we describe the core ideas behind the planner presented in this thesis. We start with a conceptual overview of the planner operation that lays out the main ideas guiding the structure of the search space. Then we specify precisely the form of the paths and trajectories that make up the plans. With those basic structures in hand we describe the graph we construct and search to generate complex sequences of paths, and the search algorithm we use. Then we describe the numerical methods we use to obtain the paths and trajectories in practice.

3.1 Overview

The overall approach of the planner is to sample a variety of trajectories specifying the future location of the vehicle on the road as a function of time, evaluate each trajectory numerically using a cost function, and then pick the best trajectory.

3.1.1 Paths

We start with Figure 3.1, showing how potential paths are sampled, i.e., chosen from among the infinite set of possible paths. At this stage we only specify pose and curvature along the path, but not yet the temporal dimensions, i.e., time, velocity, and acceleration. We select a variety of end poses along the roadway and plan a path ρ from the current vehicle pose to each sampled pose. The paths are constructed to be smooth, with curvature varying continuously as a function of arc length, so that the vehicle can follow them precisely. Depending on the speed of the vehicle, we may use paths that are continuous in curvature up to the second derivative with respect to arc length.

3.1.2 Trajectories

With several sampled paths, we next sample several acceleration profiles, specifying the velocity $v(s)$ along the path from $s = 0$ to $s = s_f$ (see Figure 3.2). That is, for each path, we sample several different ways to drive along the path. In this

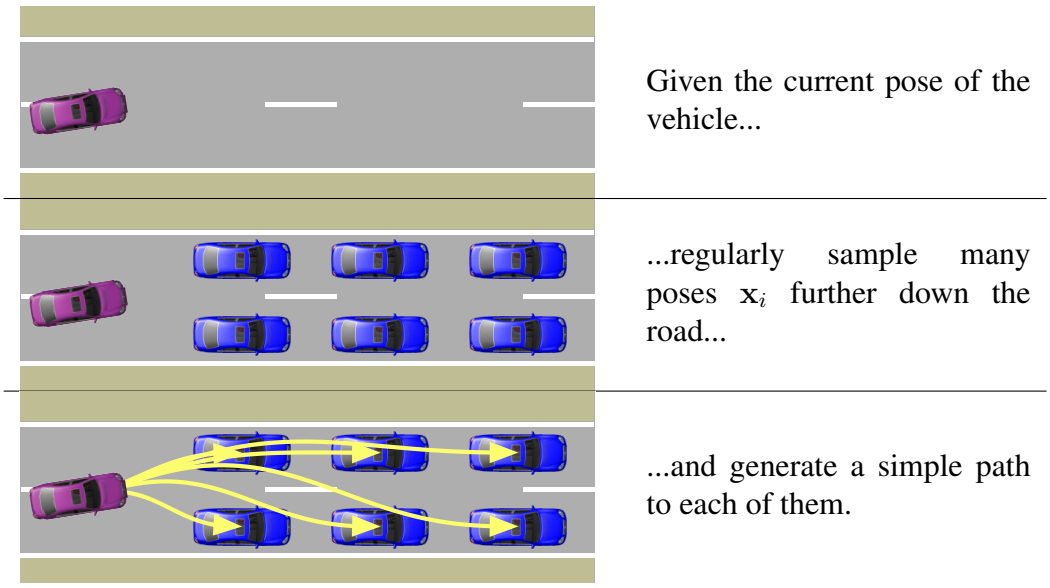


Figure 3.1: Endpoint pose sampling strategy for the planner.

example, we sample three ways: speeding up with a constant acceleration, slowing down with a constant acceleration, or keeping at the same speed, with zero acceleration. The result of each acceleration profile is a trajectory τ that specifies the velocity of the vehicle at each point along a path over the roadway. From these it is easy to calculate derived quantities such as the time, velocity, steering rate, and lateral acceleration of the vehicle at each point along the trajectory, which are used to evaluate its cost.

3.1.3 Trajectory Evaluation

Each trajectory τ defined by a path and acceleration profile fully specifies a possible future for the vehicle. What remains is to decide which trajectory to follow. We do this by evaluating each trajectory τ with a cost function $c(\tau)$ that gives a numerical ranking to each one (Figure 3.3). The cost function scores the estimated *safety* of the trajectory, e.g. by measuring the minimum distance it passes away from obstacles, the *comfort* of the trajectory, e.g. by finding the sum or maximum

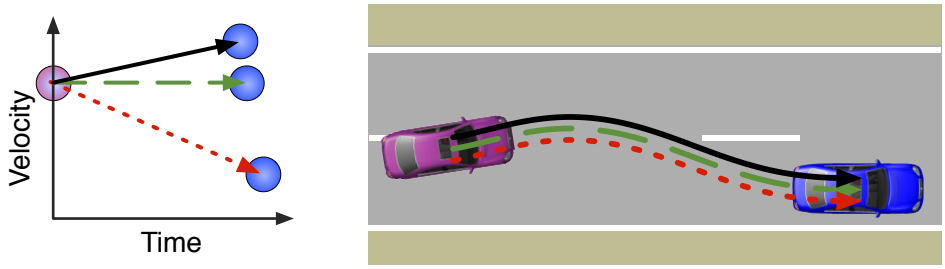


Figure 3.2: Acceleration sampling strategy for the planner. Three acceleration profiles are sampled along a single path - accelerate (black, solid), stay at the same velocity(green, long dash) and decelerate (red, short dash). While each profile takes the vehicle to the same path endpoint, they arrive at different times and with different velocities.

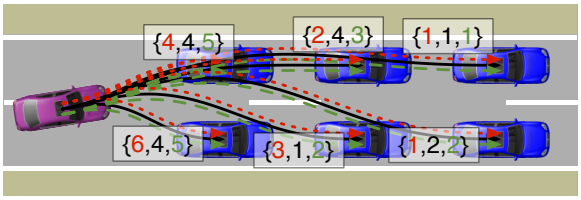


Figure 3.3: Cost function: each path sample depicted in Figure 3.1 is multiplied by the set of acceleration profiles to gives rise to several trajectories τ. Each trajectory τ is evaluated using a cost function $c(\tau)$ to find the desirability of having the vehicle follow that trajectory.

of the lateral acceleration along it, and its conformance with the *behavioral* goals specified at higher levels of the planning hierarchy – e.g. whether it stays within the preferred lane. For a pair of trajectories with the same cost $c(\tau)$, the trajectory that travels further and in less time is more desirable. In the following, $t_f(\tau)$ is the time at which the trajectory τ ends, and $s_f(\tau)$ is the road *station*, or longitudinal distance along the road, reached by the end of the trajectory. The position of the vehicle at the start of a planning cycle is defined to have station $s = 0$. Since the sampled trajectories may travel different distances and end at different times, we need to compensate for these variations and pick the best overall trajectory that minimizes the overall cost function

$$\Omega(\tau) = c(\tau) + \Phi(\tau) \tag{3.1}$$

The first term $c(\tau)$ is a weighted sum of the costs to traverse the trajectory, measuring its safety, comfort, and behavioral aspects. The second term $\Phi(\tau)$ is the final cost assigned for the destination of the trajectory,

$$\Phi(\tau) = \Phi_c(\tau) + \Phi_h(\tau). \tag{3.2}$$

The $\Phi_c(\tau)$ term gives an incremental score to the achievement of the trajectory in terms of time taken and distance traveled, specifically

$$\Phi_c(\tau) = -k_s s_f(\tau) + k_t t_f(\tau), \tag{3.3}$$

a discount for trajectories that go further (k_s term), and a penalty for those that take extra time (k_t term). The $\Phi_h(\tau)$ term penalizes horizons, related to the need to ensure that the plan lasts a minimum amount of time, and to the fact that we would prefer a path that goes to the end of the lattice, as long as it would not require aggressive maneuvering:

$$\Phi_h(\tau) = h_d(s_f(\tau)) + h_t(t_f(\tau)), \tag{3.4}$$

where $h_d(s)$ gives an additional discount for driving further than a given station (we typically set the cut-off d_h simply to be the end of the lattice),

$$h_d(s) = \begin{cases} -k_d & \text{if } s \geq d_h \\ 0 & \text{otherwise,} \end{cases}$$

and $h_t(t)$ levies an infinite penalty for *not* exceeding the time horizon:

$$h_t(t) = \begin{cases} \infty & \text{if } t < t_h \\ 0 & \text{otherwise,} \end{cases}$$

where the time horizon t_h should at least as large as the time required for a panic stop.

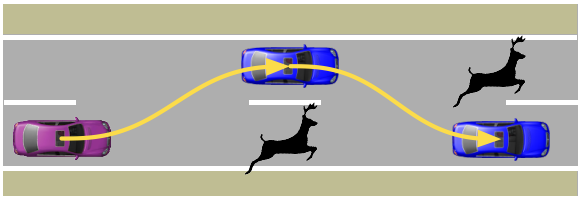


Figure 3.4: A scenario in which a double lane change maneuver is necessary to plan a safe trajectory.

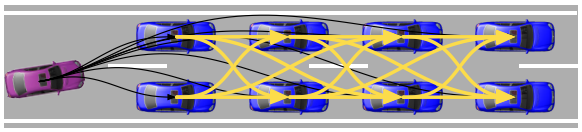


Figure 3.5: Paths joining the sampled poses, in addition to the paths reaching from the current vehicle position to the sampled poses.

3.1.4 Extended Trajectories

With the planning scheme so far, it could turn out that none of the trajectories τ is valid up to the minimum time horizon t_h. Multiple lane changes or other swerving maneuvers may be required to achieve a safe path, as in Figure 3.4. To overcome this problem we need to generate longer and more complex trajectories. First we sample additional paths to join the poses originally sampled in Figure 3.1. Figure 3.5 shows the original sample poses and sample paths from the current vehicle pose, in addition to paths joining the sample poses with each other. Extending the paths is simple, but extending the trajectories requires some care to avoid an exponential blowup in their number. When two or more trajectories end at the same point, at close times, and with a similar velocity, they can be pruned to leave only one. Figure 3.6 illustrates this pruning. With our planner, two such trajectories are "close" when they fall into the same cell in a simple grid defined over the t and v axes. When trajectories are pruned, we keep just one representative from each acceleration profile, retaining the one from each cohort with the minimum-cost trajectory, according to Equation 3.2. That is, out of the trajectories that are all "speeding up", all "slowing down", or all keeping the same speed, and land in the

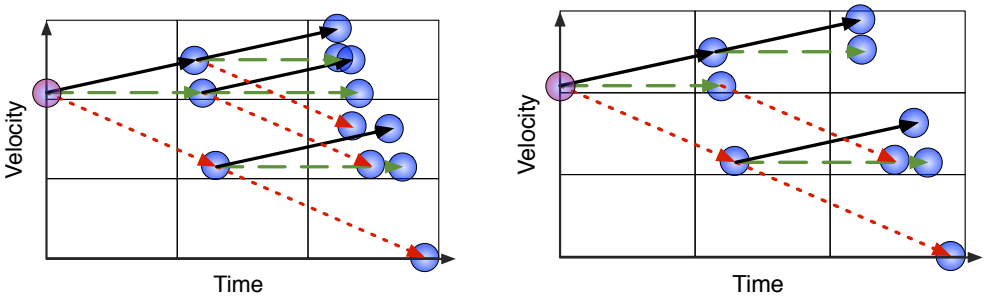


Figure 3.6: Extended trajectories (left) before pruning, and (right) after pruning

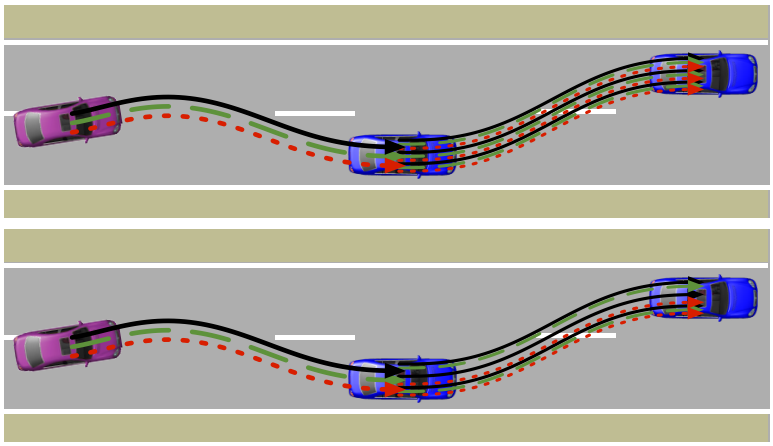


Figure 3.7: Extended trajectories before (above) and after (below) pruning. Compare to Figure 3.6.

same cell, we retain only one of each type. In Section 3.4.5 we will discuss the motivation for keeping one trajectory from each acceleration profile in each cell, rather than just one trajectory per cell. Figure 3.7 renders the effect of this pruning on the effective number of extended trajectories.

3.1.5 The Rest of this Chapter

In the first part of this chapter we have given the reader the overall concept of our planner. With the rest of this chapter we will describe the core planner mechanisms in detail. We start with our model of the vehicle and the road. Then we

describe the structure of the search graph and the cost function we use to evaluate trajectories. Finally we describe the formulation of the paths and trajectories we use and how we identify the path parameters required to drive the vehicle from a starting point to a specified ending point.

3.2 Vehicle Model

We model the state of the vehicle with five variables, $\mathbf{x} = [x\ y\ \theta\ \kappa\ v]$. The vehicle has a location (x, y), a heading θ, a curvature κ, which is the rate of change of θ as a function of distance traveled, and a longitudinal velocity v. The equations for the motion of the vehicle are

$$\begin{aligned} \dot{x} &= v\cos\theta \\ \dot{y} &= v\sin\theta \\ \dot{\theta} &= v\kappa \\ \dot{\kappa} &= \text{input.} \end{aligned} \tag{3.5}$$

These equations express the motion of the vehicle with respect to time. Paths produced by our planner express the motion of the vehicle as a function of curvature κ with respect to distance traveled s, from the start of the path, in which case we express the system as

$$\mathrm{d}x/\mathrm{d}s = \cos\left[\theta(s)\right] \tag{3.6}$$

$$\mathrm{d}y/\mathrm{d}s = \sin\left[\theta(s)\right] \tag{3.7}$$

$$\mathrm{d}\theta/\mathrm{d}s = \kappa(s). \tag{3.8}$$

We often drop velocity from the vehicle state and use just the four elements $\mathbf{x} = [x\ y\ \theta\ \kappa]$. In our simplified model, the vehicle can change its curvature κ at the same rate, no matter the current value of θ. In a real front-wheel steered vehicle the curvature is a nonlinear function of the steering wheel position. Figure 3.8 illustrates with a simple bicycle model, where the curvature $\kappa = (\tan(\omega))/L$ is a

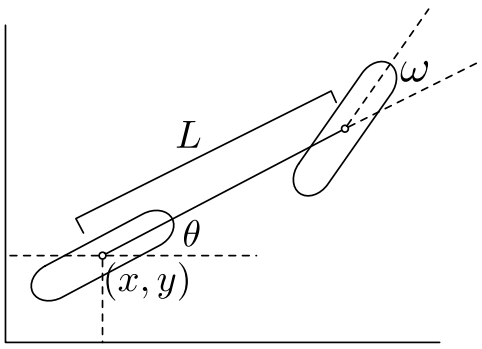


Figure 3.8: The standard bicycle model.

function of steering angle ω and wheel base L. We neglect this discrepancy for planning purposes and use the minimum value of the rate of change in curvature over the domain of ω.

3.2.1 Road Model

The road is defined by its center line, taken as a sampled function

$$r(s) = [r_x(s)\ r_y(s)\ r_\theta(s)\ r_\kappa(s)], \tag{3.9}$$

of arc length s, also known as station when referring to the road structure. We can define a point $p(s, \ell)$ away from the road center at a given lateral offset ℓ, or latitude, from the center line as $p(s, \ell) = [x_r(s, \ell)\ y_r(s, \ell)\ \theta_r(s, \ell)\ \kappa_r(s, \ell)]$ where

$$\begin{aligned} x_r(s, \ell) &= r_x(s) + \ell \cos(r_\theta(s) + \tfrac{\pi}{2}) \\ y_r(s, \ell) &= r_y(s) + \ell \sin(r_\theta(s) + \tfrac{\pi}{2}) \\ \theta_r(s, \ell) &= r_\theta(s) \\ \kappa_r(s, \ell) &= (r_\kappa(s)^{-1} - \ell)^{-1}. \end{aligned} \tag{3.10}$$

The heading θ is a function of station only, and curvature κ increases towards the inside of a turn and decreases towards the outside. We are using a right-handed coordinate system. At the origin facing towards positive x, latitude in-

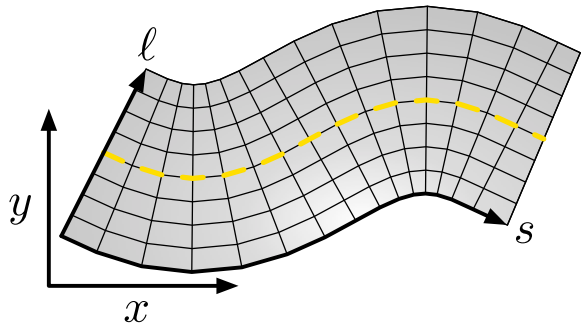


Figure 3.9: The SL coordinate frame laid over the road in X-Y space.

creases along the y-axis, to the left. Each lane of the road can be expressed as a set $\{p(s, \ell_k) : s \in \mathbb{R}^+\}$ for some constant ℓ_k unique to each lane. Here, the curvature κ is the rate of change of heading θ with respect to change in station s.[1] We call the domain of the mapping $p(s, \ell)$ the SL coordinate system.

3.3 Path Model

In this section we describe the family of paths we use to represent motions of the vehicle, how they are represented, and how specific paths are obtained. While the structures we describe here are sufficient to specify the behavior of the search, in Section 3.5 we will revisit these formulations in order to refine their numerical behavior.

A path is defined as a continuous function ρ mapping the interval $[0, 1]$ into a space of robot configurations, e.g. $\mathcal{C} = \{(x, y, \theta, \kappa)\}$:

$$\rho : [0, 1] \rightarrow \mathcal{C}.$$

The starting configuration of the vehicle is $\rho(0) = q_{init} \in \mathcal{C}$, and the configuration at the end of the path is $\rho(1) = q_{goal} \in \mathcal{C}$. We aim to find a path that

[1] This is the reciprocal of the curvature typically used in automotive applications. Automotive applications express curvature as the radius of the circle with the curvature we define here.

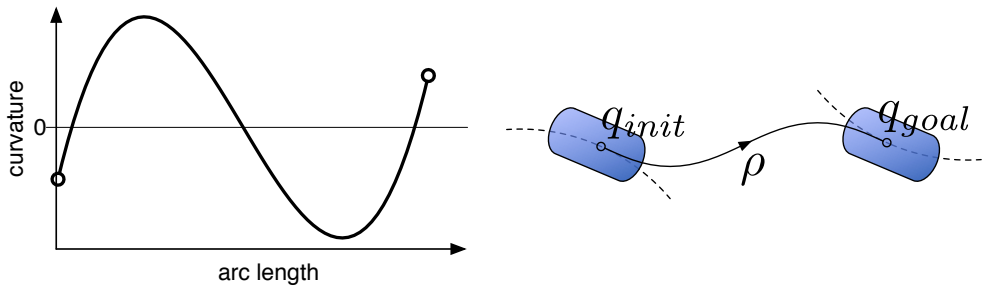


Figure 3.10: A cubic polynomial spiral, rendered (left) as a graph $\kappa(s)$ of curvature as a function of distance traveled, and (right) as the path driven by a vehicle using Equations 3.6–3.8. Dashed arcs show the starting and ending curvatures.

respects relevant constraints, i.e., one that drives the vehicle smoothly between the two configurations. Following Kelly and Nagy[49] and Howard[44], we represent paths as polynomial spirals. A polynomial spiral is a plane curve whose curvature is a polynomial function of its arclength, i.e., of the form $\kappa(s)$. We use either a cubic(third-order) polynomial,

$$\kappa(s) = \kappa_0 + \kappa_1 s + \kappa_2 s^2 + \kappa_3 s^3, \tag{3.11}$$

or a quintic(fifth-order) polynomial,

$$\kappa(s) = \kappa_0 + \kappa_1 s + \kappa_2 s^2 + \kappa_3 s^3 + \kappa_4 s^4 + \kappa_5 s^5, \tag{3.12}$$

at different phases of the plan and at different driving speeds, depending on the need for continuity in the higher-order derivatives of the curvature. Fourth-order polynomials would be a reasonable intermediate between the quick steering realized by the cubic polynomial and the smooth steering by the fifth-order polynomial, but we didn't find it necessary to implement this intermediate level of smoothness for our experiments in this work.

The cubic spline allows us to maintain a continuous curvature, but for reasons we will see in the next section, it can result in a discontinuous steering rate. At low speeds, the resulting path tracking error can be neglected, but at higher speeds the

Figure 3.11: Contrast between a cubic polynomial spiral(upper, dash-dot curve) and quintic polynomial spiral(lower, solid curve) starting and ending at the same configuration. The starting and ending steering curvatures are shown with the dashed arcs. The quintic path satisfies the constraints $\frac{d\kappa}{ds}(0) = 0$ and $\frac{d^2\kappa}{ds^2}(0) = 0$ while the cubic cannot accept these constraints and in effect picks arbitrary values for the two quantities.

discontinuity is too big to ignore. Using the quintic polynomial allows us to maintain continuity of both the curvature rate of change and its derivative. It is established in the literature[82, 5, 106, 105] that fifth-order polynomials lead to smooth robot motions, when used to control motion with respect to time, since they can solve a one-dimensional boundary-value problem of the output with equality constraints up to the second derivative (acceleration). Although we formulate the curvature as a function of distance rather than time, the two are practically equivalent at higher speeds, where velocity changes slowly relative to distance traveled. For this same reason, the fifth-order polynomials when expressed in terms of distance rather than time are not necessary or practical for moving at low speeds, where they produce exaggerated motions. In the example of Figure 3.11, the starting constraints for the quintic path are set to $\frac{d\kappa}{ds}(0) = 0$ and $\frac{d^2\kappa}{ds^2}(0) = 0$. For the cubic path there is no solution given these constraints. We relax these constraints to find a solution, which usually gives us a discontinuous $\frac{d\kappa}{ds}(0)$ and $\frac{d^2\kappa}{ds^2}(0)$ if the vehicle was already in motion. The result is that the quintic path takes longer than the cubic path to start turning back in the desired direction, which leads to a wider swing outwards. Whether a cubic or quintic polynomial is used to generate a path between two particular points depends on the vehicle's speed, and on where it would be used in the overall planning lattice. We discuss the particulars of this choice in Section 3.4.2 and Section 4.5.3.

The polynomial spirals are a convenient formulation that affords quick and reliable convergence using a simple gradient descent search, while adequately

representing the space of paths that are reasonable for a vehicle to follow while driving down a road. We will show in a later section how these parameters can be found quickly. Next, we formulate them precisely.

3.3.1 Polynomial Spirals

We wish to plot a path with continuous curvature leading from a starting vehicle configuration $q_{init} = [x_I\ y_I\ \theta_I\ \kappa_I]$ to an ending configuration $q_{goal} = [x_G\ y_G\ \theta_G\ \kappa_G]$. We can use the cubic polynomial spiral to specify $\kappa(s)$ in Equations 3.6–3.8 as

$$\kappa(s) = \kappa_0 + \kappa_1 s + \kappa_2 s^2 + \kappa_3 s^3. \tag{3.13}$$

Note that since $s = 0$ at the initial state q_{init}, Equation 3.13 implies that $\kappa_0 = \kappa_I$. This leaves the four unknown coefficients $\kappa_{1\ldots3}$ and the total path length s_G to solve for the four elements of q_{goal}.

At high speeds, we wish to ensure the continuity of higher derivatives of κ. In that case, we extend the initial vehicle state $q_{init} = [x_I\ y_I\ \theta_I\ \kappa_I]$ to include the derivatives of κ with respect to arc length s at the initial state, where $s = 0$:

$$\frac{d\kappa}{ds}(s), \frac{d^2\kappa}{ds^2}(s),$$

calling these $d_s\kappa(s)$ and $d_s^2\kappa(s)$ for short, and using $d_s\kappa_I = d_s\kappa(0)$ and $d_s^2\kappa_I = d_s^2\kappa(0)$, so that $q_{init} = [x_I\ y_I\ \theta_I\ \kappa_I\ d_s\kappa_I\ d_s^2\kappa_I]$. We note that just as Equation 3.13 implies that $\kappa_0 = \kappa_I$, it also implies that

$$\frac{\mathrm{d}\kappa(0)}{\mathrm{d}s} = \kappa_1,$$

and

$$\frac{\mathrm{d}^2\kappa(0)}{\mathrm{d}s^2} = 2\kappa_2.$$

Constraining κ_1 and κ_2 in this way leaves only two free input variables, κ_3 and s_G, to satisfy the four independent output variables in q_{goal}. It is simple to extend

Equation 3.13 to a quintic polynomial to gain two more free inputs

$$\kappa(s) = \kappa_0 + \kappa_1 s + \kappa_2 s^2 + \kappa_3 s^3 + \kappa_4 s^4 + \kappa_5 s^5, \tag{3.14}$$

or, expressed using the variables from q_{init},

$$\kappa(s) = \kappa_I + d_s \kappa_I s + d_s^2 \kappa_I s^2 + \kappa_3 s^3 + \kappa_4 s^4 + \kappa_5 s^5, \tag{3.15}$$

leaving us again with four unknowns consisting of $\kappa_{3\ldots5}$ and total path length s_G to find in order to complete the curve starting at q_{init} and ending at q_{goal}.

Now that we have defined the objects and terms we need, such as cubic and quintic path splines and trajectories, we turn to specifying precisely the structure of the planning graph and the algorithm we use to search through it for the best plan.

3.4 Search Graph

The lattice planner conducts its search for a plan within a graph representing discretely sampled vehicle configurations along the road. These discrete samples are combined with continuous regions in the time and velocity dimensions of the vehicle state space, as we showed in the overview of this chapter (Section 3.1). In the following we discuss some observations that guide our design of the search graph, then specify our graph design precisely and define the algorithm we use to search it.

3.4.1 Graph Design Considerations

Since we conduct our search in a dynamic environment, we consider both time and space dimensions. The state lattice[83] we reviewed in Section 2.1.4 is a proven method for systematically searching through static environments; however, naïvely adding time and velocity dimensions can cause an unacceptable blowup

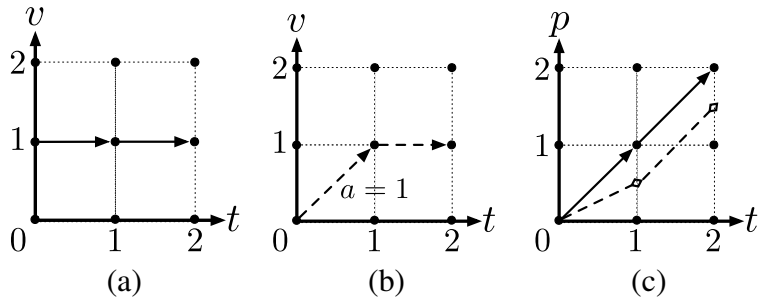


Figure 3.12: Attempted motions through a three-dimensional grid of time t, velocity v, and position p show how the dynamic constraints push an increase in the grid resolution. In (a) a path begins with $v = 1$ and continues for two time steps to arrive at $(t = 2, v = 1)$. In (b) a path accelerates from $(t = 0, v = 0)$ to $(1, 1)$ using an acceleration $a = 1$, then stays at $v = 1$ to reach $(2, 1)$. Part (c) shows the trajectories of these two paths along the position dimension. The path of part (b) does not pass through the grid points.

in the size of the search space. Vertices in the state lattice for static spaces are normally defined by the vehicle state vector $[x\ y\ \theta\ \kappa]$, such that the vehicle traverses paths that satisfy starting and ending boundary constraints coincident with the lattice vertices. The key point is that values of all state variables associated with the lattice vertices are fully specified before the edges are evaluated.

We would like to take this approach with a time-enhanced state vector $[x\, y\, \theta\, \kappa\, t\, v]$, sampling points from this higher-dimensional space to construct a graph we can search for a trajectory. Three factors make this difficult. First is the usual curse of dimensionality. The number of sample points in the lattice increases exponentially with each added dimension, as does the number of edges. That is, if we add a velocity dimension with ν different velocities, then we would multiply the number of vertices in the graph by ν, and the number of paths by ν^2. To see this, suppose we have a graph of n vertices $\{x_i\}$ representing positions along the x axis, connected by e edges $\{(x_i, x_j)\}$. Adding the time dimension to create vertices $\{[x_i v_k]\}$ would increase the number of vertices to $n\nu$. The new edges would be $\{([x_i v_k], [x_j v_l])\}$ with $v_k < v_l$, numbering $O(e\nu^2)$.

The second difficulty is that the edges joining the vertices must satisfy the

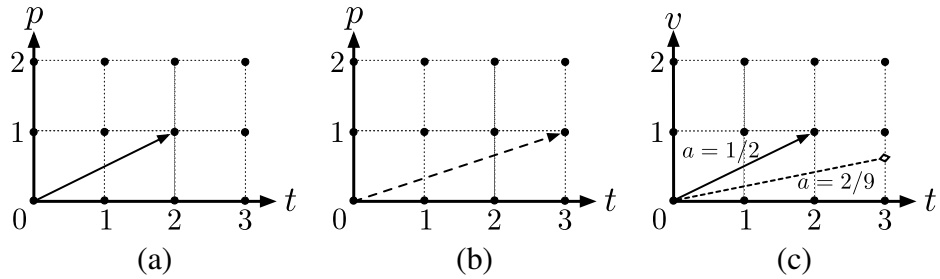


Figure 3.13: An attempted motion through the grid of position p, time t, velocity v. In part (a) and (b) two paths starting with velocity 0 accelerate with constant rate from $(t = 0, p = 0)$ and reach grid points $(2, 1)$ and $(3, 1)$ respectively, and (c) shows the velocities of the paths. The path from (b) does not reach a grid point along the velocity dimension.

kinematic and dynamic constraints of the vehicle model. These constraints necessitate that when an action changes one dimension of a state, other state variables must also change as a side effect. For example, a car cannot shift to the left without also moving forward or backward, nor change its velocity without also moving through time, since acceleration is constrained. If we were to use a standard fixed grid it would impose additional constraints on the values of the grid points that could only be satisfied by adding more grid points. Figures 3.12 and 3.13 illustrate the problem. Part 3.12(b) shows a valid path through the grid points in the (t, v) dimensions which does not pass through the grid points in the (t, p) dimensions. The resolution of the grid would have to be increased in order to represent this path. Note that if there were no constraint on acceleration, i.e., $a = \infty$ were valid, then path 3.12(b) could jump straight to $v = 1$ at time 0, and the t-p grid shown in 3.12(c) would be sufficient. Figure 3.13 shows that the pressure to increase resolution flows the other direction as well. If we plot a course through the t-p plane first, then we would still have to increase resolution along the velocity dimension.

The final difficulty is that no prior placement of points along the velocity dimension will suffice for all circumstances. For example, if the robot needs to follow another vehicle traveling at velocity v, then v must be a grid point along

the velocity dimension. Since v could be any value, we must be able to move grid points on the fly in order to plan in moving traffic.

In summary, the influences we consider in the design of our graph are:

Consideration-1 The curse of dimensionality.

Consideration-2 Constraints on motion require finer discretizations when using a fixed grid.

Consideration-3 No prior placement of fixed grid points fits all scenarios.

In the next section we specify our graph considering these forces. Following Tompkins[99], which we mentioned in Section 2.1.6, we classify each dimension as *independent* or *dependent* and treat each differently in the construction of the graph.

3.4.2 Graph Definition

In this section we define the directed search graph $G = (V, E)$ and in following sections we will describe each part in detail.

The vertices $n \in V$ of the graph are in the form of a five-dimensional tuple $n = [s_h\ \ell_i\ a_j\ [t]_k\ [v]_m]$, where

- s_h is drawn from $\{s\}$, a set of discrete samples of distance along the road, i.e., station.
- ℓ_i is drawn from $\{\ell\}$, discrete samples of lateral distance from the road center line.
- a_j is the index of an element in $\{a\}$, a set of acceleration policies such as one of those described in Section 3.8.1.
- $[t]_k$ is an interval $[t_k, t_{k+1}) \in \mathbb{R}^+$ along the time dimension, together with a distinguished value t_k in the interval, which may change during the search. The intervals $\{[t]_k\}$ are contiguous, and the last interval has $t_{k+1} = \infty$ so that arbitrarily large times can be represented in the graph.

- $[v]_m$ is just like $[t]_k$, an interval $[v_m, v_{m+1}) \in \mathbb{R}^+$ along the velocity dimension, together with a distinguished value v_m which may change during the search. The intervals are also contiguous and the top interval has no upper limit, so that we don't need to specify a prior maximum value on velocity.

We write $s(n)$, $\ell(n)$, etc. to denote the components of the vertex tuple. In Section 3.4.5 we will discuss why the vertex tuple includes the acceleration policy index a_j as a dimension.

The points $\{s\}$ and $\{\ell\}$ are sampled in a regular pattern using a discrete grid (h, i) and a mapping $(s(h), \ell(i))$ from the discrete grid points to station-latitude space using the affine functions

$$\begin{aligned} s(h) &= a_s h \\ \ell(i) &= a_\ell + b_\ell i \end{aligned} \tag{3.16}$$

so that station is monotonic increasing starting from zero and moving right in the station-latitude, or SL coordinate frame, and latitude may be positive (left) or negative (right) with respect to the center line. The warped coordinate grid in Figure 3.9 can be read as an instance of such a mapping. The states generated this way are locally parallel to the road, i.e., a vehicle driving forward from such a state and maintaining its steering curvature would continue along the road indefinitely, assuming the road were circular. Nothing prevents us from expanding the graph to include increments of heading and curvature away from parallel as e.g. $(s(h), \ell(i), \theta(j), \kappa(k))$, but we did not encounter a need for this to satisfy the performance requirements of on-road driving. This choice implies that all plans appear as a sequence of lateral swerves interspersed with brief periods where the vehicle is not moving laterally. The actual behavior is made smoother by frequent replanning.

Given a road center line $r(s)$ as in Equation 3.9 and using Equation 3.10, the elements $[s, \ell]$ of each vertex define a complete configuration

$$[x_r(s, \ell)\ y_r(s, \ell)\ \theta_r(s, \ell)\ \kappa_r(s, \ell)]$$

in the road model described in Section 3.2.1.

There is an additional vertex $n_0 \in V$ representing the current state of the robot. It has the form

$$n_0 = [x_0\ y_0\ \theta_0\ \kappa_0\ v_0\ d_s\kappa(0)\ d_s^2\kappa(0)], \tag{3.17}$$

giving the robot's current configuration, velocity, and steering rate derivatives. Without loss of generality we assume $t = 0$ at the start state.

The edges $e \in E$ of the graph are of the form $e = (n_i, n_j)$, each joining two vertices $n_i = [s_i\ \ell_i\ a_i\ [t]_i\ [v]_i]$, and $n_j = [s_j\ \ell_j\ a_j\ [t]_j\ [v]_j]$. The edge is defined by a trajectory $\tau(e) = (\rho_{ij}, a_j)$ composed of a cubic path spline ρ_{ij} of the type described in Section 3.5.1, and an acceleration profile a_j from Section 3.8.1. The path ρ_{ij} connects the start point

$$[x_r(s_i, \ell_i)\ y_r(s_i, \ell_i)\ \theta_r(s_i, \ell_i)\ \kappa_r(s_i, \ell_i)],$$

and end point

$$[x_r(s_j, \ell_j)\ y_r(s_j, \ell_j)\ \theta_r(s_j, \ell_j)\ \kappa_r(s_j, \ell_j)].$$

We can group the vertices into SL-equivalence classes that all have the same value for s and ℓ,

$$n_i \overset{s\ell}{\sim} n_j \iff s(n_i) = s(n_j) \text{ and } \ell(n_i) = \ell(n_j).$$

Two edges $e_1 = (n_i, n_j)$, $e_2 = (n_k, n_m)$ use the same path when $n_i \overset{s\ell}{\sim} n_k$ and $n_j \overset{s\ell}{\sim} n_m$. We also say $e_1 \overset{s\ell}{\sim} e_2$. We constrain $s_j > s_i$ always, so that the graph only contains paths driving forwards. Driving backwards is done at low speeds and in parking lots and so is outside the scope of this work.

For edges (n_0, n_j) proceeding from the start vertex, the start point for the cubic spline is $[x_0\ y_0\ \theta_0\ \kappa_0]$ from Equation 3.17. As discussed in Section 3.3, a quintic spline may also be used from the start state, using $d_s\kappa(0)$ as $d_s\kappa_I$ and $d_s^2\kappa(0)$ as $d_s^2\kappa_I$ in Equations 3.30–3.35. Quintic splines cannot be used for other edges in the graph since these derivatives are not uniquely specified by the other values in the vertices. To do this the vertices would have to be extended with dimensions

$[d_s\kappa]$ and $[d_s^2\kappa]$ for the curvature rate derivatives in the manner of $[t]$ and $[v]$. By joining a quintic spline from the start state to cubic splines at the subsequent edges, we create discontinuities in the plan's curvature rate, but these discontinuities are rarely encountered since the plan is regenerated frequently.

Referring to Section 3.8.1 the acceleration profile $a_j(s, t_0, v_0)$ uses $t_0 = t_i$, the distinguished point of $[t]_i$, or $t_0 = 0$ for the start state, and $v_0 = v_i$, the distinguished point of $[v]_i$. At the end of the trajectory, the final time t_G and final velocity v_G must fall into their respective intervals, i.e., $t_G \in [t]_j$ and $v_G \in [v]_j$. In practice, the edges of the graph are constructed as the search for a plan progresses, and the t_j, v_j are discovered. The distinguished points in the intervals are assigned to be the t_G or v_G of an incident edge as they are discovered.

When two edges $e_1 = (n_i, n_j)$, $e_2 = (n_k, n_m)$ have the same second vertex, i.e., $n_j = n_m$, then in general $t_G(n_j) \neq t_G(n_m)$. However, the intervals are the same, $[t]_j = [t]_m$, but there is only one distinguished point in the interval. One of the edges is in effect a dead end since any edge (n_j, n_p) proceeding onwards must use the distinguished point from n_j to calculate the acceleration profile. Therefore, once the graph is constructed there is in effect only one edge incident on each vertex. The same argument applies to the distinguished point in the velocity interval. This procedure reveals that the s, ℓ, and a dimensions are the *independent* dimensions, i.e., they are selected independently, and the t and v are the *dependent* dimensions, i.e., their values depend on the choices of s, ℓ, a. This simultaneoulsy resolves **Consideration-2** and **Consideration-3**.

In the next section we discuss how the winning edge is selected, i.e., which edge's ending t and v are selected to become the distinguished points of their respective intervals.

3.4.3 Graph Construction

In the previous section we said that the graph is not completely specified before the search begins. The actual edges of the graph are realized from a set of potential edges as the search proceeds. Out of multiple edges from the potential set that

arrive at the same vertex, only one is selected to become part of the graph. The selection criteria are based on the cost to reach the vertex via the edges. For each vertex $n \in V$ we maintain a lowest known cost $g(n)$ to reach it.

A candidate edge $\tilde{e}_{ik} = (n_i, \tilde{n}_2^k)$ has tentative values for the distinguished points in n_2. We use $\tilde{n}_2^k$ to denote the vertex n_2 with tentative values for the distinguished points drawn from the candidate edge $\tilde{e}_{ik}$. Among a set of candidate edges $\{\tilde{e}_{ik}\}$, $\tilde{e}_{ik} = (n_i, \tilde{n}_2^k)$ potentially connecting n_i and n_2, the edge e_{i2} that is retained is the one which minimizes a cost

$$e_{i2} \leftarrow \arg\min_{\tilde{e}_k} g(n_i) + c(\tau(\tilde{e}_k)) + \Phi_c(\tau(\tilde{e}_k)), \tag{3.18}$$

where g is the lowest cost to reach n_i from the start node, c is the cost function of the trajectory spline, and Φ_c the incremental portion of the final cost function defined in Equation 3.3. All other edges are discarded. The distinguished values in n_2 are set from e_{i2}. The cost $g(n_2)$ is then set,

$$g(n_2) \leftarrow g(n_i) + c(e_{i2}), \tag{3.19}$$

as the lowest cost known to reach n_2. We don't include Φ_c in this expression since $\Phi_c(n_2)$ is an increment of the final cost earned up through n_2, not the cost to reach n_2 from the start node. The expressions in Equations 3.18 and 3.19 look like the familiar f-value

$$f(n) = g(n) + h(n),$$

from the standard A* algorithm[40]. Whereas A* operates on a graph that is already (though usually implicitly) specified, we can see Equation 3.18 as guiding the construction of our graph by selecting the candidate edges in such a way that the f-value of each vertex is minimal. The A* search algorithm itself can be readily adapted to address the problem of a graph with path-dependent edge costs, which is another way of looking at our problem. Note that $\Phi(n)$ is not a good choice for the A* heuristic function $h(n)$. To use $\Phi(n)$ as part of an admissible

heuristic we would need to add an (under)estimate of the sum $\sum c(\tau)$ for the succeeding trajectories.

3.4.4 Graph Search

Since driving is a real-time application where the vehicle cannot stop and deliberate on its next action, we are concerned with minimizing the worst-case time to find a solution path through the graph.

Using the A* algorithm, the worst case is that all vertices in the graph would have to be expanded in order to find the optimal path. If all vertices may be expanded, then it is better to use an algorithm that does this as rapidly as possible, dispensing with the priority queue and other overhead that accompanies a typical A* implementation.

The worst-case outcome for A* is rendered all the more likely considering that admissible heuristic functions are in general difficult to invent. While effective functions have been found for parking lot and rover applications (cf. [25], [62]), it seems a daunting task to find an effective heuristic for the much more complex cost function that we employ with the high-speed driving problem. We describe this cost function in the next chapter.

An anytime algorithm would not improve our worst-case scenario. Although some (low-speed) driving applications have used anytime algorithms[62] to obtain a suboptimal path that can be refined as it is followed, even these algorithms must find a feasible path before it can be refined to an optimal path, and in the worst case these algorithms also expand all vertices before any path is found.

We propose a dynamic programming algorithm that simultaneously builds and traverses the graph. The majority of the search is carried out in the course of the graph construction process, where edges are evaluated and selected, and vertex locations are decided. The only constraint on the order in which edges are evaluated is that all candidate edges leading into a vertex n must be evaluated before edges leading out of n can be evaluated, so that the distinguished t and v points are determined. Figure 3.14 outlines the algorithm. For each vertex n we store

function SEARCH-DP
$\forall n : g(n) \leftarrow \infty$
for each station $s_h \in \{s\}$
$\forall n$ s.t. $s(n) = s_h : \phi(n) \leftarrow \infty$
for each vertex $n = [s_h\ \ell_i\ a_j\ [t]_k\ [v]_m]$ at station s_h
if $g(n) \neq \infty$
Form the vector
$\hat{\mathbf{x}}_\mathbf{n} = [x(n), y(n), \theta(n), \kappa(n), t(n), v(n)]$
for each outgoing edge $\tilde{e} = (n, n')$
Form the trajectory $\tau(\rho(e), a(e))$
if $g(n) + \Phi_c(\tau) < \phi(n')$
$\phi(n') \leftarrow g(n) + \Phi_c(\tau)$
$g(n') \leftarrow g(n) + c(\tau)$
$t(n') \leftarrow t_f(\tau)$
$v(n') \leftarrow v_f(\tau)$
$\text{incoming}(n') \leftarrow n$ // *backtrace info*
end if // $\hat{c}(n)$
end for // a
end if // $\neq \infty$
end for // *vertex*
end for // *station*

Figure 3.14: The dynamic programming search algorithm for the search graph.

$g(n)$, the trajectory cost to reach n, and $\phi(n)$, the trajectory cost to reach n plus the final cost to end the plan at that vertex. The former is used when extending edges from n to later vertices, and the latter is used when deciding which edge incoming to n should be chosen. To more easily reconstruct the path once $g(n)$ has been calculated for all vertices, the $\mathrm{incoming}$ table stores the winning edge incoming to each vertex.

Once the dynamic programming search algorithm is complete and all vertices have their cost-to-come $g(n)$ set, a final vertex n_f is added to V. One edge is added from each node $n \in V$ to n_f, with cost $\Phi(n)$. The overall plan becomes the lowest-cost sequence of edges connecting n_0 to n_f. In practice we don't add n_f. Rather, we pick the vertex n^* with minimum cost-to-come plus final cost as the final vertex in the plan,

$$n^* = \arg\min_n g(n) + \Phi(n). \tag{3.20}$$

Since an edge (n_i, n_j) always has $s(n_i) < s(n_j)$, we can evaluate all edges (n_i, n_j) with the same $s(n_i)$ in parallel. We will describe our parallel algorithm for the GPU in Chapter 5.

Finally, a note about the optimality of our graph search. In the previous section we mentioned that our graph has path-dependent edge costs. We use *path* in the sense of a sequence of edges through the graph, rather than a spline path ρ. Two different sequences of edges may both start from vertex n_0 and reach vertex n_k with the same cost but slightly different values for the distinguished t- and v-points. Since they have the same cost, the choice of which incoming path to use to set the distinguished points of n_k is arbitrary. The costs of the subsequent edges from n_k may vary based on this choice. For example, if the first path is followed to reach n_k there may be no subsequent path to the goal should the t-value it brings to n_k lead to an unavoidable collision. The second path with the different t-value with which it arrives at n_k might have subsequently avoided the collision and reach the goal, but the distinguished points of n_k would already be set based on the first path, so the path to the goal via the second path could never be found.

Our algorithm is optimal within the limitations imposed by the need to discard all but the lowest-cost edge incoming to each vertex.

In practice this suboptimality is not a problem as long as the t and v regions are not too large, so that the endpoints of all of the plans that fall into the same region are effectively equivalent for planning purposes. We have encountered a situation where paths that undergo hard braking for long periods are not considered, because the final cost Φ_c retains edges that travel the same distance in less time. In the next section we will see how including acceleration as a dimension in the state space mitigates this problem.

Now that we have described the structure of the graph and how it is searched, we next justify our design decisions in terms of the design factors discussed in Section 3.4.1.

3.4.5 Why include acceleration in the vertex tuple?

In this section we address **Consideration-1** by discussing why the acceleration profile should be included as a dimension in the vertex tuple, i.e., why should the form of the vertex be

$$n = [s_h \; \ell_i \; a_j \; [t]_k \; [v]_m],$$

and not simply

$$n = [s_h \; \ell_i \; [t]_k \; [v]_m].$$

Consider two edges $e_1 = (n_1, n_2)$ and $e_2 = (n_1, n_4)$ where $n_2 \overset{s\ell}{\sim} n_4$, so they start at the same graph vertex and use the same path ρ, but have differing acceleration profiles. If their final times and velocities $t_1 = t_f(\tau(e_1))$, $v_1 = v_f(\tau(e_1))$ and t_2, v_2 land in the same $[t_i, t_{i+1})$ and $[v_j, v_{j+1})$ (Figure 3.15), and acceleration were not included as a dimension, then only one of the accelerations would be represented in the result. Rather than simply refining the discretization of the intervals $[t]$ and $[v]$ along their dimensions by the amount required to overcome this problem, we found that we could include acceleration in the graph and thereby represent a greater diversity of trajectories. We also found that this more consistently yielded

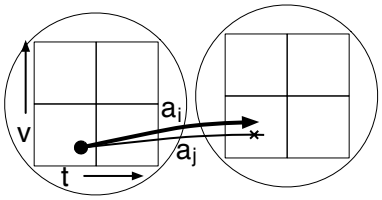


Figure 3.15: If acceleration is not a dimension in the lattice, multiple trajectories proceeding from the same starting lattice vertex and differing only in their acceleration profiles may interfere by ending in the same vertex.

final trajectories that use the same acceleration profile across consecutive edges, which is beneficial for passenger comfort.

In the previous section we mentioned that plans that undergo hard braking for long periods are not considered, because Φ_c favors and instead retains plans that travel further in less time. Using acceleration as a dimension in the state space mitigates this problem by ensuring that hard braking is always considered. Plans that apply hard braking over two consecutive edges may still be starved, however. This is an issue we must consider in future work.

Now that we have precisely defined the planning graph, we need to revisit the cubic and quintic path formulations. In the rest of this chapter we delve into the details of the path formulation we use in the planner, the algorithm we use to solve the boundary value problem connecting vehicle states by the paths, and the acceleration profiles we use to turn paths into trajectories. In the next section we look at an alternate formulation of the polynomial needed to improve numerical stability.

3.5 Stable Path Model

In Section 3.3 we described the polynomial spiral path model. In this section we change the formulation of the polynomial in order to improve numerical stability, so that in Section 3.6 we can show how to solve for the polynomial coefficients in order to satisfy the endpoint constraints.

3.5.1 Stable Cubic Paths

In practical applications, the formulation given in Equation 3.13 of Section 3.3.1 introduces round-off errors due to the large discrepancies in magnitude between κ_1 and κ_3. The coefficient κ_1 of the s term is typically much larger than the coefficient κ_3 of the s^3 term, for instance, and this problem worsens as the required path length s_G increases. It becomes especially problematic later when we need to compute and invert the Jacobian of the spiral endpoint with respect to the coefficients.

To improve numerical accuracy, the parameters to be solved for should be of a similar scale. To this end, we introduce new parameters $\mathbf{p} = [p_{0\ldots3}, s_G]$, again following [44], and compute the polynomial coefficients indirectly as

$$\kappa(s) = a(\mathbf{p}) + b(\mathbf{p})s + c(\mathbf{p})s^2 + d(\mathbf{p})s^3 \tag{3.21}$$

where the parameters are constrained to be equal to the path curvature at equally spaced points along the path:

$$\begin{aligned} \kappa(0) &= p_0 && (3.22)\\ \kappa\left(\frac{s_G}{3}\right) &= p_1 && (3.23)\\ \kappa\left(\frac{2s_G}{3}\right) &= p_2 && (3.24)\\ \kappa(s_G) &= p_3, && (3.25) \end{aligned}$$

thus ensuring that the parameters are of comparable scale since the curvature is bounded by the steering limits of the vehicle. The exception is s_G. The discrepancy in scale between s_G and the $\{p_i\}$ is up to approximately three orders of magnitude, for example for a curvature at one-third the length along the path of $p_1 = 0.1$ and path length of $s_G = 100$ meters. However, we have found in practice that this magnitude of discrepancy is not a problem. Values of s_G are usually on the order of 20. The coefficients from Equation 3.21 can be solved using the

constraints of Equations 3.22–3.25, giving

$$
\begin{aligned}
a(\mathbf{p}) &= p_0 && (3.26)\\
b(\mathbf{p}) &= -\frac{11p_0 - 18p_1 + 9p_2 - 2p_3}{2s_G} && (3.27)\\
c(\mathbf{p}) &= \frac{9(2p_0 - 5p_1 + 4p_2 - p_3)}{2(s_G)^2} && (3.28)\\
d(\mathbf{p}) &= -\frac{9(p_0 - 3p_1 + 3p_2 - p_3)}{2(s_G)^3}. && (3.29)
\end{aligned}
$$

Since paths are intended to join a specified starting vehicle state q_{init} with an ending vehicle state q_{goal}, we note that Equation 3.26 implies $p_0 = \kappa_I$ and Equation 3.25 implies that $p_3 = \kappa_G$. This leaves just three unknowns $\hat{\mathbf{p}} = [p_1\ p_2\ s_G]$, one fewer than we had in Section 3.3.1.

Stable Quintic Paths

We can extend the stable cubic polynomial presented in [44] to formulate a stable quintic polynomial. As with the cubic polynomial, we define new parameters $\mathbf{p} = [p_{0\ldots5}, s_G]$ to optimize in place of the original $\{\kappa_i\}$ of the naive formulation in Equation 3.14. The constraints defining the p_i in this case are similar to the cubic constraints (Equations 3.22–3.25), but use the additional elements in the start state:

$$
\begin{aligned}
\kappa(0) &= p_0 = \kappa_I && (3.30)\\
d_s\kappa(0) &= p_1 = d_s\kappa_I && (3.31)\\
d_s^2\kappa(0) &= p_2 = d_s^2\kappa_I && (3.32)\\
\kappa\left(\frac{s_G}{3}\right) &= p_3 && (3.33)\\
\kappa\left(\frac{2s_G}{3}\right) &= p_4 && (3.34)\\
\kappa(s_G) &= p_5 = \kappa_G. && (3.35)
\end{aligned}
$$

To use the parameters, we restate Equation 3.14 using functions of the parameters $\mathbf{p}$:

$$\kappa(s) = a(\mathbf{p}) + b(\mathbf{p})s + c(\mathbf{p})s^2 + d(\mathbf{p})s^3 + e(\mathbf{p})s^4 + f(\mathbf{p})s^5.$$

Solving for the constraints of Equations 3.30–3.35, we obtain

$$a(\mathbf{p}) = p_0 \tag{3.36}$$

$$b(\mathbf{p}) = p_1 \tag{3.37}$$

$$c(\mathbf{p}) = \frac{p_2}{2} \tag{3.38}$$

$$d(\mathbf{p}) = -\frac{575p_0 - 648p_3 + 81p_4 - 8p_5 + 170p_1 s_G + 22p_2(s_G)^2}{8(s_G)^3} \tag{3.39}$$

$$e(\mathbf{p}) = \frac{9\left(37p_0 - 45p_3 + 9p_4 - p_5 + 10p_1 s_G + p_2(s_G)^2\right)}{2(s_G)^4} \tag{3.40}$$

$$f(\mathbf{p}) = -\frac{9\left(85p_0 - 108p_3 + 27p_4 - 4p_5 + 22p_1 s_G + 2p_2(s_G)^2\right)}{8(s_G)^5}. \tag{3.41}$$

As with the stable cubic formulation of the previous section, this leaves just three unknowns, this time $\hat{\mathbf{p}} = [p_3\ p_4\ s_G]$.

In this and the previous section we derived a more numerically-stable parameterization of the polynomial spirals. That is, the unknowns are closer in magnitude, which will make it easier to solve for them using numerical methods. We now turn to describing that method, a gradient-descent search.

3.6 Path Optimization

In the following we describe how we find the parameters $\mathbf{p}$ for each of the cubic polynomial spirals and quintic polynomial spirals.

3.6.1 Path Optimization, Cubic Case

We wish to find the unknown parameters $\hat{\mathbf{p}} = \{p_1, p_2, s_G\}$ extending a cubic polynomial spiral between two given states, a starting position $q_{init} = [x_I\ y_I\ \theta_I\ \kappa_I]$ and

desired ending position $q_{goal} = [x_G\ y_G\ \theta_G\ \kappa_G]$. For simplicity and better numerical behavior, we can transform q_{init} and q_{goal} together to place q_{init} at the origin, to derive a new start point $q_{init} = [0\ 0\ 0\ \kappa_I]$ and new end point q_{goal}. Given a candidate $\mathbf{p}$, we can calculate the endpoint as

$$\begin{aligned} x_{\mathbf{p}}(s) &= \int_0^s \cos\left[\theta_{\mathbf{p}}(s)\right] \mathrm{d}s \\ y_{\mathbf{p}}(s) &= \int_0^s \sin\left[\theta_{\mathbf{p}}(s)\right] \mathrm{d}s \\ \theta_{\mathbf{p}}(s) &= a(\mathbf{p})s + b(\mathbf{p})s^2/2 + c(\mathbf{p})s^3/3 + d(\mathbf{p})s^4/4 \\ \kappa_{\mathbf{p}}(s) &= a(\mathbf{p}) + b(\mathbf{p})s + c(\mathbf{p})s^2 + d(\mathbf{p})s^3, \end{aligned} \tag{3.42}$$

denoting the dependency on $\mathbf{p}$ by the subscript. That is, for a given parameter vector $\mathbf{p}$, we write the x-coordinate at a distance s along the path described by $\mathbf{p}$ as $x_{\mathbf{p}}(s)$, and similarly for the other variables.

Gradient Descent Algorithm

We use a shooting method to solve for the unknown parameters $\hat{\mathbf{p}}$ that will satisfy the endpoint constraints. Extending the notation introduced in the previous section, we use $x_{\mathbf{p}}(s_G)$ to refer to the endpoint of the path implied by the s_G element in $\mathbf{p}$ itself, i.e., $s_G = s_G(\mathbf{p})$. The configuration at the endpoint of the path defined by $\mathbf{p}$ is $q^{\mathbf{p}}_{init}(s_G) = [x_{\mathbf{p}}(s_G)\ y_{\mathbf{p}}(s_G)\ \theta_{\mathbf{p}}(s_G)\ \kappa_{\mathbf{p}}(s_G)]$. Treating it as a function of the parameters $\mathbf{p}$, we can calculate the Jacobian of the endpoint state vector with respect to the unknown parameters,

$$\mathbf{J}_{\hat{\mathbf{p}}}(q^{\mathbf{p}}_{init}(s_G)) = \begin{bmatrix} \frac{dx_{\mathbf{p}}}{dp_1}(s_G) & \frac{dx_{\mathbf{p}}}{dp_2}(s_G) & \frac{dx_{\mathbf{p}}}{ds_G}(s_G) \\ \frac{dy_{\mathbf{p}}}{dp_1}(s_G) & \frac{dy_{\mathbf{p}}}{dp_2}(s_G) & \frac{dy_{\mathbf{p}}}{ds_G}(s_G) \\ \frac{d\theta_{\mathbf{p}}}{dp_1}(s_G) & \frac{d\theta_{\mathbf{p}}}{dp_2}(s_G) & \frac{d\theta_{\mathbf{p}}}{ds_G}(s_G) \end{bmatrix}.$$

We don't need the derivative for κ since $\kappa_{\mathbf{p}}(s_G) = p_3$, i.e., the value at the endpoint is not a function of the unknowns. We can then solve the problem using Newton's method.

We wish to find the $\mathbf{p}$ that makes $q^{\mathbf{p}}_{init}(s_G) = q_{goal}$. We assume a reasonable

initial guess for the unknowns $\hat{\mathbf{p}}$ (we'll discuss how to select the initial guess in Section 5.1.4), and use the Jacobian $\mathbf{J}_{\hat{\mathbf{p}}}(q^{\mathbf{p}}_{init}(s_G))$ to iteratively refine $\hat{\mathbf{p}}$ into a better estimate $\hat{\mathbf{p}}'$ by following the gradient of $q_{goal} - q^{\mathbf{p}}_{init}(s_G)$ towards zero.

$$\begin{aligned} \mathbf{J} &\leftarrow \mathbf{J}_{\hat{\mathbf{p}}}(q^{\mathbf{p}}_{init}(s_G)) \\ \Delta\mathbf{q} &\leftarrow q_{goal} - q^{\mathbf{p}}_{init}(s_G) \\ \Delta\hat{\mathbf{p}} &\leftarrow \mathbf{J}^{-1}\Delta\mathbf{q} \\ \hat{\mathbf{p}}' &\leftarrow \hat{\mathbf{p}} + \Delta\hat{\mathbf{p}}. \end{aligned} \tag{3.43}$$

We can repeat this procedure substituting $\hat{\mathbf{p}}'$ for $\hat{\mathbf{p}}$ until $\Delta\mathbf{q}$ is sufficiently small for our purposes, or a maximum number of iterations has been reached, indicating that the initial guess was not close enough to the true value of $\mathbf{p}$ to ensure convergence.

Calculating the Jacobian

Calculating the Jacobian is non-trivial. Both $\theta_{\mathbf{p}}(s)$ and $\kappa_{\mathbf{p}}(s)$ can be evaluated in closed form, but the elements $x_{\mathbf{p}}(s_G)$ and $y_{\mathbf{p}}(s_G)$ are a special case of the generalized Fresnel integrals (as per [49]) and have no closed form solution. To solve for these, we integrate numerically. Since the cubic polynomial is not stiff, and few function evaluations are necessary to achieve sufficient error bounds, we can use a simple quadrature method to evaluate the integrals. We use the composite Simpson's rule:

$$\int_a^b f(x)\,dx \approx \frac{h}{3}\left[f(x_0) + 2\sum_{j=1}^{n/2-1} f(x_{2j}) + 4\sum_{j=1}^{n/2} f(x_{2j-1}) + f(x_n)\right] \tag{3.44}$$

where $x_j = a + jh$ for $j = 0, 1, \ldots, n-1, n$ with $h = (b-a)/n$. Other authors computed the Jacobian numerically using central differencing[44]. By using small integration steps, that method can handle inequality constraints such as saturation of the steering angle at points along the path, but we assume that in highway driving scenarios kinematic limits would not be reached until well after dynamic limits have been surpassed. Therefore, since we assume no inequality constraints

need be enforced during the integration, we can use fewer integration steps than previous approaches to calculate the endpoint. We use $n = 8$ in Equation 3.44, and symbolically differentiate the entire expression to obtain a function closely approximating the true derivative, which allows us to calculate the Jacobian very efficiently, compared to calculating the Jacobian numerically using a differencing method. Generating a single C++ function containing the entire calculation also allows us to optimize the code by, for example, factoring out common subexpressions between values of j in Equation 3.44. Our implementation of our method to calculate the Jacobian is approximately 6 times faster than our implementation that follows [44] using a forward differencing method with the same number of Euler integration steps. In practice, one must use more Euler integration steps to achieve the same degree of accuracy.

3.6.2 Path Optimization, Quintic Case

The quintic polynomial optimization is the same as the cubic case, with the necessary substitutions:

$$\begin{aligned} x_{\mathbf{p}}(s) &= \int_0^s \cos\left[\theta_{\mathbf{p}}(s)\right] \mathrm{d}s \\ y_{\mathbf{p}}(s) &= \int_0^s \sin\left[\theta_{\mathbf{p}}(s)\right] \mathrm{d}s \\ \theta_{\mathbf{p}}(s) &= a(\mathbf{p})s + b(\mathbf{p})s^2/2 + c(\mathbf{p})s^3/3 + d(\mathbf{p})s^4/4 + e(\mathbf{p})s^5/5 + f(\mathbf{p})s^6/6 \\ \kappa_{\mathbf{p}}(s) &= a(\mathbf{p}) + b(\mathbf{p})s + c(\mathbf{p})s^2 + d(\mathbf{p})s^3 + e(\mathbf{p})s^4 + f(\mathbf{p})s^5, \end{aligned} \tag{3.45}$$

using Equations 3.36–3.41. Out of the parameters $\{p_{0\ldots5}, s_G\}$ defining the quintic polynomial spiral between the transformed start configuration $q_{init} = [0\ 0\ 0\ \kappa_I]$ and the end configuration $q_{goal} = [x_G\ y_G\ \theta_G\ \kappa_G]$, only p_3, p_4, s_G are unknown, with the others given in Equations 3.30, 3.31, 3.32, 3.35. For the rest of the optimization, we follow the method of the previous section - using a symbolic differentiation of an expression derived from Simpson's method to calculate the Jacobian quickly, and employing the procedure of Equation 3.43 to find the unknown parameters.

3.7 Path Integration

We noted in the previous section that we don't need to calculate points along a path in order to find its endpoint. However, we do need points along the path in order to evaluate it in the search. We use a trapezoidal integration method for $x(s)$ and $y(s)$. It has sufficient accuracy for our purposes and allows convenient incremental evaluation of the integrals at several regularly-spaced points along the path. Recall the trapezoidal integration formula is

$$\int_a^b f(x)dx \approx \frac{b-a}{2N}\left[f(x_0) + 2f(x_1) + \cdots + 2f(x_{N-1}) + f(x_N)\right], \qquad (3.46)$$

with

$$x_k = a + k\frac{b-a}{N}, \text{for } k = 0, 1, \ldots, N.$$

We want to calculate the integral at the intermediate steps:

$$\int_a^{x_k} f(x)dx, \text{for } k = 0, 1, \ldots, N.$$

This can be done incrementally, using

$$\begin{aligned} s_k &\leftarrow \tfrac{ks_G}{N} \\ \theta_k &\leftarrow a(\mathbf{p})s_k + b(\mathbf{p})s_k^2/2 + c(\mathbf{p})s_k^3/3 + d(\mathbf{p})s_k^4/4 + e(\mathbf{p})s_k^5/5 + f(\mathbf{p})s_k^6/6, \end{aligned} \qquad (3.47)$$

for all $k = 0, 1, \ldots, N$, initializing

$$x_0, y_0, \Delta x_0, \Delta y_0 \leftarrow 0,$$

and then for $k > 0$, iteratively calculating

$$\begin{aligned} \Delta x_k &\leftarrow \Delta x_{k-1}\tfrac{k-1}{k} + \tfrac{\cos\theta_k + \cos\theta_{k-1}}{2k} \text{ for } k > 0 \\ \Delta y_k &\leftarrow \Delta y_{k-1}\tfrac{k-1}{k} + \tfrac{\sin\theta_k + \sin\theta_{k-1}}{2k} \text{ for } k > 0 \\ x_k &\leftarrow s_k \Delta x_k \\ y_k &\leftarrow s_k \Delta y_k. \end{aligned} \tag{3.48}$$

This gives us an evenly-spaced sequence of points $(s_i, x_i, y_i, \theta_i, \kappa_i)$ along the path which can be evaluated individually through the cost function and summed to give a cost for the overall path. Next we look at how we augment paths with times and velocities to create alternative trajectories.

3.8 From Paths to Trajectories

We define a trajectory τ as an extension of a path ρ traveling through configuration space $\mathcal{C}$ into a more general state space manifold $\mathcal{M}$ that includes time. The definition of τ is similar to that of ρ in Section 3.3:

$$\tau : [0, 1] \rightarrow \mathcal{M},$$

where $\mathcal{M} = \{(x, y, \theta, \kappa, t, v)\}$. A trajectory is produced from a path by applying an acceleration profile $a(s)$, given $v(0)$, the velocity at the start of the path segment, and $t(0)$, the starting time.

As we have said, the path is defined by a polynomial function of curvature κ as a function of arc length s. In order to evaluate the goodness of a trajectory in terms of such quantities as its curvature rate, and ensure smooth steering across replan cycles, we need the time-derivatives of curvature, which we can obtain using the chain rule once we have $v(s) = \dot{s}$:

$$\dot{\kappa}^{\text{cubic}}(s) = b(\mathbf{p})\dot{s} + 2c(\mathbf{p})s\dot{s} + 3d(\mathbf{p})s^2\dot{s}, \tag{3.49}$$

and

$$\dot{\kappa}^{\text{quintic}}(s) = b(\mathbf{p})\dot{s} + 2c(\mathbf{p})s\dot{s} + 3d(\mathbf{p})s^2\dot{s} + 4e(\mathbf{p})s^3\dot{s} + 5f(\mathbf{p})s^4\dot{s}. \quad (3.50)$$

3.8.1 Accelerations

We use two types of acceleration profile $a(s, t_0, v_0)$ to generate trajectories given $t_0 = t(0)$ and $v_0 = v(0)$:

- A constant acceleration $a(s) = a$
- An acceleration produced by a PD-controller for distance-keeping to a vehicle in front

Constant Accelerations

For the constant-acceleration case, we can determine a in one of two ways. The first way is to select a constant a directly, for example using $a = 0$ to travel at a constant speed, or using a large deceleration like $-5ms^{-2}$ to represent the vehicle's performance limits. The second way is to select a desired final velocity $v(s_G) = v_G$ to be reached by the end of the path, such as a local speed limit, so that the value for a is

$$a = \frac{v_G^2 - v_0^2}{2s_G}. \quad (3.51)$$

Given a constant acceleration a, the time and velocity reached at each point along the path is

$$\begin{aligned} v_a(s) &= \sqrt{2as + v_0^2} \\ t_a(s) &= \frac{2s}{v_0 + v_a(s).} \end{aligned} \quad (3.52)$$

An implementation note - numerical round-off errors can lead to undesirable results. If the acceleration a is intended to yield a target velocity at the end of the path of $v_a(s_f) = 0$, then the radicand in Equation 3.52 could be a small negative number at the end of the path. For this reason we also perform the test

if radicand $> -\epsilon$ **then**
  $v = \sqrt{\max(0, \text{radicand})}$
else
  no solution
end

In the case of no solution, we set v to a small positive value and continue at that constant rate until the end of the path. We will have more to say on this in Chapter 4.5.2.

Distance-keeping accelerations

For the PD-controlled distance-keeping acceleration, we need the obstacle-detection code to identify a vehicle in the same lane ahead of the robot. If there is such a vehicle, the obstacle-detection code must predict its future motion. We assume a simple model identifying the current road station $S_o(0)$ at time zero ("now", the start of the planning cycle) and a constant velocity v_o, so that the road station $S_o(t)$ of the obstacle at time t is

$$S_o(t) = S_o(0) + v_o.$$

We could easily imagine a more complex predictive model with a changing velocity. Note that here we use S for station to distinguish from the arc length s of the robot's path. We set parameters to construct a desired following gap $g_{\text{des}}(t)$ using the current velocity of the robot

$$g_{\text{des}}(t) = k_a^g + k_b^g v(t), \tag{3.53}$$

so that as the robot drives faster the desired gap becomes larger. The actual gap at time t we name $g(t)$. The desired acceleration for distance-keeping mode is

$$a_{\text{dk}}(t) = k_1^g(g(t) - g_{\text{des}}(t)) + k_2^g(v_o - v(t)), \tag{3.54}$$

the first term closing the gap and the second term equalizing the velocity. The desired acceleration $a_{\mathrm{dk}}(t)$ is further clamped within desired acceleration and deceleration bounds to keep it within the bounds of what the vehicle is capable of and to ensure comfort. For example, we do not want distance-keeping mode to involve hard acceleration, though hard deceleration is acceptable when necessary. The time $t_a(s)$ and velocity $v_a(s)$ at points along the path are computed numerically using a simple Euler integration based on Equation 3.52.

3.9 Summary

In this chapter we gave an overview of our planner, and described the structure of its constituent paths and trajectories, with the path optimization approach that makes the planner possible. The planner rests on the ability to solve a boundary-value problem in the spatial dimensions. We then looked at the problems caused by trying to form a search grid in a high-dimensional space when the system is subject to dynamic constraints. We proposed that allowing the grid points to move along some dimensions could resolve these difficulties. Considering this, we showed our scheme for concatenating the constituent trajectories in order to build sufficiently long and complex overall trajectories to be confident that it can generate safe and reasonable plans in challenging circumstances. We described our approach to evaluating component trajectories and how we decide which overall plan to accept, given that driving to the end of the planning horizon is not always the best choice.

The main idea of our planner is that it is necessary to evaluate a dense and diverse set of candidate plans in order to find a safe one, let alone one that maximizes a complex set of desired behavior preferences. The cost function and the search graph structure together determine the space of behaviors the planner can generate. Our aim has been to devise a search space that is likely to contain a desirable plan no matter the circumstances, so that the only remaining question is what the cost function should be. In the next chapter we will look at how to

manipulate the cost functions $c(\tau)$ and $\Phi(n)$ in order to select the one plan from the many considered that most closely achieves the behaviors we desire for a safe and comfortable ride. We will find that the search space structure mostly satisfies our aims, but in the course of tuning the cost function we will bump against limitations imposed by the search space. This will lead us to consider improvements to the search space for future work.

Chapter 4

Cost Functions and Behavior Tuning

The philosophy of our planner is to examine many candidate plans while excluding as little of the search space *a priori* as possible. We use a cost function to rank the plans numerically, and pick the lowest-cost plan to execute.

We use an iterative process to specify the cost function. First, we identify physical quantities that should be penalized or discounted, such as lateral acceleration, vehicle speed, and proximity to obstacles. Second, we compose terms that measure these quantities and assign weights to them. Third, we run experiments and observe the resulting behavior. The experiments may clarify the behavioral tradeoffs implicit in the weightings, or point to new quantities that should be regulated. We repeat these three steps until we obtain satisfactory behavior in the scenarios of interest.

In this chapter we aim to show that our approach of using a cost function to rank a large number of plans is viable. That is, we demonstrate a cost function that is inexpensive to compute while generating reasonable behaviors. The amount of engineering effort required to compose and tune an ideal cost function is outside the scope of this thesis, but we aim to convince the reader that the ideal cost function is within reach.

In the rest of this chapter we present the principles that guide our search for the cost function, the terms of our proposed cost function, and the reasoning behind each of them.

4.1 Relentless Optimization

In designing our cost function we must contend with the phenomenon of *relentless optimization*, whereby a planner tends to select "brittle" plans that fail in the presence of errors in modeling or execution.

The trajectories of interest in autonomous driving tend towards an extreme, where it is incrementally better to increase or decrease some quantity of interest, right up to a catastrophic limit. A small error in the vehicle or environmental model, or in the execution of a plan, can push the system to the other side of such

a limit. For example, it is sensible to select trajectories that drive faster, all other things being equal, as long as the speed limit is not exceeded. When approaching a corner, trajectories that start braking later will tend to score higher on time. The braking force that can be applied by the vehicle has some limit beyond which the vehicle will skid. If speed is the only objective, the best trajectory will therefore be one which applies braking force just below that limit, and starts braking as late as possible. The slightest error in perception or control execution may therefore lead to a catastrophic failure. The cost function must be designed to promote both safety and efficiency.

The main idea in mitigating the effect of relentless optimization is to identify the terms in the cost function that lead towards a precipice, and add more expensive regions leading up to the precipice that cancel out the benefits of approaching the limit. For example, we can surround lethal static obstacles with higher-cost areas that increase sharply in cost closer to the vehicle. This keeps the planner from approaching them too closely unless absolutely necessary. We can also take perception noise into account by expanding obstacles that are further away from the vehicle, to keep the planner from committing to a plan that will turn out to be infeasible as the robot draws nearer to the object.

When a limit is inadvertently crossed, we would like the cost function to promote a rapid return back to safety without making the situation worse by causing a panic stop. For example, if the vehicle's lateral acceleration is already over the limit when the planning cycle begins, we would like the cost function to guide the planner smoothly back below it, rather than, for example, simply slamming on the brakes. Where static or moving obstacles are involved, we would like a cost function that allows the vehicle to collide with obstacles, should that be inevitable, while keeping it away in all other circumstances. When a collision is inevitable, we would like to actively plan to minimize damage, rather than simply braking as hard as possible and hoping for the best. However, once we permit collisions, we must take care that the planner only uses them as a last resort. This is an open problem which we leave for future work. For this work, we try to avoid using

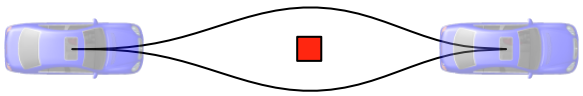


Figure 4.1: Hysteresis motivating scenario: two equally good but different paths around an obstacle can cause indecision when replanning.

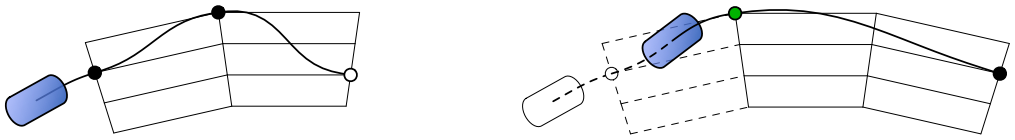
lethal, or infinite values in the cost function which can leave the planner with no feasible plan if the vehicle should happen to stray into a lethal-cost region. We do use infinite costs for obstacles in our experiments, since for the traffic conditions and the safety protocols we use in our testing on the robot, we would rather the vehicle brake hard when it can't find a plan rather than reason that it should collide with something.

4.2 Hysteresis and Stability

Slight changes in the world model used by the planner can result in highly dissimilar plans having the lowest cost in successive planning cycles. Figure 4.1 shows an example scenario with two dissimilar ways to get around an obstacle, though they would have a similar cost. Sensor noise and discretization artifacts can cause the left and right plans to appear best in successive planning cycles. It is better to pick one path at the beginning and stick with it. Rapidly alternating between commands to steer left and right may prematurely wear vehicle actuators and alarm passengers. Hysteresis is a robust way of filtering spurious changes to the plan that is independent of the vehicle modeling and control. We apply a cost discount to those plans that are most similar to the previous plan. Expressing the plan as a sequence of graph vertices n:

$$n = [n_0, n_1, \ldots n_f],$$

we propose a scheme for promoting stability through hysteresis on the notion that it is most important to maintain consistent plans at the start of the path. We do



(a) Plan cycle 1 (b) Plan cycle 2

Figure 4.2: Hysteresis applied to vertices in the world-fixed road lattice. The black vertices in 4.2a mark the first two lattice points in the plan. By the second planning cycle (4.2b), the vehicle has passed the first station in the lattice. The lattice is extended along the road. The green vertex receives a discount due to its inclusion in the previous plan and is chosen for the plan, but the plan is different after that point since only the first two points in the first cycle received a discount. The first two vertices in the new plan are marked to receive a discount in the third cycle.

two things to effect hysteresis.

First, we ensure that similar trajectories are actually available in the space searched, and second, we apply a cost bonus to similar trajectories. To achieve the first point, we fix the lattice to the world frame rather than letting it slide with the vehicle. This ensures that after the vehicle moves, the same path segment from the vehicle to the lattice point p_0 is still available, though without the initial portion that just was traversed.

Second, we decrement the cost $c(\tau(e))$ of any graph edge e that terminates at a vertex with the same (s, ℓ) index as either of the first two vertices (after the start state n_0) of the previous plan, with an additional discount if the a coordinate also matches. The indices are adjusted if the lattice just advanced an increment for the later planning cycle. Figure 4.2 illustrates. We assign the discount only to the first two lattice points along the plan because it is most important to filter spurious changes to the first part of the plan.

The hysteresis cost we just discussed was calculated across plans, i.e., so that the cost of a plan is affected by the previous plan. We also promote stability by penalizing spurious acceleration changes within a single plan. An edge

$$e = ([s_1\ \ell_1\ a_1 \ldots], [s_2\ \ell_2\ a_2 \ldots])$$

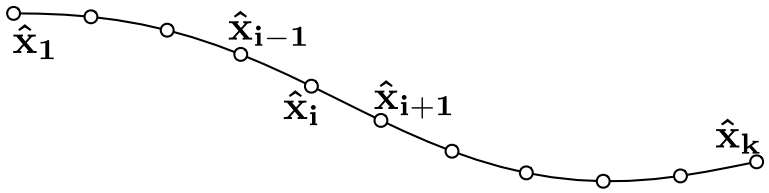


Figure 4.3: Sample points along a path. The cost function $c(\hat{\mathbf{x}})$ is evaluated at these discrete sample points.

receives an additional penalty $k_{\text{accel}}^{\text{change}}$ when $a_1 \neq a_2$.

4.3 Trajectory Costs

For each sampled trajectory corresponding to a single edge in the plan, the cost of having the vehicle traverse the trajectory is computed using a function $c(\hat{\mathbf{x}})$ evaluated over a set of k samples $\{\hat{\mathbf{x}}_i\}_{i=1}^{k}$ along the trajectory, as in Figure 4.3. We use $c(\tau)$ for the cost over the whole trajectory, or $c(\hat{\mathbf{x}})$ for a single sample along the trajectory, depending on the context. The extended state vector used to evaluate the cost is $\hat{\mathbf{x}} = [x\ y\ \theta\ \kappa\ t\ v\ a\ \dot{\kappa}]$. Terms used in the cost function include:

$c_{\textbf{static}}(\hat{\mathbf{x}})$ Static cost term based on the state of the vehicle without regard to quantities dependent on time. This is broken into further cost terms:

- $c_{\textbf{static}}^{\textbf{obs}}(x, y)$ Static obstacle cost term, used to avoid static hazards on the road, such as potholes, stopped cars, and debris detected by the perception system.
- $c_{\textbf{static}}^{\textbf{lane}}(x, y)$ Static road potential cost terms, used to promote lane centering and control the magnitude and direction of deviations during emergency maneuvers.
- $c_{\textbf{static}}^{\textbf{curv}}(\kappa)$ Static curvature cost term, used to exclude paths that exceed the steering limits of the vehicle. Does not account for additional effective limitations such as slip angle, which is dependent on velocity.

$c_{\textbf{dynamic}}(\hat{\mathbf{x}})$ Dynamic cost term based on vehicle quantities that depend on time. This is broken into further cost terms:

- $c^{\textbf{obs}}_{\textbf{dynamic}}(x, y, t)$ Dynamic obstacle cost term based on the location of the vehicle and the time at which it reaches the location. Used to avoid other traffic on the road.
- $c^{\textbf{accel}}_{\textbf{dynamic}}(a)$ Penalize hard acceleration and decelerations, to stay within the passenger comfort zone.
- $c^{\textbf{speed}}_{\textbf{dynamic}}(v)$ Penalty based on longitudinal speed of the vehicle. Used to ensure that the vehicle stays below the speed limit when possible.
- $c^{\textbf{lataccel}}_{\textbf{dynamic}}(\kappa, v)$ Penalty based on the absolute lateral acceleration of the vehicle, $|\kappa|v^2$, used to avoid uncomfortably high or dangerous lateral accelerations.
- $c^{\textbf{curv}}_{\textbf{dynamic}}(\dot{\kappa})$ Penalize high steering wheel turning rates.

The total cost of traversing each trajectory τ is the sum of the static terms and dynamic terms applied to its underlying path samples $\{\hat{\mathbf{x}}_i\}_{i=1}^{k}$. The sum of these costs is scaled by the path length and divided by the number of samples, so that the total cost for a path or trajectory depends on its arc length and not on the number of samples used,

$$c(\{\hat{\mathbf{x}}_i\}_{i=1}^{k}) = \frac{s_f(\tau)}{k} \sum_{i=1}^{k} \left(c_{\text{dynamic}}(\hat{\mathbf{x}}_i) + c_{\text{static}}(\hat{\mathbf{x}}_i) \right). \tag{4.1}$$

In the following sections we describe these cost terms in more detail.

4.4 Static Costs

Paths are evaluated by sampling points $\mathbf{x} = (x, y, \theta, \kappa)$ along their length and evaluating the cost function $c_{\text{static}}(\mathbf{x})$. Two cost functions are prepared from external perception data: the first, $c^{\text{lane}}_{\text{static}}$, assigns a cost to points based on their position

with respect to the desired lane of travel, and the second, $c^{\text{obs}}_{\text{static}}$, gives a cost for their proximity to obstacles. In the next section we define $c^{\text{lane}}_{\text{static}}$.

4.4.1 Lane Centering

We want the vehicle to drive in the center of the selected lane, but obstacle avoidance and other emergency maneuvers may require the vehicle to depart from the lane center. We balance these demands using a cost function $c^{\text{lane}}_{\text{static}}$ based on the lateral deviation of a point (x, y) from the center of the desired travel lane. We assume that descriptions of the available lanes on the road are provided, either by prior knowledge obtained from a map, or on-line from a road-detecting perception system.

The premise of our lane centering cost function is that it should cost little for the robot to make minor corrections within its own lane, but it should tend back towards the center. If it needs to swerve outside its desired travel lane, it should prefer lanes in the same travel direction over opposing lanes. Finally, the cost should be monotonic increasing away from the center of the desired travel lane, to avoid getting stuck in a distant local minimum after an evasive maneuver.

Our scheme is depicted in Figure 4.4. The tuning parameters we use to shape the cost potential are:

c^{des}_a Constant term for desired traveling lane. This should be lower than all other lanes, and we typically set it to zero.

c^{trav}_a Constant term for other traveling lanes, i.e., those in the same direction as the desired lane. This gives a base cost to each alternate lane. We set it high enough to mark a boundary between minor in-lane lateral deviations and more consequential maneuvers.

c^{opp}_a Constant term for opposing lanes. This gives a very high base cost to enter opposing lanes, so that it is only done in emergencies.

c_b^{trav} Linear term for desired *and* traveling lanes. This creates a 'V'-shaped cost centered on the desired lane, to encourage staying in the center of the lane. We use a small value for c_b^{trav} to allow the vehicle to depart from the lane center for long periods, allowing smooth deviations around obstacles. The same slope is extended over other traveling lanes, encouraging the vehicle to stay close to the desired lane even when it must make a large departure.

c_b^{opp} Linear term for opposing lanes. We make it more expensive the further the vehicle departs into opposing lanes.

Other values extracted from the road shape and used to form the cost potential are

ℓ^{des} Latitude of the center of the desired traveling lane.

ℓ^{div} Latitude of the line dividing the traveling lanes from the opposing lanes.

ℓ^{w} Width of lanes, assumed to be the same for all lanes.

For each point (s, ℓ) on the road, the cost potential is computed as

$$c(\ell) = \begin{cases} c_a^{\text{opp}} + c_b^{\text{opp}} |\ell - \ell^{\text{div}}| & \text{if } \ell > \ell^{\text{div}} \\ c_a^{\text{des}} + c_b^{\text{trav}} |\ell - \ell^{\text{des}}| & \text{if } \ell \leq \ell^{\text{div}} \text{ and } |\ell - \ell^{\text{des}}| < \ell^w/2 \\ c_a^{\text{trav}} + c_b^{\text{trav}} |\ell - \ell^{\text{des}}| & \text{if } \ell \leq \ell^{\text{div}} \text{ and } |\ell - \ell^{\text{des}}| \geq \ell^w/2 \end{cases} \tag{4.2}$$

Finally, there is a high penalty in the shoulders on either side of the road. The lane centering cost map is defined in (s, ℓ) coordinates, but we need to evaluate the path using its (x, y) coordinates. We use a map

$$\text{XYSL} : (x, y) \to (s, \ell)$$

which performs this transformation. The implementation of this map is described further in Section 5.3.3.

Further experimentation in lane costs might consider increasing the cost of opposing lanes as a function of station, to discourage committing the vehicle to a plan that ends in an opposing lane.

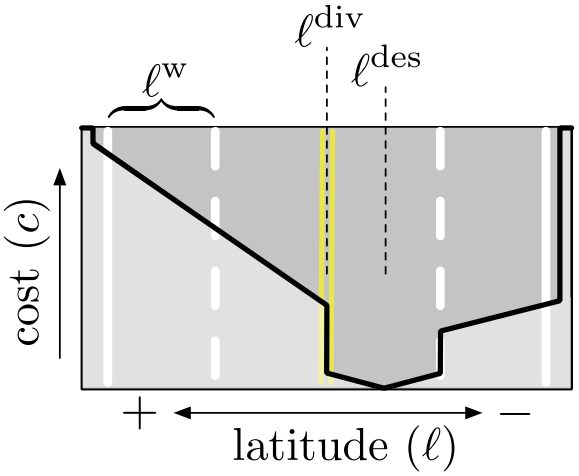


Figure 4.4: A lane cost potential function that encourages driving in the center of the desired lane while allowing necessary deviations.

4.4.2 Static Obstacles

Each (x, y) point in the world is assigned a cost $c^{\text{obs}}_{\text{static}}(x, y)$ based on its proximity to static obstacles identified by the perception system. Like $c^{\text{lane}}_{\text{static}}$, this cost is assigned without regard to the velocity of the vehicle, or the time. Note that the orientation θ of the vehicle is not considered in this cost function - we assume that the vehicle will remain essentially parallel to the road during highway driving so that its orientation will not appreciably affect the cost we would like to assign. We will justify this assumption later in this section.

For simplicity, we define the cost as a potential function over the plane which can take on just three distinct values:

lethal A path causing the vehicle to intersect with an obstacle is assigned the maximum cost, $c^{\text{obs}}_{\text{static}}(x, y) = \infty$. No path will be accepted if it passes through such a region.

high A finite but expensive cost to penalize paths that pass near an obstacle but don't contact it.

zero Regions a safe distance away from any obstacle are assigned a value of zero.

We use only three values because in practice the potential function is implemented as a coarse grid on the order of 30 cm. Since 1 meter, for example, is a safe lateral distance from other vehicles, this leaves little space for a smoother gradient of costs in the map. We said in Section 4.1 that lethal costs should be used sparingly lest they leave the planner with no way to recover from an error. We address this later in the section.

The nonzero-cost (*lethal* and *high*) regions are drawn around each obstacle according to its distance from the vehicle. Figure 4.5 illustrates. Given $\hat{s}$, the distance from the vehicle's current position to the obstacle, the obstacle is expanded longitudinally by $a_0^s + a_1^s \hat{s}$ and laterally by $a_0^\ell + a_1^\ell \hat{s}$ to obtain the lethal-cost region. For the constants a_0^s and a_0^ℓ we use the half-length and half-width of the vehicle, so that this region is locally a $\mathcal{C}$-space expansion of the vehicle. The linear coefficients a_1^s and a_1^ℓ cause this $\mathcal{C}$-space-expanded region to grow with the distance from the vehicle. This mitigates the tendency towards relentless optimization, as we discussed in Section 4.1. If we did not so expand the object, an error in obstacle perception or plan execution could cause the planner to commit to a plan that later turns out not to be feasible as the vehicle draws closer to the obstacle. The *high*-cost region is constructed similarly, but with larger coefficients b_i^j. When obstacle regions overlap we take the maximum value. It would be worthwhile to investigate an enhancement such as $\max(a_0^s + a_1^s \hat{s}, a_2^s)$ and similarly for the other variables to limit the expansion of distant obstacles beyond a certain size. Preliminary experiments suggest that a nonlinear growth rate would be desirable, where the dilation increases rapidly with distance at first and then levels off. For our high-speed experiments we used a simulation environment where we assumed a high enough perception accuracy that the a_1, b_1 terms were relatively small. For robot experiments we use lower speeds and object distances which lie within the rapid-growth dilation regime.

As we said at the start of this section, $c^{\text{obs}}_{\text{static}}(x, y)$ does not consider the orientation θ. In highway driving, even during rapid lateral maneuvers the vehicle remains almost parallel to the road. To mitigate the small error that does exist, we

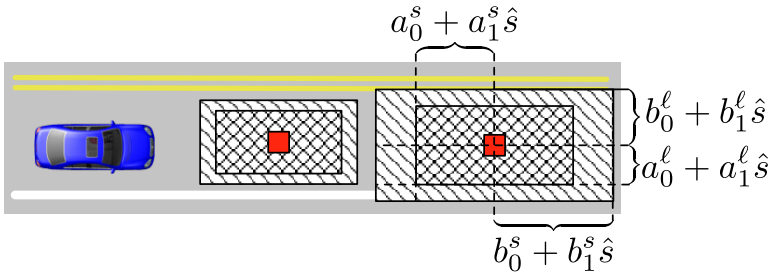


Figure 4.5: Construction of the cost function potential field generated by static obstacles. The red squares are the obstacles themselves. The cross-hatched regions have infinite cost. The hatched regions have a high, finite cost. The boundaries of the regions are defined by first-order polynomials, growing with $\hat{s}$, the distance from the vehicle to the obstacle in station.

dilate the obstacles by the size of the error. Assuming a vehicle with the dimensions of Boss[101], i.e., 5.6 meters long by 2.3 meters wide, a six-degree deviation from parallel to the road, which is a typical deviation during a rapid lane change, yields a 30-cm lateral displacement of the corner of the vehicle. We dilate the obstacles by approximately this much. Larger deviations in orientation can occur at lower speeds. For example, a 24-degree deviation is possible while changing lanes from behind a stopped vehicle, which translates to a 1-meter lateral deviation in the corner of the vehicle. This additional error is mitigated by the fact that our planner considers entire paths. The kinematics of the vehicle ensure that while our collision checking scheme may wrongly estimate a single pose to not be in collision, other poses in the path will be estimated correctly, as Figure 4.6 shows. We have not fully quantified this effect, but in empirical tests we did not observe any unanticipated collisions.

The lane potential cost function $c_{\text{static}}^{\text{lane}}$ and the static obstacle potential $c_{\text{static}}^{\text{obs}}$ are combined to compute the cost of a path ρ. To evaluate a trajectory, we need to complete c_{dynamic}, the cost for the extended state vector that includes time and derived quantities, $\hat{\mathbf{x}} = [x\ y\ \theta\ \kappa\ t\ v\ a\ \dot{\kappa}]$.

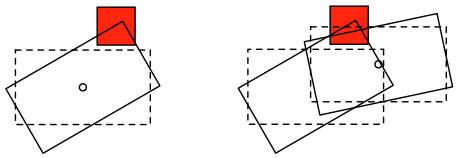


Figure 4.6: The kinematics of the vehicle mitigate the error due to ignoring θ in the occupancy grid. The left figure shows the real vehicle (solid line) colliding with an obstacle, while the road-parallel vehicle actually used for the collision checking would not (dashed line). The right figure shows that even though the vehicle is turning away from the obstacle, its kinematics ensure that the road-parallel version collides with the obstacle.

4.5 Dynamic Costs

Once the paths ρ are evaluated, we need to evaluate the trajectories τ. We use several terms to evaluate aspects of the vehicle's behavior. The first is a potential function $c^{\text{obs}}_{\text{dynamic}}(x, y, t)$ to penalize getting too close to moving obstacles.

4.5.1 Dynamic Obstacles

We create a cost potential function for dynamic obstacles similar to the one for static obstacles. We add the time dimension t to create a cost function $c^{\text{obs}}_{\text{dynamic}}(x, y, t)$ giving the cost for the vehicle's center point to occupy a given point in space at the given time. As with $c^{\text{obs}}_{\text{static}}$, and for the same reasons, we neglect changes in θ.

For each moving obstacle we predict its location and velocity as a function of time up to some horizon. At a given plane of the (x, y) space for a fixed value of t, the obstacles are treated much like static obstacles. Each (x, y) point is assigned a cost based on its proximity to the obstacles. As with $c^{\text{obs}}_{\text{static}}$, we assign to each point one of three values: *lethal*, for intersections of the robot with the obstacle, *high* for being in close proximity, a linear cost *follow*(Δs) for points behind the vehicle, or zero, for further distances.

Just like static obstacles, the moving obstacles are dilated along the station and latitude dimensions. We define a function that gives the amount each obstacle

sample should be dilated along the station direction to create a lethal cost region,

$$D^S_{\text{lethal}}(t, v) = a^s_0 + a^s_1 tv.$$

Obstacles are dilated more when they are further away in time, thus diminishing the impact that noisy sensor readings could have in tricking the robot into committing to brittle maneuvers near other vehicles several seconds into the future. Faster-moving obstacles are dilated even more, since a small error in estimating the velocity of an approaching vehicle is more consequential at higher speeds.

We define another function for the lateral expansion of the lethal-cost region, similar to $D^S_{\text{lethal}}(t)$ but without the factor of v,

$$D^L_{\text{lethal}}(t) = a^\ell_0 + a^\ell_1 t.$$

For the lateral expansion we can dispense with the dependency on velocity since lateral motion is very slow compared to longitudinal motion, diminishing the margin of safety required in case the perception system's estimate of a vehicle's lateral velocity is in error.

We define functions $D^S_{\text{high}}(t, v)$ and $D^L_{\text{high}}(t)$ similarly to $D^S_{\text{lethal}}(t, v)$ and $D^L_{\text{lethal}}(t)$. Regions of $c^{\text{obs}}_{\text{dynamic}}$ with the boundaries defined by these functions are assigned either *high* or *lethal*.

To make the planner follow the vehicles ahead at a safe distance, we can create a region of graduated cost behind each vehicle. The length f_{len} of the follow-cost region is proportional to the velocity of the vehicle, so that we follow faster vehicles at a larger distance. The constant $k^{\text{size}}_{\text{follow}}$ is simply the following gap in terms of time, multiplied by the vehicle velocity to get the following gap in terms of distance.

$$f_{\text{len}} = k^{\text{size}}_{\text{follow}} v_o.$$

The cost of the region starts at 0 at a distance f_{len} and increases linearly closer to the vehicle, using a factor $k^{\text{slope}}_{\text{follow}}$. We use the lateral dilation of the obstacle $D_{\text{lethal}}(t)$ to bound the region's size laterally. We denote the distance from a point

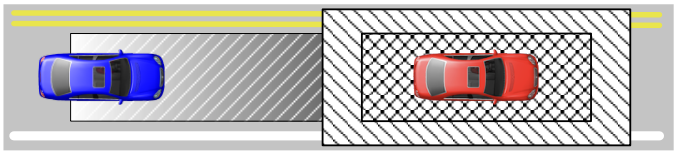


Figure 4.7: Shape of the cost function for moving obstacles. The cross-hatched area is *lethal* for the center of the to robot occupy, the hatched area is *high*-cost, and the graduated hatched area is low cost, diminishing in cost towards the left.

(x, y) to the vehicle by $f_{\text{dist}} = S_o(t) - s(x, y)$, where $S_o(t)$ is the station of the obstacle at time t, and $s(x, y)$ is the station of the point (x, y) in the road model. The resulting function is

$$c_{\text{follow}}(x, y, t) = \begin{cases} k^{\text{slope}}_{\text{follow}}(f_{\text{len}} - f_{\text{dist}}) & \text{if } 0 \leq f_{\text{dist}} \leq f_{\text{len}} \text{ and } \ell(x, y) \in D_{\text{lethal}}(t) \\ 0 & \text{otherwise,} \end{cases} \tag{4.3}$$

where $\ell(x, y)$ is latitude of the point (x, y). Figure 4.7 illustrates the resulting cost potential $c^{\text{obs}}_{\text{dynamic}}(x, y, t)$ at a single value of t.

The dynamic obstacle cost potential we have described treats vehicles behind the robot the same as vehicles in front of it, i.e., as though their motions are determined independently of the robot's motion. For example, if a vehicle behind the robot is traveling faster than the robot, the planner may select a trajectory that causes the robot to accelerate or move aside, rather than the more desirable behavior of expecting the following vehicle to regulate its own speed and position in response to the robot's behavior. This is arguably the best response when a very fast-moving vehicle is approaching from behind, but in most normal driving it is not. We leave this problem for future work. A possible solution is to predict future locations of vehicles behind the robot as though they were braking starting at the current time.

The moving obstacle cost function controls how close the robot can get to other vehicles. We must balance the need to stay a safe distance from other vehicles against the need to avoid them using graceful movements. This is the subject of the next section.

4.5.2 Velocities and Accelerations

Autonomous vehicles must respect speed limits, their own mechanical limitations, and their passengers' comfort. To keep the vehicle's behavior in line, we levy costs on the velocities and accelerations over a trajectory τ, making use again of the extended vehicle state vector

$$\hat{\mathbf{x}} = [x\ y\ \theta\ \kappa\ t\ v\ a\ \dot{\kappa}]\,,$$

and the sampling $\{\hat{\mathbf{x}}_i\}_{i=1}^{k}$ of states along τ.

Speed Limit

We encourage the vehicle to stay below the speed limit by imposing a constant cost $k^{\text{speed}}_{\text{dynamic}}$ if at any point along τ it exceeds the speed limit *limit*, i.e.,

$$c^{\text{speed}}_{\text{dynamic}}(\tau) = \begin{cases} k^{\text{speed}}_{\text{dynamic}} & \text{if } \max_i v(\hat{\mathbf{x}}_i) > \textit{limit} \\ 0 & \text{otherwise.} \end{cases}$$

Given a speed limit of v_{max}, this binary cost scheme will prevent the planner from selecting a plan that goes over the speed limit as long as the gain k_t in Equation 3.3 doesn't outweigh $k^{\text{speed}}_{\text{dynamic}}$. The vehicle will not speed unnecessarily as long as $k^{\text{speed}}_{\text{dynamic}}$ is chosen so that

$$k^{\text{speed}}_{\text{dynamic}} > \frac{k_t s_f(\tau)}{\textit{limit}}.$$

This scheme costs the same regardless of whether the vehicle edges ϵ over the limit or doubles it. In scenarios where speeding is necessary, more experimentation is needed to find a suitable schedule of penalties for higher speeds. Note that the vehicle's speed is limited absolutely, though indirectly, by the time horizon h_t from Equation 3.4 and the largest station value s_h of any vertex in the graph (Section 3.4.2). No plan can be generated that contains a velocity v_{big} so high that the vehicle would travel a distance s_h in time less than h_t, even were it to start

braking as hard as possible after reaching v_{big}.

An enhancement to investigate for future work would be making $c^{\text{speed}}_{\text{dynamic}}$ dependent on station as well as velocity, for regions where the speed limit is about to change and we wish the robot to anticipate and adjust its speed smoothly.

Longitudinal Acceleration

Any vehicle has a limited ability to accelerate given its weight and engine power, and to decelerate, given its tires, brakes, the road conditions, and other factors. We wish to avoid generating trajectories that exceed these limits. For simplicity, we use a pair of constants $a^{\text{max}}_{\text{hard}}$ and $a^{\text{min}}_{\text{hard}}$ for the maximum acceleration and deceleration the vehicle can sustain at any speed. Even when they are within the abilities of the vehicle, unnecessarily sharp acceleration and braking are undesirable. They cause premature brake and tire wear, waste fuel, and are uncomfortable to the passenger and potentially hazardous for other users of the road. We define a second set of limits $a^{\text{max}}_{\text{soft}}$ and $a^{\text{min}}_{\text{soft}}$ on acceleration to draw the line between normal and unusual accelerations.

The acceleration profiles described in Section 3.8.1 are selected or modified according to these limits. For constant accelerations (Equation 3.52), we simply avoid configuring the planner to use a value for a outside the allowed range. For a constant acceleration obtained by a target-velocity profile (Equation 3.51), we clamp the value of a within $[a^{\text{min}}_{\text{hard}}, a^{\text{max}}_{\text{hard}}]$. In these two cases, the value of a is constant over the trajectory. For distance-keeping acceleration profiles, a changes at each sample. In this case, we clamp a within the range $[a^{\text{min}}_{\text{hard}}, a^{\text{max}}_{\text{soft}}]$ at each integration step. That is, we are willing to brake hard when in distance-keeping mode to avoid a collision, but since the purpose of distance-keeping mode is smoother driving, we don't want to accelerate hard to just to catch up to traffic ahead. We discuss the distance keeping behavior further in Section 4.6.

In order to choose between normal accelerations, i.e., those within $[a^{\text{min}}_{\text{soft}}, a^{\text{max}}_{\text{soft}}]$, and unusual ones, those within $[a^{\text{min}}_{\text{hard}}, a^{\text{max}}_{\text{hard}}]$ but outside $[a^{\text{min}}_{\text{soft}}, a^{\text{max}}_{\text{soft}}]$, we use a binary cost, similar to the one we use to set speed limits. If the acceleration a is outside

Value	Description
$a = a^{\text{max}}_{\text{hard}}$	Hardest possible acceleration
$a = a^{\text{min}}_{\text{hard}}$	Hardest possible braking
$a = a^{\text{max}}_{\text{soft}}$	Hardest acceleration allowed without penalty
$a = a^{\text{min}}_{\text{soft}}$	Hardest braking allowed without penalty
$a = 0$	Maintain current speed
$v_f = 0.99*$ Speed limit	Stay a hair below the speed limit.
$v_f = 1m/s$	Make sharp turns at low speeds
$v_f = 0.01m/s$	Come to a virtual stop by trajectory end
$a = a_{\text{dk}}(t)$	Distance keeping (Equation 3.54)

Table 4.1: Acceleration profiles used in the planner

the range $[a^{\text{min}}_{\text{soft}}, a^{\text{max}}_{\text{soft}}]$ at any point along the trajectory τ, then a cost penalty is applied, that is,

$$c^{\text{accel}}_{\text{dynamic}}(\tau) = c_{accel}(\tau) + c_{decel}(\tau),$$

where

$$c_{accel}(\tau) = \begin{cases} k^{+}_{\text{accel}} & \text{if } \max_i a(\hat{\mathbf{x}}_i) > a^{\text{max}}_{\text{soft}} \\ 0 & \text{otherwise,} \end{cases}$$

and

$$c_{decel}(\tau) = \begin{cases} k^{-}_{\text{accel}} & \text{if } \min_i a(\hat{\mathbf{x}}_i) < a^{\text{min}}_{\text{soft}} \\ 0 & \text{otherwise.} \end{cases}$$

This binary scheme, much like the speed limit cost function $c^{\text{speed}}_{\text{dynamic}}$, is sufficient to discourage unnecessarily hard acceleration, but does not discriminate between degrees of hard acceleration. We set k^{+}_{accel} and k^{-}_{accel} small enough to allow hard acceleration to be used to avoid the *high* cost areas of obstacles. We leave finer distinctions among acceleration for future work.

The behavior of the planner depends on the family of acceleration profiles given it to choose from. Table 4.1 lists the acceleration profiles we use in our experiments. We cover both the soft and hard constant acceleration limits, ensure that we can reach the speed limit smoothly, maintain our current speed, and use

low target velocities to make hard turns at low speeds and come smoothly to a stop.

A large constant deceleration may bring the vehicle to a stop at some station s_{stop} before the end of a trajectory. This would mean the trajectory had effectively an infinite stopping time, and thus infinite cost. Emergency braking would thus be impossible along long paths. To remedy this, we fix v at a small value for stations $s > s_{\text{stop}}$. This ensures that the stopping time is finite, and that an emergency braking plan should be available.

We discuss the distance keeping behavior in Section 4.6, but we note here that when a trajectory τ uses the special distance keeping acceleration profile defined in Equation 3.54, it receives a cost bonus, i.e., when $a_2 = a_{\text{dk}}(t)$ in its corresponding graph edge $e = ([s_1\ \ell_1\ a_1 \ldots], [s_2\ \ell_2\ a_2 \ldots])$, it receives a discount $(-k_{\text{accel}}^{\text{distkeep}})$.

Future work should focus on a smoother set of acceleration profiles, defined for example by a higher-order polynomial. The cost functions we discussed in this section were applied to control inputs determined directly by the planner. In the next we section we describe the cost function for lateral acceleration, which is not determined directly by the planner. This requires a different way of assigning costs.

Lateral Acceleration

Just like the longitudinal acceleration, the lateral acceleration must be kept within reasonable limits. The lateral acceleration of the vehicle is the rate of change of velocity in the axis perpendicular to its heading. For our vehicle model, we derive $a_\perp$ from Equation 3.5. For simplicity, and without loss of generality, we pose the vehicle with $\theta = 0$, so that the lateral axis of the vehicle is aligned with the y axis.

Then the lateral acceleration is

$$\begin{aligned} a_\perp = \ddot{y} &= \frac{d}{dt}\dot{y} = \frac{d}{dt} v \sin\theta \\ &= a \sin\theta + \dot{\theta} v \cos\theta = 0 + (v\kappa)v \\ &= \kappa v^2, \end{aligned}$$

so $a_\perp$ depends just on v and κ.

Unlike the speed limit, the vehicle cannot physically exceed the limit on $a_\perp$. If the planner were to issue a plan that exceeds that limit the vehicle would not follow it. We would be justified in marking any trajectory that exceeds the limit as having an infinite cost. However, the control architecture we chose for our system renders this a doubtful measure. We used the architecture of the Tartan Racing system[28], where the state of the vehicle is perceived by the inertial navigation system (INS) and control system and fed into the planner. This means that an error in the vehicle control effort or noise in the perception could yield a starting state for the planner that is already above the limit on $a_\perp$. If we were to set trajectories that exceed $a_\perp$ to infinite cost, such a situation would leave us with no plans. Another possible control architecture is to set the starting state of the vehicle for the planner to the state predicted by the previous plan for the starting time, and depend on the controller to track that plan closely. In this case it may be reasonable to set non-conforming trajectories to infinite cost, because the planner would always use a starting state that was expressed in a previous plan and thus by construction would have to be below the limit. We will discuss the control system further in Section 5.2.1.

With the preceding discussion in mind, we set a limit $a^\perp_{\text{soft}}$ on absolute lateral acceleration that is below the actual capabilities of the vehicle. Any trajectory that goes above $a^\perp_{\text{soft}}$ is penalized by a large constant $k^{\text{soft}}_{a\perp}$. That is, we define

$$c^{\text{soft}}_{a\perp}(\tau) = \begin{cases} k^{\text{soft}}_{a\perp} & \text{if } \max_i |a_\perp(\hat{\mathbf{x}}_i)| > a^\perp_{\text{soft}} \\ 0 & \text{otherwise.} \end{cases}$$

This allows the planner to continue on a plan that is already above $a_{\text{soft}}^{\perp}$ while encouraging it to select a plan that is below it. A possible disadvantage is that since it costs the same to exceed the limit no matter by how much, a trajectory may be selected that goes above the absolute, hard limit on lateral acceleration $a_{\text{hard}}^{\perp}$. We have not observed such behavior in practice, but future work should address this concern.

Besides the binary penalty $c_{a\perp}^{\text{soft}}$ that marks the absolute limits of lateral acceleration, we use a linear penalty $c_{a\perp}^{\text{max}}$ that increases proportionally to the maximum lateral acceleration over the trajectory,

$$c_{a\perp}^{\text{max}}(\tau) = k_{a\perp}^{\text{max}} \max_i |a_{\perp}(\hat{\mathbf{x}}_i)|.$$

We choose to penalize the maximum rather than, for example, the integral of the lateral acceleration, since we reason that e.g. a constant $|a_{\perp}| = 0.3g$ over τ is preferable to a constant $0.1g$ with a brief spike at $0.5g$, but that integrating $a_{\perp}$ could cause the latter option to receive the lower cost. Note that applying a linear weighting to the max is non-linear: breaking a trajectory into two pieces at a particular point and evaluating them separately may give a different overall cost versus breaking the trajectory at some other point. In summary, the cost penalty for lateral acceration along a trajectory is

$$c_{\text{dynamic}}^{\text{lataccel}}(\tau) = c_{a\perp}^{\text{soft}}(\tau) + c_{a\perp}^{\text{max}}(\tau).$$

Future work should investigate whether a linear penalty is most appropriate, or whether for example $c_{\text{dynamic}}^{\text{lataccel}}(\tau)$ should increase with the square of the maximum $a_{\perp}(\hat{\mathbf{x}}_i)$.

Curvature Rate

Just like longitudinal acceleration, the rate at which the vehicle can turn the steering wheel has a hard limit. The turning rate is a pure output of the planner, that is, it does not depend on prior actions taken by the controller or any information from

the perception system. Therefore we can assign an infinite cost to any candidate edge in the graph where $\dot{\kappa}$ exceeds its limit, that is, we define

$$c^{\text{curv}}_{\text{dynamic}}(\dot{\kappa}) = \begin{cases} \infty & \text{if } |\dot{\kappa}| > k^{\text{curv}}_{\text{dynamic}} \\ 0 & \text{otherwise.} \end{cases}$$

There is no need to penalize higher turning rates, since this does not affect passenger comfort except indirectly through lateral acceleration, which is already penalized with $c^{\text{lataccel}}_{\text{dynamic}}(\tau)$.

4.5.3 Choosing the Path Type

In Section 3.3 and Section 3.4.2 we discussed whether to use a cubic or quintic polynomial to generate the paths from the vehicle vertex n_0 onto the sampled vertices n_i in the planning search graph. The purpose of the quintic spline was to ensure smooth steering at high speeds, since it ensures continuity in the curvature rate with respect to arc length, and its derivative, i.e.,

$$\frac{\mathrm{d}\kappa}{\mathrm{d}s}, \frac{\mathrm{d}^2\kappa}{\mathrm{d}s^2} \in C^0,$$

while the cubic spline guarantees only $\kappa \in C^0$. At low speeds, the cubic spline is acceptable because the discontinuities of curvature rate with respect to time are negligible, since arclength is changing slowly. Our question is how to choose between when to use a cubic, and when a quintic spline, for edges (n_0, n_i) coming from the start vertex.

One possibility is to set a cutoff point for some quantity, e.g. longitudinal velocity, and refuse to use cubic splines above that value. A disadvantage of this approach is that it can cause plan inconsistency by abruptly removing the plan most similar to the previous one as the vehicle's speed increases. Another option is to allow cubic splines at any speed, but ramp up the cost as speed increases. This has the disadvantage that it can allow the vehicle to make harsh, abrupt maneuvers,

which may be hard for the controller to follow, increasing tracking error outside the margin of error. However, assuming the cost is high enough, this would only occur in emergency situations. An even better solution would be to measure the practical limits on steering rate and its derivative and filter out trajectories that cannot be executed, as we did with $c^{\text{curv}}_{\text{dynamic}}$ in the previous section.

Since we were not able to run a real robot at high speeds to compare the effects of these policies, we chose a quadratic cost,

$$c_{\text{choose}}(\tau) = \begin{cases} k_{\text{choose}} v_0^2 & \text{if } \tau \text{ is cubic} \\ 0 & \text{if } \tau \text{ is quintic,} \end{cases}$$

where τ is the trajectory for an edge at the start vertex, i.e., (n_0, n_i), v_0 is the velocity of the vehicle at the start vertex n_0, and k_{choose} is a constant.

4.5.4 Cost Function Summary

In the foregoing we have described one possible cost function for our planner. It was not our goal to find the best cost function, just to demonstrate that a feasible cost function can be found relatively easily, and demonstrate a few variations, which we will see in the next section. Naturally, there are many ways to structure the cost function. Much more engineering would be necessary to settle on a function that engenders the desired behaviors, making tradeoffs between possible actions in various scenarios, assessing ride quality, and testing the vehicle in an exhaustive catalog of scenarios to show that it responds as desired in each.

We used an iterative approach to construct and tune the cost function. The first step is to identify physical quantities evaluated along the trajectory that should be either penalized or discounted. For example, lateral acceleration should be kept low, vehicle speed should be limited except in exceptional circumstances, and the trajectory should not come too close to obstacles. The second step is to combine constant, linear, and quadratic terms with step functions based on the characteristics of the quantity. For example, we combine step functions to penalize proximity

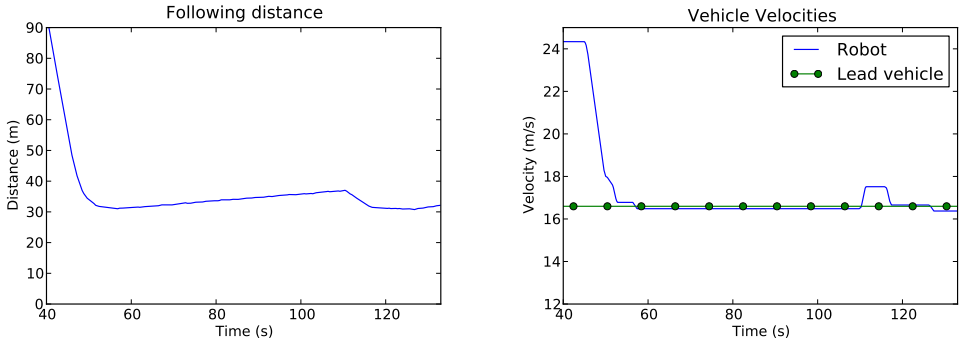
to obstacles, since it is not necessary to stay more than a few meters distant from another vehicle. As another example, we use a linear penalty against lateral acceleration up to a critical limit where the penalty steps up sharply. The third step in the iterative process is to evaluate a candidate cost function by running the planner through a variety of scenarios. Observations of its behavior suggest new quantities that should be penalized or discounted in the cost function. They also clarify the tradeoffs implicit in the weights of the terms. Tuning the parameters is a matter of trading x amount of one quantity, such as making forward progress, with y amount of another quantity, such as lateral acceleration when cornering or changing lanes.

In Chapter 6 we will see a system-level demonstration of the planner's behavior. In the following sections we investigate two emergent behaviors that come out of the planner and its cost function. We will see how a distance-keeping behavior emerges naturally from our planner and its cost function, and how we added special cases to refine its performance while staying within the planner framework. We will also see how we can build a safe lane changing behavior by adding a higher-level planner to manipulate the lane costs.

4.6 Distance Keeping

Adaptive cruise control (ACC) was one of the first and simplest autonomous behaviors to be sold in a stock commercial vehicle. The function of ACC is to regulate the speed of a vehicle so that it follows at a safe distance behind a leading vehicle, automatically adjusting speed as the lead vehicle speeds up or slows down.

To implement a distance-keeping behavior with our planner, we can use either the special distance-keeping acceleration profile $a_{\text{dk}}(t)$ (Equation 3.54), or the follow-cost region $c_{\text{follow}}(x, y, t)$ (Equation 4.3). In fact, these two are alternatives that can be used independently of one another. First we look at the behavior of $c_{\text{follow}}(x, y, t)$ without $a_{\text{dk}}(t)$.



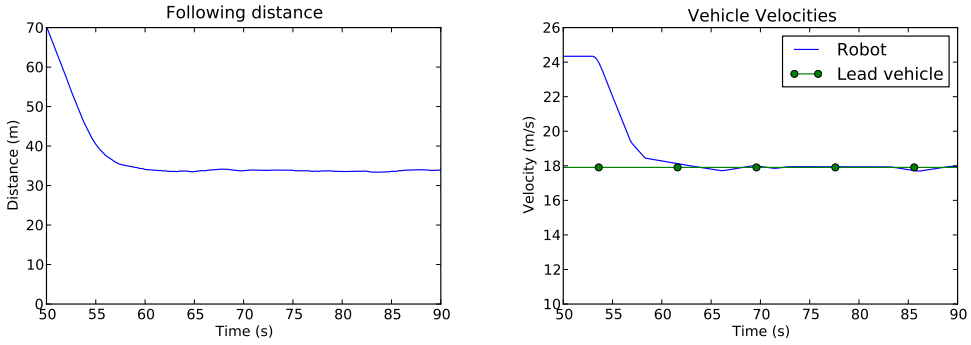
(a) Following distance (b) Velocities

Figure 4.8: Distance-keeping behavior using only constant acceleration profiles.

Without $a_{\text{dk}}(t)$, the final cost function Φ_c(Equation 3.3) encourages forward motion, while $c_{\text{static}}^{\text{lane}}$ keeps the robot in its current lane, and c_{follow} keeps the robot from getting too close to the vehicle ahead. Figure 4.8 shows an instance of this behavior with the robot following a leading vehicle at a constant speed. In this case $a_{\text{dk}}(t)$ is removed from the set of acceleration profiles (Table 4.1). We can see a problem with the behavior shown in Figure 4.8. The robot cannot arrive at exactly the same speed as the leading vehicle using the set of constant accelerations, causing it either to encroach into the *follow*-cost region or gradually fall behind. Though it may remain at the same velocity for long periods, it repeatedly accelerates in order to increase Φ_c or decrease the penalty due to *follow*.

We can add small constant acceleration and deceleration profiles of smaller magnitude than $a_{\text{soft}}^{\text{min}}$ and $a_{\text{soft}}^{\text{max}}$ (from Table 4.1) so that the planner can make small corrections without overshooting. Figure 4.9 shows the results of adding $a = \pm 0.1 ms^{-2}$. Using these smaller accelerations, the robot is able to stay closer to the leading vehicle's speed and maintain a more consistent following distance. However, it makes adjustments almost constantly.

We have just seen that a small set of fixed accelerations can create an emergent distance-keeping behavior without explicitly identifying a leading vehicle to follow. If in the behavior planning layer we choose which vehicle to follow, we



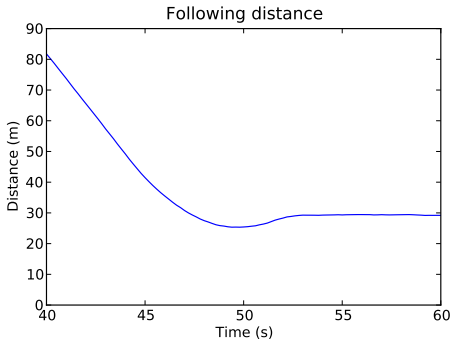
(a) Following distance

(b) Velocities

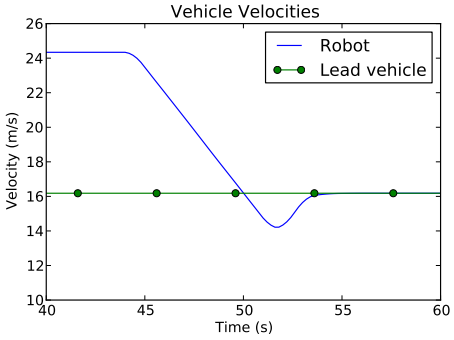
Figure 4.9: Distance-keeping behavior using only constant acceleration profiles, including small constant accelerations.

can take another approach. We can add an acceleration profile a_{lv} that converges directly to the velocity of the leading vehicle using Equation 3.51. This allows the planner to stabilize at the same speed as the leading vehicle. Figure 4.10 shows the result of adding this acceleration profile. The robot encroaches into the *follow*-cost region, then retreats and settles on the a_{lv} profile, matching the leading vehicle's velocity perfectly. The length of the robot's encroachment into the *follow*-cost region is governed by $k^{\text{slope}}_{\text{follow}}$(Equation 4.3). We can cure this overshoot by increasing $k^{\text{slope}}_{\text{follow}}$. Figure 4.11 shows the result of increasing $k^{\text{slope}}_{\text{follow}}$ by a factor of 5 over the experiment in Figure 4.10.

To obtain an even smoother velocity profile, we devised a PD-controlled acceleration profile $a_{\text{dk}}(t)$ especially for distance-keeping. We described this scheme in Section 3.8.1. We use the leading vehicle's velocity and location for $S_o(t)$ and v_o. To encourage the planner to use $a_{\text{dk}}(t)$, we apply a cost discount $-k^{\text{distkeep}}_{\text{accel}}$ to each trajectory edge that uses it, as we said in Section 4.5.2. We tune $k^{\text{distkeep}}_{\text{accel}}$ compared to k_t in Equation 3.3 to exceed any cost discount that could be gained from the $k_t t_f(\tau)$ term by accelerating and drawing closer to the leading vehicle. This ensures that the planner uses the distance-keeping profile whenever it is trailing another vehicle, even if the PD setpoint is outside the *follow*-cost region. Figure 4.12 shows $a_{\text{dk}}(t)$ in action. The planner begins using a_{dk} as soon as the

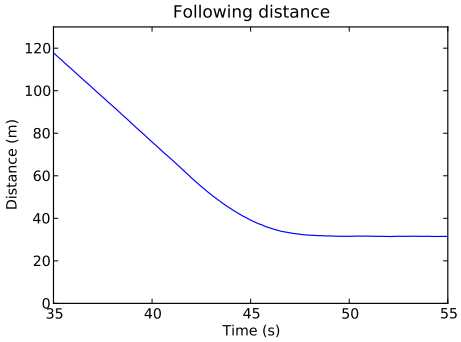


(a) Following distance

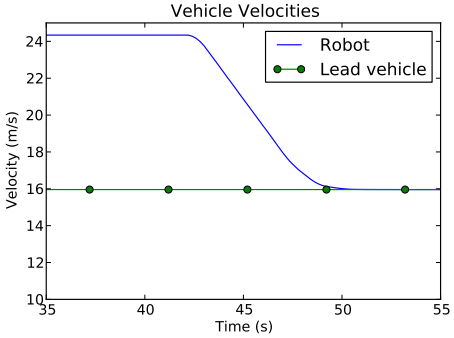


(b) Velocities

Figure 4.10: Distance-keeping behavior with an acceleration profile that mimics the lead vehicle's velocity using Equation 3.51. The robot comes too close to the lead vehicle before falling back.

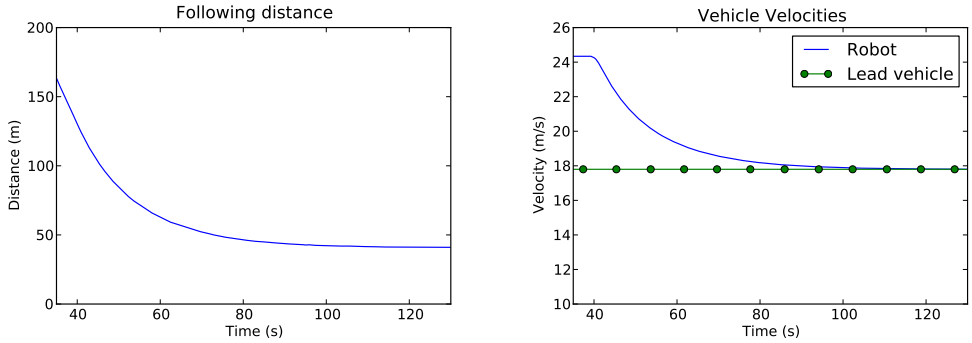


(a) Following distance



(b) Velocities

Figure 4.11: Distance-keeping behavior with an acceleration profile that mimics the lead vehicle's velocity using Equation 3.51, and an increased value for the *follow*-cost penalty $k_{\text{follow}}^{\text{slope}}$ to prevent the encroachment we saw in Figure 4.10.



(a) Following distance (b) Velocities

Figure 4.12: Distance-keeping behavior using the PD-control acceleration profile a_{dk}.

leading vehicle comes within view. The velocity adjustment downward is smooth and gradual. The desired spacing constants k_a and k_b from Equation 3.53 are set to prefer a larger following distance than the length of the *follow*-cost region.

In this section we explored several strategies for tuning our planner to make it exhibit a distance-keeping behavior. Given a diverse enough set of constant accelerations, we can obtain an emergent distance-keeping behavior without any special effort in the structure of the planning graph itself, as we showed in Figure 4.9. We can get smoother driving by using the higher-level behavior to identify the vehicle to follow and then adding edges to the planning graph, for example, by adding acceleration profiles that explicitly match the leading vehicle's velocity (Figure 4.11), or use a PD-control formula (Figure 4.12). Finally, we tuned parameters in the cost function to balance competing tendencies, getting the planner to draw up smoothly behind the leading vehicle, without overshooting. We finish the chapter in the next section by looking at a lane changing behavior.

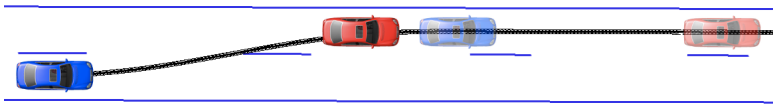


Figure 4.13: The SELECT-LANE behavioral algorithm (Figure 4.14) avoids passing other traffic on the right. It changes into the left lane and waits for the leading vehicle to move aside.

4.7 Lane Changing

When multiple lanes are available, the planner must choose which to use. We need to consider the speeds of vehicles in the current and neighboring lanes compared to the desired speed of the robot, i.e., the speed limit. The planner should attempt to drive in the lane that allows it to get closest to the speed limit without exceeding it. Given multiple such lanes, it should drive in the rightmost lane, as long as that would not cause it to pass another vehicle to its left. We must also consider whether the global road map shows a highway exit or turn coming up. The planner should change lanes well in advance of the turn.

We developed a simple behavioral algorithm SELECT-LANE, shown in Figure 4.14, that examines the available lanes and the other vehicles detected by the perception system, then selects a desired lane and changes the value of ℓ^{des} in Equation 4.2 accordingly. The policy is to pick the right-most lane that has no leading vehicle, or has a leading vehicle that is going at least as fast as the speed limit, or, if it has a leading vehicle that is going slower than the speed limit, is still far ahead (lines 7–9). It will not pass on the right (line 10), as shown in Figure 4.13. The planner should not try to change lanes right in front of a trailing vehicle (lines 11–16). There is also a limit on how frequently the behavior can choose a different lane. E.g. it will not change the value of ℓ^{des} more than every 5-seconds. The SELECT-LANE policy does not account for faster-moving traffic in the lane behind it. For example, if the speed limit is 100 km/h, and the traffic in the left lane is traveling at 110 km/h, while in the right lane it is 90 km/h, SELECT-LANE will choose to drive 100 km/h in the left lane. Our intention with SELECT-LANE was only to demonstrate that a simple algorithm can create

```
function SELECT-LANE
1:    require: lanes – list of available lanes
2:    require: vehicles – list of other vehicles
3:    global: ℓ^des – current desired lane
4:    global: lane_time – time of last change in ℓ^des

5:    ℓ^new ← leftmost lane in lanes
6:    for lane l in lanes from leftmost to rightmost do
7:       if ∃ closest vehicle v_a ahead of robot in lane l then
8:          if v_a exceeds speed limit or v_a is far ahead then
9:             ℓ^new ← l
          else
             // Stop searching; don't pass on the right
10:          break
          end if
11:      else if ∃ closest vehicle v_b in lane l behind robot then
12:         if v_b is far behind robot then
13:            ℓ^new ← l
14:         else // v is close behind
15:            if l ≠ ℓ^des then
                  // Stop searching; don't cut off v by changing into l
                  // (But do stay in l if already in it)
16:               break
               else
17:               ℓ^new ← l
               end if
         else
            // No vehicles ahead or behind in this lane
18:         ℓ^new ← l
         end if
      end for
      // Avoid frequent lane changes
19:   if ℓ^new ≠ ℓ^des and time − lane_time > hysteresis_lane
20:      ℓ^des ← ℓ^new
21:      lane_time ← time
      end if
```

Figure 4.14: The SELECT-LANE algorithm selects the desired lane based on the location and speed of other traffic.

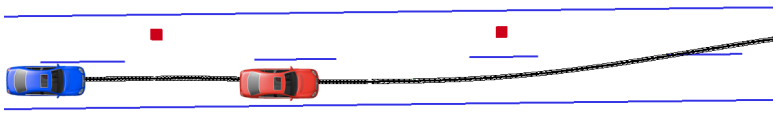


Figure 4.15: The SELECT-LANE behavioral algorithm (Figure 4.14) chooses the left lane based on the traffic ahead, but ignores static obstacles. The $c^{\text{lane}}_{\text{static}}$ and $c^{\text{obs}}_{\text{static}}$ cost functions cause the best plan to remain in the current lane, biased to the left, until the desired lane is clear.

reasonable lane-changing behavior that does not require careful engineering to ensure safety, since the trajectory planner can be relied upon to choose a safe action regardless of the errors in the planning layers above. Figure 4.15 illustrates that even though SELECT-LANE doesn't consider static obstacles when deciding which lane to be in, the robot plans to drive in its current lane until space opens up. The architecture of our system ensures that the flaws in SELECT-LANE nevertheless do not prevent safe vehicle behavior. This attribute stands in contrast to the top-down approaches reviewed in Section 2.1.10, where correct decision-making in the higher layers are needed to ensure safe behavior in the lower layers. In the next chapter we will show failures of other planners in the literature to cope with this situation.

A production lane selection behavior would be more complex, and we can see several flaws with this algorithm. For example, it does not take into account upcoming turns or exits, and the insistence on not passing other traffic on the right could cause our planner to park behind a stalled car in the left lane even when the right lane is free, if the cost penalty imposed by $c^{\text{lane}}_{\text{static}}$ for driving around the vehicle were high enough.

An alternative method is to simplify behavior algorithms such as SELECT-LANE even further by letting the trajectory planner choose the lane. Figure 4.16 illustrates an alternative $c^{\text{lane}}_{\text{static}}$ function. Each lane in the direction of travel has a valley shape to encourage lane centering. The trajectory planner automatically balances the potential amount to be gained in Φ_c (Equation 3.3) by changing lanes to get around a slow vehicle against the penalty of crossing the "hump" between

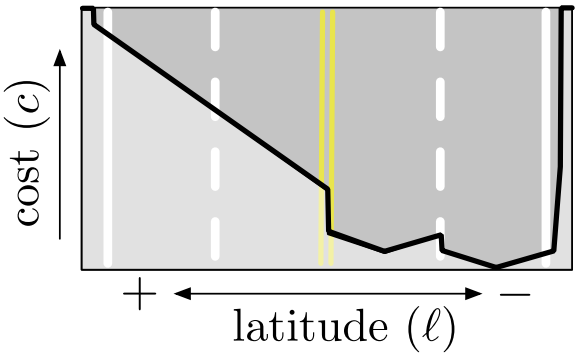


Figure 4.16: A lane cost potential function that allows the trajectory planner to choose the desired lane automatically, without the intervention of a higher-level behavioral planner.

the two lanes, to discourage frequent lane changing. The lower overall cost of the right-hand lane encourages the trajectory planner to choose the rightmost lane when there is no slower traffic ahead. We wil present the results of experiments with this method in Chapter 6. Preliminary results show a tendency to change lanes frequently around curves and when foliage is close to the edge of the road. One solution to the former would be to apply a higher base level of cost based on the curvature of each lane. We would need to invent additional mechanisms in the cost function to replace the behavioral considerations we made in SELECT-LANE. For example, to forbid passing on the right using this alternate lane selection method, we might add penalties in the $c^{\text{obs}}_{\text{dynamic}}$cost function over large areas to the right of each moving obstacle. We could also add a penalty similar to the *follow*-cost region of $c^{\text{obs}}_{\text{dynamic}}$, but in front of each moving obstacle, to discourage the planner from cutting them off.

A partial solution to these concerns is a combination of the preceding two approaches, shown in Figure 4.17. We can use SELECT-LANE to select a desired lane, but use it to draw lane costs in a manner similar to Figure 4.16. The selected lane has the lowest cost, and more distant lanes have progressively higher costs, but each with its own local minimum to promote centering when the vehicle must depart from the selected lane. In Chapter 6 we will compare all three

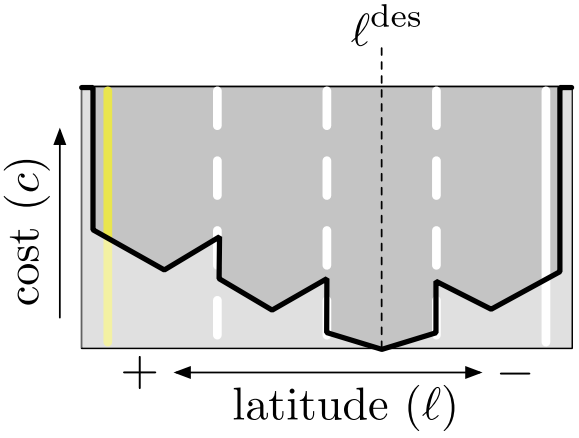


Figure 4.17: A lane cost potential function that blends the high-level lane selection behavior SELECT-LANE (Figure 4.14) with the approach in Figure 4.16.

lane-changing approaches experimentally.

4.8 High-Level Behaviors

In this work we have focused mainly on the challenges of highway driving. We have been particularly concerned with developing a planner that can respond to emergency situations, which we demonstrate in Chapter 6. We have left complex traffic interactions for future work, particularly those that involve coming to a complete stop. This includes stop signs, traffic lights, and making unprotected left-hand turns. In this section we briefly discuss how we might approach the creation of such behaviors within our planning framework.

For stop signs, we would need a high-level behavior planner similar in spirit to SELECT-LANE to determine where the robot should stop and when it is safe to proceed. A possible means of bringing the planner to a stop would be to manipulate the weightings in the final cost function (Equation 3.2) to give a discount to final states that have a velocity near zero in the vicinity of the stop line. If there is traffic in front then the normal distance keeping behavior would work until it was the robot's turn to stop at the line. It may be necessary to shorten the distance

between station samples $\{s\}$ (Section 3.4.2) in order to ensure that a feasible plan exists in the search space. Such an approach would allow the robot to stop at the stop line in normal situations, and to make alternate plans in emergency situations. Another method might be to keep the weights in Equation 3.2 the same and set the area covered by the intersection in the static cost map to high values, ensuring that paths which enter the intersection would cost more than paths which stop before it, even after factoring in the discount given in Equation 3.2 to paths that travel further.

For an unprotected left-hand turn, the high-cost areas projected by the predicted future locations of oncoming vehicles into the dynamic obstacle map should prevent the planner from proceeding with a turn until the way was clear. However, a vehicle should not proceed with a turn until the crosswalk is clear of pedestrians. A higher-level behavior could monitor the pedestrians and use either a high-cost area in the static cost map or manipulate Equation 3.2, just like the stop sign case previously.

When approaching a yellow traffic light, a car should stop if possible. Yellow lights could be handled by calculating the time at which the vehicle would reach the intersection while braking at the comfortable braking level $a_{\text{soft}}^{\min}$. The dynamic cost map would then be painted with higher costs for all later times, increasing sharply at the time the light is anticipated to turn red. If the light would turn red before the vehicle could stop at this rate, then the planner would select a plan using a harder degree of braking up to $a_{\text{hard}}^{\min}$, as long as the penalty for hard braking was lower than the penalty for entering the intersection.

In summary, we envision the approach to planning for intersections and other complex interactions as similar to the approach we took for distance-keeping and lane-changing. That is, higher level behavior algorithms would manipulate the cost function to reflect changing priorities, in keeping with the overall philosophy of the planner we articulated at the beginning of this chapter.

4.9 Summary

In this chapter we presented a trajectory cost function and higher-level behavior algorithm for highway driving. Our cost function blends safety and efficiency by using safety-oriented cost terms to mitigate the tendency of efficiency-oriented cost terms to push the system towards a catastrophic limit. We showed that we can satisfy higher-level behavioral constraints such as distance keeping and lane changing by manipulating the cost function and action set rather than specifying plans directly. Our planner provides a safety cushion against errors in higher-level behavioral routines.

The cost function we presented in this chapter was intended to draw an outline of the scope of the problem, and give an example of our reasoning process for designing a cost function to obtain desired vehicle behavior. A production system would require further engineering. For example, the cost function we proposed in this chapter would allow the robot to drive in another vehicle's blind spot. This and many other behavioral tweaks, some obvious, and some that would have to be uncovered through extensive testing, would have to be addressed through additional cost function refinements.

In the next chapter we will describe the implementation of our planner. The GPU is a novel parallel computing platform that will be unfamiliar to most readers. Programming strategies particular to the GPU are needed to reach their performance potential. These strategies can affect the design of the planner itself and therefore we consider the implementation itself to be a contribution of this thesis.

Chapter 5

Implementation and System Integration

In this chapter we describe the implementation of the planner. That is, we show how the theoretical ideas of the previous two chapters can be put into practice in an effective motion planning system integrated into a robot. We begin with the overall sequence of operations conducted by the core planner in Section 5.1, breaking down the order of operations. In Section 5.2 we describe how the planner is integrated into the Tartan Racing system, interfacing with the existing perception, planning, and control systems, and overcoming friction between the theoretical assumptions about the robot and its physical reality. Finally in Section 5.3 we return to the core planner and the cost functions of the previous chapter and show that we must invent new data structures and algorithms to accelerate the planner on a GPU.

5.1 Core Planner Implementation

In this section we describe the basic order of operations and data structures in the core planner implementation, assuming an "ideal" CPU implementation. We describe modifications to use the GPU beginning in Section 5.3.

5.1.1 Overall System Flow

Figure 5.1 shows the flow of major events in the planner. Each of these blocks is broken out into an additional block diagram.

Planner initialization (block (0) of Figure 5.1) sets the size and scale of the lattice in station-latitude and in time-velocity space, and the number of acceleration profiles to use when generating trajectories out of each lattice vertex. In the present implementation these quantities are set once and cannot be changed again. Other quantities that are more readily changed between planning cycles are the boundaries of the lookup tables for $c^{\text{obs}}_{\text{static}}$ and $c^{\text{obs}}_{\text{dynamic}}$, the limits on longitudinal acceleration and deceleration to be considered by the planner, penalties on excessively high lateral or longitudinal acceleration and the boundaries of what is

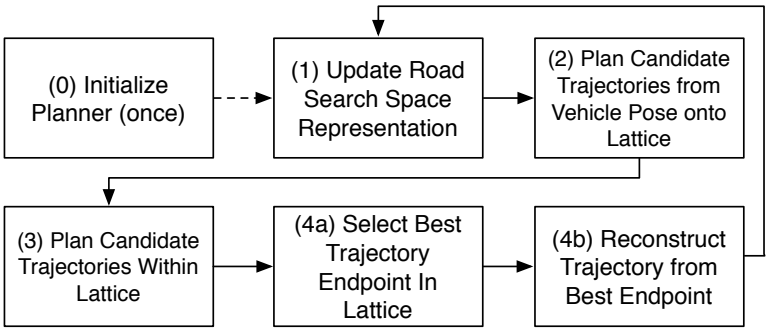


Figure 5.1: Block diagram of the overall flow of the planning system. Numbered blocks are broken down further in other figures as (1): Figure 5.2, (2): Figure 5.9, (3): Figure 5.12, (4a) and (4b): Figure 5.14.

considered excessive, and constraints on the steering system, including the maximum curvature that can be obtained and the maximum rate of change of curvature in the bicycle model.

The planning procedure repeated at every planning cycle is to update the search space representation with the perceived road shape and the obstacles and other vehicles perceived by the perception system (block (1)). These items are used to prepare data structures used for evaluating and comparing trajectories.

Using the terminology of Section 3.4.2, two types of candidate search graph edges are evaluated following the search space update. The first type, represented by block (2), are edges that start at the current vehicle state n_0. The second type, represented by block (3), are all other edges. The procedures for generating and evaluating each of these two types of trajectories are similar, but differ sufficiently to deserve separate descriptions.

After all trajectories are evaluated, the cost table $g(n)$ has a value for each vertex, giving the lowest cost known to reach the vertex. The next stage is to select the vertex representing the end of the best trajectory available, as per Equation 3.20. Given the vertex that represents the endpoint of the best trajectory, the actual trajectory that is the final output of the planner must be reconstructed using information stored alongside the cost table. The planner cannot store the full representation of the trajectory segments while it is planning due to space constraints

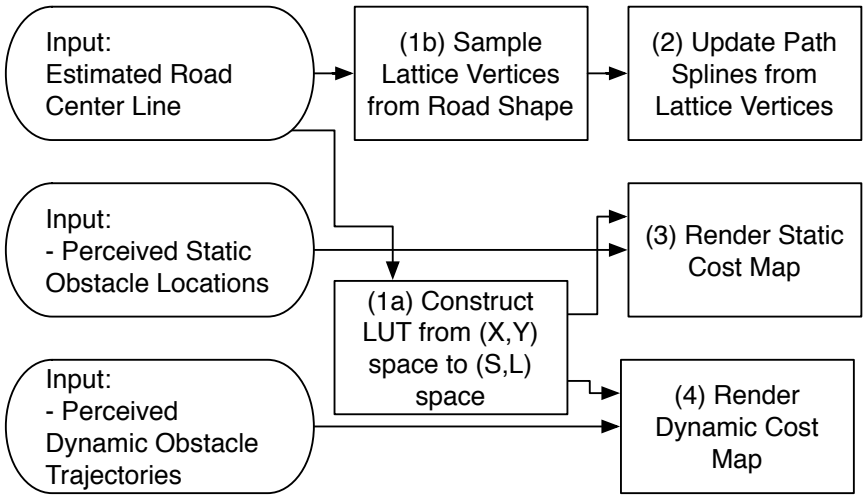


Figure 5.2: Detail of block (1) in Figure 5.1: Block diagram for the procedure to update the road search space representation, including the road shape and obstacle data structures.

- there is only room to store a small amount of information. The diagrams detailing the two blocks for selection (block 4a) and reconstruction (block 4b) of the best trajectory are represented in a single figure, Figure 5.14.

5.1.2 Update of the Road Search Space

At the start of each planning cycle, data structures representing road shape and perceived obstacles are updated based on information from the perception system. Figure 5.2 shows the major stages of this process. The first major input is the road center line $r(s) = [x, y, \theta, \kappa]$ given in the vehicle coordinate frame for a set of samples $\{s\}$. The Tartan Racing software system distinguishes between obstacles that are judged to be intrinsically static, and those that are inferred to be vehicles, either moving or with the ability to move. These two types of obstacles are provided in separate maps, and this distinction carries into the planning system, since the Tartan Racing platform was the first testbed for the implementation. The static obstacle map is expected to be a grid indexed by (x, y) in a world-fixed frame, which is translated and rotated into the vehicle coordinate frame. The

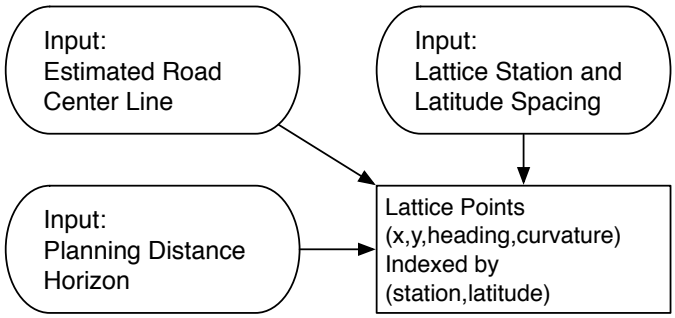


Figure 5.3: Detail of block (1b) in Figure 5.2: Block diagram for the procedure for sampling lattice vertices from the perceived road shape.

construction of the static map look-up table used by the planner is broken down further in Figure 5.5. The moving obstacles are expected to be accompanied by a prediction of their future locations in either of two forms: either in XY space as a set of samples indexed by time, or in station-latitude (a.k.a. SL) space, again as time-indexed samples. Figure 5.6 shows the stages in rendering the dynamic cost map.

We use a look-up table mapping from XY space to SL space (Block (1a) of Figure 5.2). The trajectories are generated and evaluated using (x, y) points, but some of the quantities making up the static and dynamic cost map tables are given in terms of (s, ℓ) points. Examples are the penalty for departing from the lane centers, and the extended shapes of moving obstacles, when painted into the dynamic cost map. We describe the construction of this table in Section 5.3.3.

The grid defining the embedding of the search graph in the vehicle state space (Equation 3.16) is updated in this stage (Block (1b) of Figure 5.2) using the road center line. This is described further in Figure 5.3. From the updated positions of the lattice, the cubic polynomial spiral paths ρ shared by the graph edges are updated so that they join the samples at their new locations.

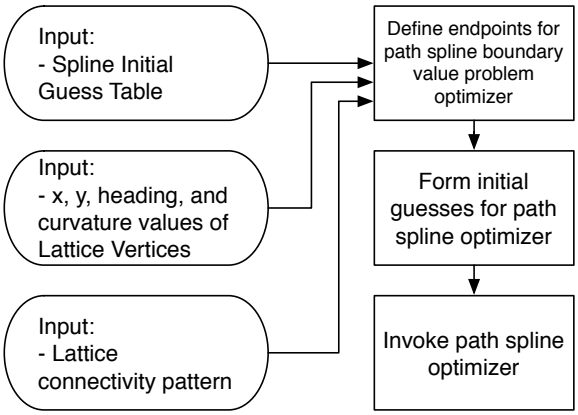


Figure 5.4: Detail of block (2) in Figure 5.2: Block diagram for the procedure of updating the parameters for splines describing the paths connecting the lattice vertices.

5.1.3 Sample Lattice Vertices

Figure 5.3 is a block diagram showing the process of constructing the lattice representation from the road center line. To maintain stability, the lattice does not move with the vehicle, but rather remains fixed to the road (requiring an estimate of vehicle station displacement since the last planning cycle), with vertices at the earliest station removed once the vehicle passes them, and vertices added at the furthest station. This helps the planner produce stable plans by ensuring that the plan produced in one planning cycle is still part of the search space on the subsequent planning cycle. The lattice (s, ℓ) coordinates are translated to (x, y, θ, κ) values by reference to the center line using Equation 3.10.

5.1.4 Update Lattice Path Splines

Figure 5.4 shows the block diagram for updating the parameters for cubic polynomial spirals that connect the vertices within the lattice. These spline parameters are used later in the search to generate candidate trajectories between lattice vertices.

The path splines that are to be computed are defined by an edge connectivity

pattern given as input to the planner (bottom input, Figure 5.4). Each set of SL-equivalent vertices, i.e., sets of vertices having the same (s, ℓ) coordinates as per Equation 3.4.2 are connected to other vertices using a set of station–latitude offsets $\{(\delta s_i, \delta \ell_i)\}_{i=1}^{k}$, so that a vertex

$$n_1 \overset{s\ell}{\sim} [s_j \; \ell_k \ldots]$$

is connected to all other vertices

$$n_2 \overset{s\ell}{\sim} [s_{j+\delta s_i}, \ell_{k+\delta \ell_i}],$$

for such n_2 as are in the graph. Since the $(x_0, y_0, \theta_0, \kappa_0)$, $(x_1, y_1, \theta_1, \kappa_1)$ coordinates of the samples only change slightly from one planning cycle to the next, the spline parameters from the paths of the previous cycle can be used as the initial input to the next cycle, reducing the number of iterations required to re-converge. If the corresponding path did not converge in the previous cycle, it is re-initialized.

When a path from the previous cycle is not available, either because it is the first planning cycle, or it did not converge, the spline initial guess table (top input, Figure 5.4) is used.

5.1.5 Initial Guess Table

The initial guess table is a precomputed table that stores initial values of $\mathbf{p}$ for the parameters in Equation 3.21 for use in the optimization algorithm of Section 3.6 to generate the cubic polynomial spiral paths used in the planner.

The table is constructed by solving boundary value problems for a start point $(x, y, \theta, \kappa) = (0, 0, 0, \kappa_0)$ and end point $(x_1, y_1, \theta_1, \kappa_1)$ by sampling in the space $I = (\kappa_0, x_1, y_1, \theta_1, \kappa_1)$, which defines the initial guess table. Table 5.1 shows sampling scheme of the initial guess table. A value for an initial guess can be obtained for any start- and end-point pair $(x_0, y_0, \theta_0, \kappa_0)$, and $(x_1, y_1, \theta_1, \kappa_1)$ by a rigid transformation of the pair that brings the start-point into the form

Parameter	Value/Range
Steps	16
κ_0, κ_1	$[-0.19, 0.19]$ radm^{-1}
x_1	$[1, 50]$ m
y_1	$[-50, 50]$ m
θ_1	$[-\pi/2, \pi/2]$ rad

Table 5.1: Sampling scheme of the initial guess table.

$\hat{\mathbf{x}}_\mathbf{0} = (0, 0, 0, \kappa_0)$ and changes the end-point to $\hat{\mathbf{x}}_\mathbf{1} = (\hat{x}_1, \hat{y}_1, \hat{\theta}_1, \kappa_1)$, then using a nearest-neighbor lookup into the guess table.

The initial guesses themselves are obtained using a relaxation method. The transformed start point $\hat{\mathbf{x}}_\mathbf{0} = (0, 0, 0, \kappa_0)$ is replaced with a simplified $\tilde{\mathbf{x}}_\mathbf{0} = (0, 0, 0, 0)$, and the transformed end point $\mathbf{x_1}$ is replaced with a simplified $\tilde{\mathbf{x}}_\mathbf{1} = (\hat{x}_1, 0, 0, 0)$. The path parameters are initialized to $\mathbf{p} = [p_3\ p_4\ s_G] = [0\ 0\ \hat{x}_1]$. The gradient-descent algorithm from Section 3.6 is then run for a fixed number of iterations k on $\tilde{\mathbf{x}}_\mathbf{0}$ and $\tilde{\mathbf{x}}_\mathbf{1}$. Before each iteration j the endpoints are adjusted,

$$\begin{aligned} \tilde{\mathbf{x}}_\mathbf{0} &\leftarrow (0, 0, 0, (j/k)\kappa_0) \\ \tilde{\mathbf{x}}_\mathbf{1} &\leftarrow (\hat{x}_1, (j/k)\hat{y}_1, (j/k)\hat{\theta}_1, (j/k)\kappa_1). \end{aligned}$$

Finally the gradient descent algorithm is run to convergence with the endpoints set to $\hat{\mathbf{x}}_\mathbf{0}$ and $\hat{\mathbf{x}}_\mathbf{1}$.

We do not use an initial guess table for the quintic splines. Since they are only used for the initial edges (n_0, n_i), there are fewer of them. We use a similar relaxation method to find the parameters for each spline as it is needed in the search.

5.1.6 Static Cost Map

Paths are evaluated by sampling points $\mathbf{x} = (x, y, \theta, \kappa)$ along their length and evaluating the cost function $c_{\text{static}}(\hat{\mathbf{x}})$, one term of which is $c_{\text{static-map}}(x, y)$, the static map term, which is a look-up table prepared in Figure 5.2. The static cost map is

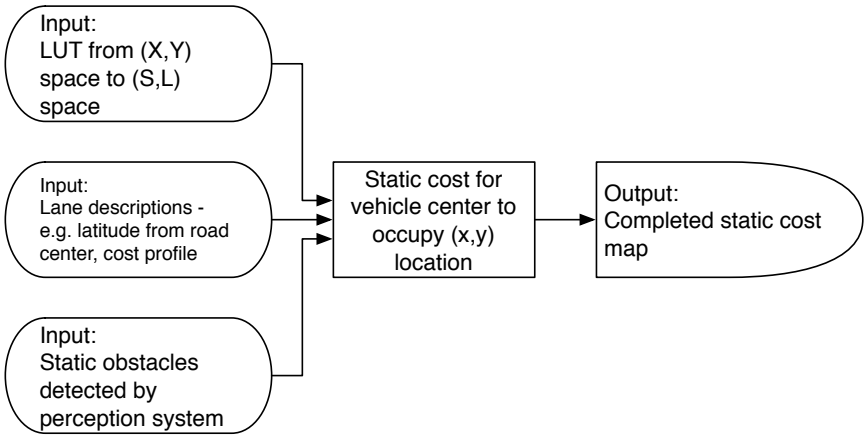


Figure 5.5: Detail of block (3) in Figure 5.2: Block diagram for the procedure to render the road shape and perceived static obstacles into a look-up table giving the cost for the vehicle to occupy any given (x,y) point at an orientation parallel to the road.

constructed using the lookup table mapping XY to SL space; the description of the number of lanes in their roads and lateral distances of their center lines from the main center line of the road; and the static obstacles detected by the perception system. These quantities are used to construct a look-up table that can quickly give the cost for the vehicle to occupy a point (x, y), regardless of the time or velocity at which it does so.

The static cost map is constructed using two types of data provided by the perception and higher-level behavioral system. First are the descriptions of the available lanes on the road, and their lateral distance from the road center line which defines the lattice search space. These should come from prior knowledge of the road obtained through mapping, or from the road detection system. Second are static obstacles detected by the perception system. The system currently assumes that anything moving is a vehicle. Static obstacles are provided as an occupancy grid with a given scale, origin and orientation with respect to the vehicle coordinate frame.

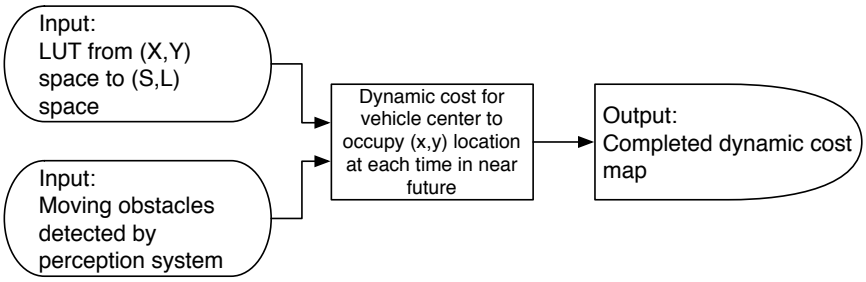


Figure 5.6: Detail of block (4) in Figure 5.2: Block diagram for the procedure to render perceived dynamic obstacles into a lookup table indexed by x, y, and time, giving the cost for the vehicle to occupy any given (x,y) point at an orientation parallel to the road, at a given time.

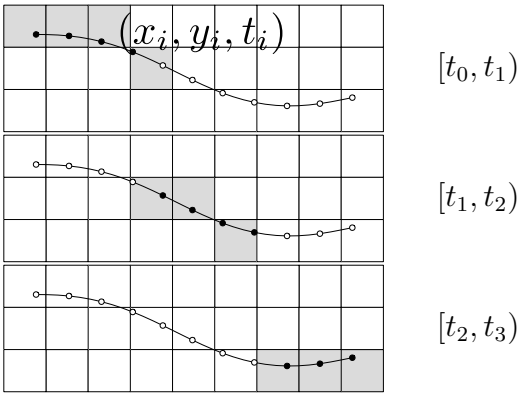


Figure 5.7: Using the (x, y, t) components of trajectory samples to index into the dynamic cost map.

5.1.7 Dynamic Cost Map

The dynamic cost map described in Section 4.5 is implemented as a lookup table CDYNAMIC that gives the cost for a trajectory to pass through a given (x, y, t) coordinate. Each cell in the table represents the presence of an obstacle at a range of times and locations described by a grid cell $[x_i, x_{i+1})$, $[y_j, y_{j+1})$, and $[t_k, t_{k+1})$. The future locations of perceived vehicles are sampled at multiple time steps $\{t_m\}$ and "painted" into the (x, y) layer for the time step $[t_k, t_{k+1})$ such that $t_k \leq t_m < t_{k+1}$. The painting is done similarly to static obstacles. Figure 5.8 shows how the predicted future locations of an object are painted into the appropriate t-layer of

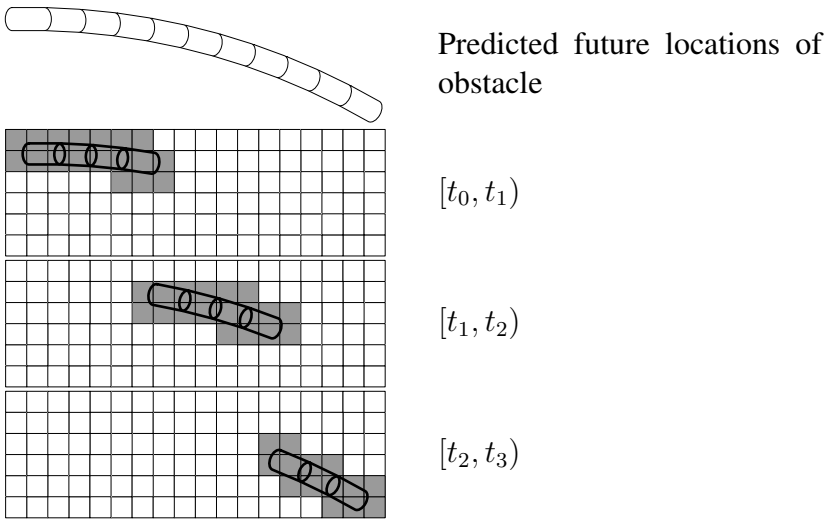


Figure 5.8: Painting a dynamic obstacle's future locations into CDYNAMIC.

the dynamic cost map CDYNAMIC. The footprint of the obstacle vehicle is dilated by two sets of distances. The first longitudinal dilation distance is half the length of the robot, and the first latitudinal dilation distance is half the width of the robot. Cells with (x, y) coordinates that fall within this dilation boundary are assigned the infinite "lethal" cost value. The next dilation distance is configurable, and cells falling within that distance are painted with a "hazard" cost unless already assigned "lethal". This penalizes trajectories that would bring the robot too close to other vehicles, either beside, behind, or in front.

At present, the dynamic map as constructed and used by the planner causes it to treat vehicles behind it the same as vehicles in front of it, so that, for example, if a vehicle behind the robot is traveling faster than the robot, the planner may select a trajectory that causes the robot to accelerate or move aside, rather than the more desirable behavior of expecting the following vehicle to regulate its own speed and position in response to the robot's behavior. We leave this problem for future work. A possible solution is to predict future locations of vehicles behind the robot as though they were braking starting at the current time.

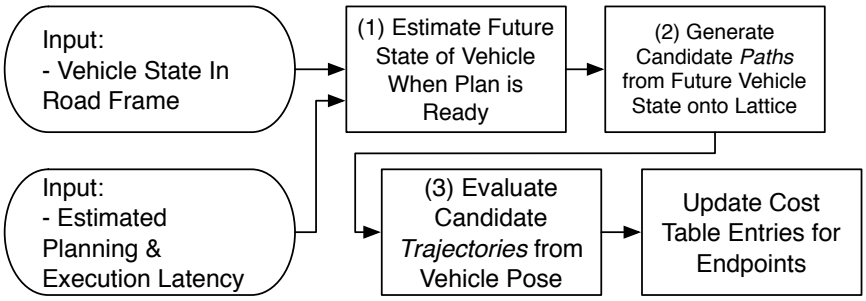


Figure 5.9: Detail of block (2) in Figure 5.1: Block diagram for the procedure to evaluate all candidate trajectories that start at the vehicle pose and end on the lattice.

5.1.8 Trajectories From the Vehicle to the Lattice

Once the steps for updating the road representation are completed, the trajectories for the edges (n_0, n_i) leading from the vehicle's position onto the lattice must be generated and evaluated. One of these trajectories will form the first segment of the full trajectory produced by the planner. The first step (Figure 5.9 block (1)) is to compensate for the latency in the planner and the execution latency of the control system. Since the vehicle continues moving while the planner works, we must estimate the future state of the vehicle at the time the plan to be produced is expected to come into effect and use that state at n_0. This procedure depends on the control system and is described further in Section 5.2, which describes the integration with the Tartan Racing system. Once the future vehicle start state is estimated, the planner can begin to generate paths that start at the vehicle's projected future static state $\mathbf{x} = [x\ y\ \theta\ \kappa]$, and end on the lattice. Once these endpoints are selected from the lattice, the planner optimizes path splines satisfying the endpoint constraints and samples along them to provide (x, y) points for evaluation with the cost function (block (2) of Figure 5.9). Then the planner samples a set of acceleration profiles along each path to generate trajectories, which are also evaluated (block (3)), and the costs of each are applied to the cost table (final block). The workings of blocks (2) and (3) are described further in the next sections.

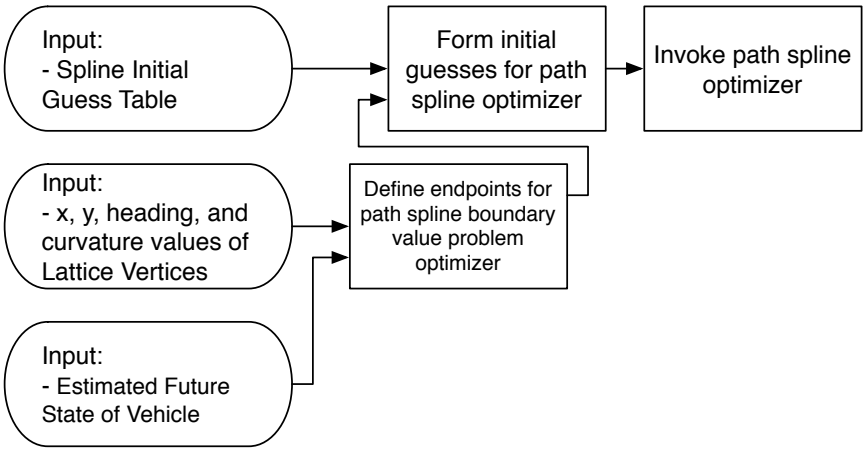


Figure 5.10: Detail of block (2) in Figure 5.9: Block diagram of the procedure for generating candidate path splines from the predicted future vehicle state onto the lattice vertices.

Generating Candidate Paths Onto the Lattice

The first stage in evaluating trajectories from the vehicle onto the lattice is to derive the underlying path splines. The actual optimization procedure is the same as the one used to update the splines joining the lattice (Section 5.1.4), though selection of endpoints in the lattice and the initial guesses differ, which we describe in this section.

The nodes of the lattice that are selected to be path endpoints are those close enough to the vehicle to:

- Keep numerical errors in the integration to obtain path (x, y) samples below a desired threshold. As we described in Section 3.6.1, the integration method (Simpson's with 8 steps) used to quickly optimize the path spline is different from that used to generate sample points along the path spline (Trapezoidal with up to 128 steps). For very long paths, these two methods can yield path points that diverge sufficiently to cause small though appreciable error. Out of the 379030 paths in our initial guess table (described in the next section) that are less than 60 meters long, 374955 of them, or 99 percent, had less than 30 cm error. The paths with large errors tend to con-

tain large changes in heading and yaw that are not feasible at high speeds. Of the 65928 paths that could be traveled at faster than 10 meters per second while experiencing less than 1 g of lateral acceleration, all had less than 15 cm error, with 99.7% less than 10 cm. Paths with error beyond a reasonable amount can be discarded in the search.

- Ensure that the spacing between (x, y) samples along each path are close enough together to ensure that almost all cells in the static and dynamic costs maps along the path are considered in the trajectory evaluation. If the path passes through a small corner of a cell it is reasonable to expect that no sample will fall in that cell. We rely on a safety margin in the obstacle dilation to mitigate the effects of such omissions.

Once the subset of lattice vertices to be sampled as path endpoints is selected, initial guesses are derived for the splines, as described for the intra-lattice splines in Section 5.1.4. For splines going onto the path, we use the initial guess table only. Once the guesses are obtained, the spline optimizer is invoked. Next, trajectories based on the paths are generated, sampled, and evaluated.

Evaluating Trajectories Onto the Lattice

Figure 5.11 illustrates the parts of the planner described in this section. The paths obtained in the previous section are sampled, using the predicted vehicle state and the spline parameters to integrate the system forward using Equation 3.21 and Equations 3.6–3.8. Samples are spaced at even values of arc length s. For each sample $(x(s), y(s), \theta(s), \kappa(s))$ along a path spline, the static cost terms of the cost function are evaluated. Then for each path, several trajectories are generated by sampling time and velocity coordinates using either a constant acceleration profile according to Equation 3.52 or the distance-keeping profile acceleration from Equation 3.54. For constant accelerations provided as a target velocity to be reached by the end of the path, the actual value a_v for the acceleration will be derived for the trajectory using the starting velocity v_0 of the vehicle, the length

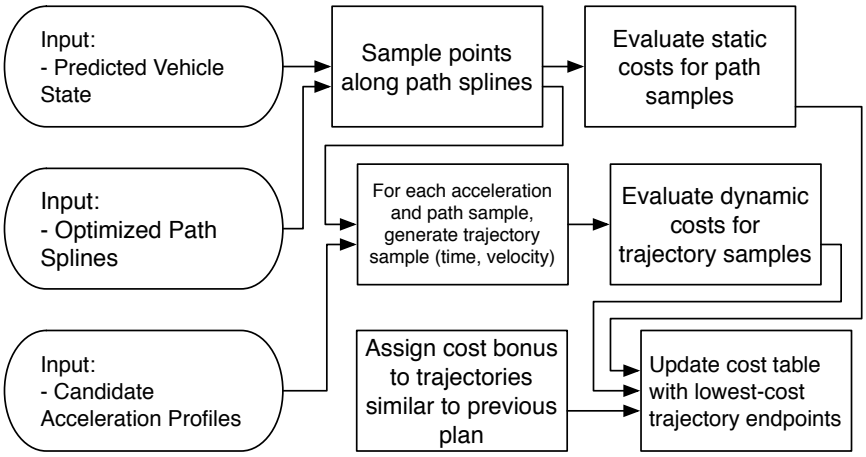


Figure 5.11: Detail of block (3) in Figure 5.9: Block diagram of the procedure to evaluate candidate trajectories using the path splines proceeding from the vehicle pose onto the lattice (Figure 5.10), and update the cost table.

of the path s_f, and the target ending velocity v_f. If a_v is outside the vehicle's abilities, it is clamped to the feasible range.

Given the acceleration profile, the trajectory is evaluated at the path samples, calculating the time t and velocity v at each sample given its arclength s. For a constant deceleration, the velocity may reach zero before the end of the path, making the time value at the end of the path infinite. We then bump up the velocity to a small constant value and run the rest of the path to obtain a finite ending time. We describe the implementation of the cost function further in Section 5.1.11.

5.1.9 Trajectories Within the Lattice

Figure 5.12 frames the discussion in this section. In the previous section we described the procedure for generating trajectories that originate at the vehicle and end on the lattice. In this section we describe the generation and evaluation of *internal* trajectories, i.e., those that start and end within the graph and are not connected to the start vertex n_0.

The internal path splines are generated and updated during the road represen-

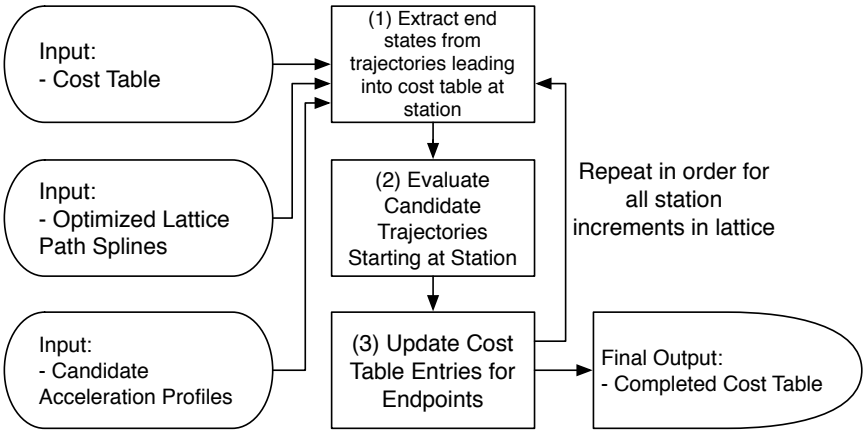


Figure 5.12: Detail of block (3) in Figure 5.1: Block diagram for the procedure to evaluate all candidate trajectories interior to the lattice, and update the cost table with the values of the best trajectories incoming to each lattice point.

tation update phase (Section 5.1.2). Beginning at vertices with $s(n) = 0$, vertices that already have a finite cost assigned are extended by sampling their outgoing edges, which necessarily land at later stations due to the structure of the graph. In this phase, multiple candidate trajectories are likely to end in the same cost table cell, and the planner chooses the lowest-cost trajectory to represent the cell among the candidates. At the end of this phase, all edges have been generated and evaluated, and we know the lowest cost to reach each vertex in the graph.

Referring to Figure 5.13, we see that the procedure to generate and evaluate trajectories within the lattice is similar to that used to generate and evaluate trajectories onto the lattice (Figure 5.11 and Section 5.1.8). The main difference is that instead of starting from a single state, the predicted future vehicle state when the plan takes effect, for each station s, the planner iterates over all $(l, a, [t], [v])$ coordinates, extracting the precise time $t([t])$ and velocity $v([v])$ of the best trajectory ending at that cell. For each spline path p leading out of the (s, l) coordinate, each of the acceleration profiles a is applied in turn, and the trajectory is evaluated.

Referring to block (3) of Figure 5.12, each trajectory τ is assigned a cost $c(\tau)$ using the cost function, and ends on a lattice node $(s, l, a, [t], [v])$. When two

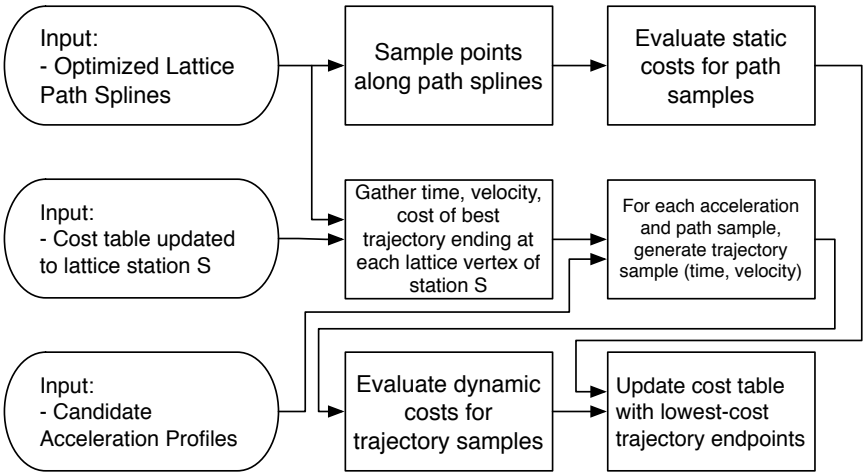


Figure 5.13: Detail of block (2) in Figure 5.12: Block diagram for the procedure to evaluate candidate trajectories starting at all lattice vertices at a given station, and proceeding to some lattice vertex at a greater station.

trajectories τ_1, τ_2 end at the same point (s, l) on the lattice, were generated by the same acceleration profile a, and have ending times $t_f^1 = t(s_f(\tau_1))$, $t_f^2 = t(s_f(\tau_2))$ falling in the same time index range $t_f^1, t_f^2 \in [t]$, and likewise for ending velocities, then the trajectory with the lowest cost is selected for assignment to the cell. In order to reconstruct the best trajectory at the end of the search, besides the cost, we also store:

- the ID of the originating vertex of the lowest-cost incoming edge
- the final time of the trajectory
- the final velocity of the trajectory
- the acceleration profile of the trajectory
- the spline parameters used to generate the underlying path ρ

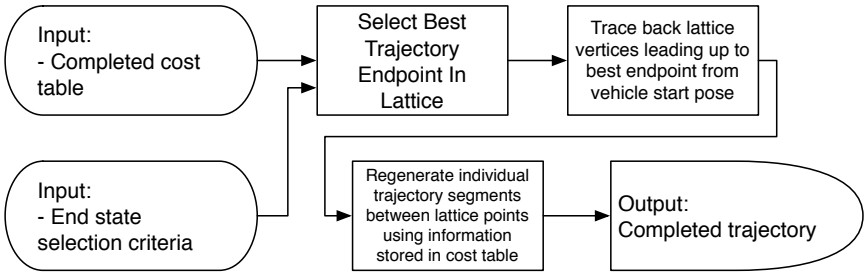


Figure 5.14: Detail of block (4a) and (4b) in Figure 5.1: Block diagram for the procedure for reconstructing the best trajectory through the lattice by analyzing the the cost table.

5.1.10 Reconstructing the Best Trajectory

Once the cost table has been completed, each entry represents the end of a trajectory starting at the vehicle position and transiting through possibly several intermediate entries. Each entry (s, l, a, t, v) contains a cost along with other data used to reconstruct the trajectory ending at that entry.

We use the final cost function $g(n) + \Phi(n)$ given in Equation 3.20 to choose the best vertex n_f at which to end the trajectory. From this vertex we trace back the sequence of edges and vertices n_i that led to it through the graph from the start vertex n_0. The final trajectory in the form of a sampled function of time $\tau(t) = (x, y, \theta, \kappa, v, a)$ is reconstructed by concatenating the trajectories generated for the edges, in the same way the trajectories are first generated during the evaluation phase of planning. This trajectory is the final output of the planner, to be passed to the controller.

5.1.11 Cost function

In Section 4.3 we described the trajectory cost function. In this section we describe how the terms of the cost function are actually evaluated in the implementation.

function STATIC-COST(ρ,$\{s_i\}$) $\rightarrow c_{\text{static}}$
  $c_{\text{static}} \leftarrow 0$
  for each sample $(x, y, \theta, \kappa) = \rho(s_j), s_j \in \{s_i\}$
    if $|\kappa| > \kappa_{\text{max}}$ **then** $c_{\text{static}} \leftarrow \infty$
    $c_{\text{static}} \leftarrow c_{\text{static}} + c^{\text{lane}}_{\text{static}}(x, y) + c^{\text{obs}}_{\text{static}}(x, y)$
  $c_{\text{static}} \leftarrow (c_{\text{static}})\,(s_f(\rho))\,/\,(\#s_i)$ *// Normalize by steps/meter*

Figure 5.15: Algorithm for computing the static cost of a spline path given as samples.

Path (static) cost function

For each spline path $\rho(s) = (x(s), y(s), \theta(s), \kappa(s))$ expressed as a set of samples over station $\{s_i\}$, $\rho(\{s_i\}) \rightarrow \{(x(s_i), y(s_i), \theta(s_i), \kappa(s_i))\}$, the cost is assigned according to the algorithm in Figure 5.15. The cost is the sum of the static map costs, normalized by the number of samples and the length of the path, so that neither changing the number of samples nor evaluating the path in smaller pieces changes the value.

Trajectory (dynamic) cost function

Terms in the cost function including time and velocity are computed separately from the path (static) cost function. Figure 5.16 shows the algorithm used to to evaluate the trajectory. First, the acceleration profile index is mapped to an actual constant acceleration to apply over the course of the trajectory (lines 2–4). Next, the cost of the trajectory samples is computed (line 5). Finally, if the acceleration(or deceleration) exceeds a soft limit demarcating the boundary between comfortable and uncomfortable levels of throttle and braking, a further penalty is applied to the cost (lines 6–7). Figure 5.17 describes the cost function applied to the trajectory samples. For each sample along the path ρ, the time and velocity at which the sample is reached are computed (line 3), given the velocity starting the trajectory, the arc length of the sample, and the acceleration profile being applied. From the velocity, the lateral acceleration $a_\perp$ is calculated (line 5). We track the

function DYNAMIC-COST(n, ρ, a_{index}) $\rightarrow c_{\text{dynamic}}$

1: $c_{\text{dynamic}} \leftarrow 0$
2: **if** a_{index} is an acceleration value type **then**
3: $a \leftarrow a(a_{\text{index}})$
else
// a_{index} is a final-velocity value type
4: $a \leftarrow \text{ACCEL}(v_0(n), s_f, v_f(a_{\text{index}}))$
end
5: $c_{\text{dynamic}} \leftarrow$ DYNAMIC-COST-SAMPLES$(\rho, \{s_i\}, t_0(n), v_0(n), a)$
// Penalize extreme longitudinal accelerations
6: **if** $a >$ soft-limit[accel] **then** $c_{\text{dynamic}} \leftarrow c_{\text{dynamic}} +$ penalty
7: **if** $a <$ soft-limit[decel] **then** $c_{\text{dynamic}} \leftarrow c_{\text{dynamic}} +$ penalty

Figure 5.16: Algorithm for computing total cost of a trajectory segment, also invoking DYNAMIC-COST-SAMPLES (Figure 5.17)

maximum lateral acceleration $\max_{a_\perp}$ and longitudinal velocity encountered over the course of the trajectory (lines 4–6). The curvature rate $\dot{\kappa}$ is calculated using the spline parameters and velocity and compared to a limit derived from the abilities of the steering system (line 7). If it exceeds the limit then the path cannot be followed at that speed and acceleration, and thus the trajectory is rejected. The cost is then incremented by the dynamic obstacle cost function (line 8). After all samples have been evaluated, the table-based costs are normalized by the path length and the number of samples (line 9) as with the static cost function (Figure 5.15). The maximum lateral acceleration is penalized in two ways. The first is a linear weighting (line 10). We choose this approach rather than apply a penalty based on the integrated lateral acceleration over the trajectory because we reason that e.g. a constant $0.1g$ acceleration is preferable to a constant $0g$ acceleration with a brief spike at $0.3g$ – and that integrating the acceleration could cause the latter option to receive a lower cost. Note that applying a linear weighting to the maximum lateral acceleration along a trajectory is non-linear: breaking two trajectories into two pieces each and evaluating them separately might give costs with a different

function DYNAMIC-COST-SAMPLES(ρ,$\{s_i\}$, t_0, v_0, a) $\rightarrow c_{\text{dsamp}}$

1: $c_{\text{dsamp}}, \max_{a_\perp}, \max_v \leftarrow 0$
2: **for** each sample $(x, y, \theta, \kappa) = \rho(s_j), s_j \in \{s_i\}$
3: $(t, v) \leftarrow$ TIME-VELOCITY$(v_0, s_i, a(s_i))$
4: $\max_v \leftarrow \max(\max_v, v)$
// $a_\perp$ is lateral acceleration
5: $a_\perp \leftarrow \kappa v^2$
6: $\max_{a_\perp} \leftarrow \max(\max_{a_\perp}, |a_\perp|)$
7: **if** $|\dot{\kappa}| >$ hard-limit$[\dot{\kappa}]$ **then** $c_{\text{dsamp}} \leftarrow \infty$
8: $c_{\text{dsamp}} \leftarrow c_{\text{dsamp}} + c^{\text{obs}}_{\text{dynamic}}(x, y, t)$
end for
9: $c_{\text{dsamp}} \leftarrow (c_{\text{dsamp}})\,(s_f(\rho))\,/\,(\#s_i)$
// Linear penalty for maximal lateral acceleration
10: $c_{\text{dsamp}} \leftarrow c_{\text{dsamp}} + (\max_{a_\perp})\,(\text{penalty}[\max_{a_\perp}])$
// Constant threshold penalty for maximal lateral acceleration
11: **if** $\max_{a_\perp} >$ soft-limit$[a_\perp]$ **then** $c_{\text{dsamp}} \leftarrow c_{\text{dsamp}} + \text{penalty}[a_\perp]$
// Penalty for exceeding the speed limit
12: **if** $\max_v >$ speed-limit **then** $c_{\text{dsamp}} \leftarrow c_{\text{dsamp}} + \text{penalty}[\text{speed}]$

Figure 5.17: Algorithm for computing the cost of trajectory samples.

ordering. The second way is to add a constant (typically high) penalty if the lateral acceleration exceeds a safety limit derived (line 11) from the performance envelope of the vehicle.

5.1.12 Accessing the Cost Table

The cost table stores the cost of the best trajectory known to reach each vertex in the search graph (Section 3.4.2).

Cost Table Indexing

The cost table is in five dimensions, indexed $(s, \ell, a, [t], [v])$. The major indices are the integer coordinates (s, ℓ) designating a particular pose (x, y, θ, κ) along the road. The acceleration profile index $a = a_{\text{index}}$ of the best trajectory is also part of the index for reasons discussed in the next section. Finally, the time and velocity are included, with a particular final acceleration profile. The final two coordinates are the indices of the ranges $[t]$ and $[v]$, containing the exact time and velocity at which the best trajectory arrives at the lattice vertex.

As an example, take a trajectory segment τ with cost $c(\tau)$, ending at the lattice point indexed (s_τ, ℓ_τ). Suppose τ is obtained from its underlying path by applying acceleration profile index a_τ, so as to arrive at time t_τ carrying a velocity v_τ. The cost table entry will be indexed $(s_\tau, \ell_\tau, a_\tau, j, k)$ such that $t_j \leq t_\tau < t_{j+1}$ and $v_k \leq v_\tau < v_{k+1}$.

Cost Table Updating

As each trajectory τ is evaluated, the cost table cell $C[\tau] = C(s, \ell, a, [t], [v])$ in which it lands is identified according to Section 5.1.12. If $c(\tau)$ is less than the current value, then the table is updated with $c(\tau)$ and additional data needed to reconstruct the path up to that point are also stored. See Algorithm Cost-Table-Update in Figure 5.18.

function COST-TABLE-UPDATE(ρ, τ)
    cost-index $\leftarrow (s_\tau, \ell_\tau, a_\tau, [t_\tau], [v_\tau])$
    if $c(\tau) < C$[cost-index].c **then**
        C[cost-index].$c \leftarrow c(\tau)$
        C[cost-index].$t \leftarrow t_\tau$
        C[cost-index].$v \leftarrow v_\tau$
        // The actual constant acceleration applied, as opposed to the profile index
        C[cost-index].$a \leftarrow$ rendered-accel(a_τ)
        // start(τ) is the cost table index from which τ originates
        C[cost-index].cost-pred $\leftarrow$ start(τ)
        C[cost-index].$\rho \leftarrow \rho$
    end

Figure 5.18: The cost table update algorithm applied to the end point of each trajectory after it is evaluated.

5.2 Tartan Racing Integration

The implementation of the planner was tested by integrating it into the Tartan Racing Urban Challenge (TR) system, replacing the "local planner" task[30]. In the race system there is an intimate relationship between the "behavior executive"[8], which infers a semantic state from the traffic and the structure of the nearby road network, and the local planner which drives along a lane center line inferred by the behavior task as the correct course, up to a speed suggested as appropriate given the speed limit and traffic conditions. Our proposed lattice planner holds a similar position in the planner hierarchy to the local planner, but replicating the interaction with the behavior planner would have been overwhelmingly complex. We effectively replaced the behavior task by limiting our tests to road networks with simpler structure, and avoiding complex interactions with traffic that would not help to demonstrate the contributions of our planner. An example is waiting for the robot's turn in a four-way stop scenario.

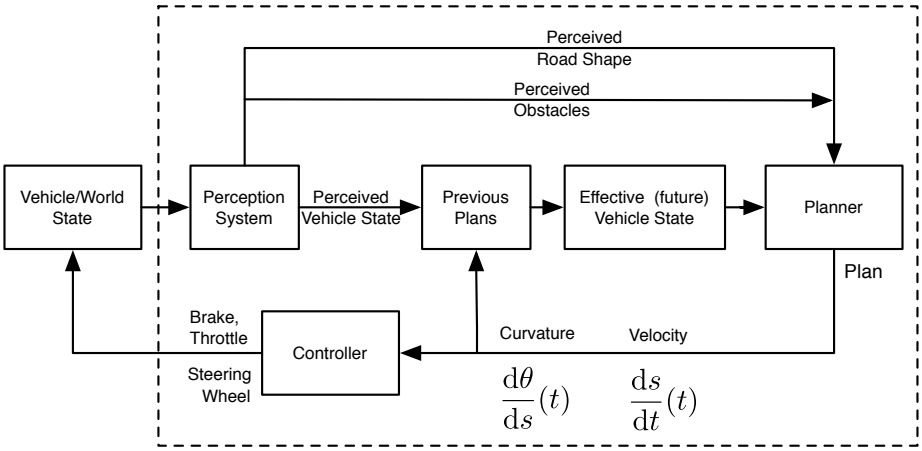


Figure 5.19: The system block diagram including the planner (block "Planner")

5.2.1 Control Architecture

Figure 5.19 outlines the control architecture of the Tartan Racing system, showing how the planner relates to the controller that turns commanded path curvatures and velocities into signals to the steering wheel, throttle, and brake actuators. The local planner is responsible for producing a trajectory $[\kappa(t)\ v(t)\ a(t)]$, expressing path curvature, longitudinal velocity, and longitudinal acceleration as a function of time. The local planner generates these trajectories at a rate of 10 Hz, and sends them to a controller task that issues $[\kappa(t)\ v(t)\ a(t)]$ vectors at a rate of 100 Hz to a vehicle control system that translates these quantities into control signals for the vehicle's steering, throttle, and brake actuators.

By the time the planner produces a trajectory from its input, the vehicle has moved on. The planner must compensate for this latency, as well as the latency in the vehicle's physical response to the actuator commands. That is, control signals that have already been issued for the steering, throttle, etc. have yet to actually take effect on the vehicle. The local planner (and our proposed planner) uses an estimate of the delay in the control system and the delay of the planner itself to forward-simulate the effect of the plan generated at the previous (or earlier if necessary) planning cycle on the vehicle, starting from the given current estimate,

using the same bicycle model of Figure 3.8 used to generate the plan. We estimate the control delay on Boss to be 80 ms, and our planning latency is estimated to be 200 ms, so that the given vehicle state $\hat{\mathbf{x}}$ is run through the simulator using the segment of the previously generated plan starting 80 ms earlier and ending 200 ms later than the current time. The planner's current assumption is that the vehicle steering and velocity controllers will execute the plan sufficiently well that any error in plan execution can be neglected. This is a flawed assumption, which should be addressed in future work.

5.2.2 Trajectory Queue

The trajectory τ_i sent to the controller is a vector-valued function $[\kappa_i(t)\ v_i(t)\ a_i(t)]$, defined over $t \in [t_i^0 \ldots t_i^f]$. The planner and controller maintain a list $\{\tau_i\}$ of the trajectories produced by the planner, such that each trajectory meshes smoothly with the next, i.e., $\kappa_i(t_{i+1}^0) = \kappa_{i+1}(t_{i+1}^0)$ and $v_i(t_{i+1}^0) = v_{i+1}(t_{i+1}^0)$. The acceleration a is permitted to be discontinuous in our implementation due to the design of our search space. When multiple trajectories τ_i are valid over a time t, the controller picks the trajectory with the most recent start time. It is necessary to retain older trajectories to perform latency compensation, which we describe next.

5.2.3 Latency compensation

We would like each trajectory to start from the current state of the vehicle. This is impossible because the vehicle state will have changed by the time the trajectory has been generated and is ready for the controller to execute. In addition, commands already sent by the controller do not immediately take effect on the vehicle. Latency in the actuators causes additional latency in trajectory execution. To compensate for this latency, the future state $\hat{\mathbf{x}}$ of the vehicle when the next trajectory is expected to actually begin to take effect is used in place of $\mathbf{x}$. This future state $\hat{\mathbf{x}}$ is derived by simulating the effect of the trajectories already issued and in the queue on the most recent perceived state $\mathbf{x}$.

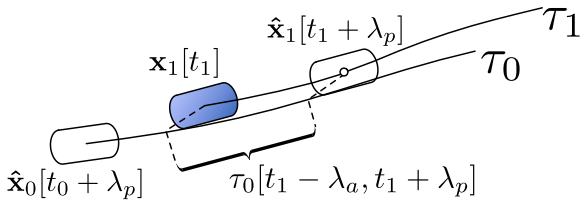


Figure 5.20: Latency compensation in the planner. We begin with a trajectory τ_0 that was generated from a predicted future state $\hat{\mathbf{x}}_0[t_0 + \lambda_p]$. At time t_1 we receive a new observation $\mathbf{x}_1[t_1]$ of the vehicle state. We extract the portion of τ_0 for the time range $[t_1 - \lambda_a, t_1 + \lambda_p]$ and apply it to the vehicle model (Equation 3.5) with initial state $\mathbf{x}_1[t_1]$ to obtain the predicted future state $\hat{\mathbf{x}}_1[t_1 + \lambda_p]$. The predicted state $\hat{\mathbf{x}}_1[t_1 + \lambda_p]$ is used as the current state n_0 of the robot (Equation 3.17) in the planning algorithm to produce a new trajectory τ_1.

We define λ_p as the latency due to the time taken to generate the plan, and λ_a the actuation latency. The future state $\hat{\mathbf{x}}$ is derived by extracting the trajectory commands in $[t - \lambda_a, t + \lambda_p]$ from the queue of past trajectories. The interval $[t - \lambda_a, t + \lambda_p]$ is discretized into samples $\{t_j\}$ and the command for each t_j is drawn from the most recent τ_i that contains it. The samples are then applied in open-loop fashion to the latest perceived state $\mathbf{x}$. Figure 5.20 illustrates this process.

5.3 GPU Implementation

Getting maximum performance out of the Nvidia GPU can require some tricky algorithmic maneuvering. The GPU offers the greatest speedup when a problem can be divided into thousands or millions of independent subproblems, where one thread solves each subproblem, and where threads can be grouped *a priori* into cohorts that can be expected to usually follow the same branches in the program, and usually access near-contiguous memory locations.

In the rest of this section we describe our implementation of the lattice planner on the GPU, with a discussion of how we structured the data and operations in order to get the most performance within the constraints set by the platform.

The planning cycle goes through several phases, which we will describe in order:

- initialization
- plan-onto-lattice
- plan-within-lattice
- extract best trajectory

The control flow of the GPU acceleration version is the same as the CPU version, so we can refer back to Figure 5.1 for the following, and we describe the changes made to accelerate the planner on the GPU in the same order.

The reader is referred to Appendix A for an overview of the Nvidia GPU and the main features that the programmer must take into account when programming it. In the following we focus on the interplay between the GPU features and the lattice planner data structures and functions. We begin in the next section with a brief recapitulation of the concepts that guide programming on the GPU.

5.3.1 Processing Concepts

At a gross design level, we arrange the interaction between CPU and GPU according to the following general principles:

Minimize size and frequency of data transfer

Data transfer between the CPU and GPU is restricted, with relatively high latency and low bandwidth. Ideally one would transfer input data to the GPU, allow all processing to happen on the GPU, and then copy results back. However, some algorithms run so poorly on the GPU that it may be better to transfer intermediate results back to the CPU, process them, then send them back to the GPU. These situations often require clever programming in order to keep the performance advantage of the GPU from leaking away.

Minimize number of distinct kernel invocations

The *kernel* is the basic CUDA function. Threads are invoked to run the same function, and they must all run until completion before another kernel can be invoked (see Section A.3 for more). The overall computation takes place on the CPU, and parts that can be done in parallel are delegated to the GPU to run in a batch processing mode. When all threads in a batch need to synchronize, a new kernel must be run (at least on contemporary GPUs). When the data organization changes between phases such that the number of discrete data items and therefore threads changes radically, it is usually better, again, to invoke a distinct kernel. For many algorithms, these concerns are not relevant, but for the lattice planner, multiple global synchronization steps are required to correctly update the cost table. A kernel cannot be launched until the previous one is finished, and due to the block organization of threads, it is possible for most of the processors on the GPU to be idle while the last few threads in a kernel finish.

Asynchronous and interleaved processing and copying of data

Memory transfers between CPU and GPU can be performed simultaneously with computations on both of them. Independent sequences of operations in the overall computation should be arranged to maximize simultaneity.

Batch processing of concise data items

On a CPU it poses no problem to performance to process one data item all the way through from beginning to end using a single function. On a GPU it may be better to break a processing pipeline up into discrete kernels in which each thread loads one data item from global memory, processes it, stores the results back, and exits to make way for the next kernel in the pipeline. In the lattice planner we do this if a data item d must be fissioned into multiple items d_i, which can be processed independently. This happens with paths in the planner as they are multiplied into many trajectories each. On a CPU the d_i for each d can be processed in a loop, but on the GPU we

```
struct S {
  int item1[ArraySz];
  int item2[ArraySz];
};
S array;
```

(a)

```
struct S {
  int item1;
  int item2;
};
S array[ArraySz];
```

(b)

Figure 5.21: The two typical ways of organizing arrays of structured data are (a) Structure-of-Arrays(SoA), and (b) Array-of-Structures(AoS).

prefer to have one thread process each data item. Since the total number of d_i is much greater than the number of d, we launch distinct kernels each with the appropriate numbers of threads.

Consecutive threads should access consecutive data

The GPU achieves a high memory bandwidth using a wide memory bus, with some versions up to 512 bits. The bus can transfer a lot of data when memory accesses are contiguous. It is important to choose the correct layout for data considering how the algorithm will access it. It may be beneficial to change the layout of a large data structure between phases of processing. This can mean transposing a matrix when the access pattern changes from iterating along rows to iterating along columns, or changing an array of structures into a structure of arrays, as shown in Figure 5.21.

Having these processing concepts in mind, we can now describe how they are applied to the lattice planner. We begin with the initialization phases.

5.3.2 Center Line-Derived Data Structures

At the beginning of each planning cycle, we use the center line estimated by the perception system (in the Tartan Racing system, it is extracted from the global road map) to prepare the road-related data structures. Figure 5.2 shows the pieces.

The road center line is given in an array of evenly-spaced samples indexed by station, where s is station, and $s = 0$ where the current vehicle projects onto the center line.

$$\{(s_i, x_i, y_i, \theta_i, \kappa_i)\}_i .$$

Since they are evenly spaced, the station of each sample is implicit in the index, and the remaining four elements (x, y, θ, κ) fit conveniently into `float4`, a 4-element float structure that is supported natively by the GPU texture lookup and interpolation hardware.

```
struct float4 {
  float x, y, z, w;
};
```

Although the GPU texture hardware is built specially for indexing into and optionally interpolating within 1-, 2-, and 3-D graphical textures to apply to polygons, it is useful for other kinds of data and operations, as long as they are read-only. Rather than have threads access the center line array directly from global memory, we deposit it into a one-dimensional GPU texture.

When we should choose to use a texture over the global RAM depends on the expected access patterns. For best memory performance with regular global RAM, threads in the same "warp" should access a contiguous block of memory. When it is not feasible to make this arrangement, and if all the accesses are read-only, textures can be used instead. Texture accesses perform best when threads in the same warp access locations that are spatially near one another. Vanilla RAM is accessed directly, with no intervening cache. Textures, however, are accessed from the RAM through a special cache that opportunistically groups nearby requests. The main constraint on textures is that they are read-only during the life of a single kernel invocation, they can only be changed in between kernels, and they use fixed global names, i.e., they cannot be passed as parameters. Since the center line texture is relatively small, the texture cache should be able to hold all or most of it.

The first data structure derived from the center line is the array of lattice points, mapping a grid (i, j) to points $(s(i), \ell(j))$, as described in Section 5.1.3. This is inexpensive and is performed on the CPU.

5.3.3 Mapping from X-Y to S-L Space

The map entry is constructed for each XY cell by projecting onto the road center line and interpolating to find the smallest perpendicular distance from the cell to a segment joining center line samples. The station of the interpolated point is used as the station corresponding to the XY cell, and the distance is the latitude.

We will make much use later of XYSL, a continuous mapping from (x, y) space to (s, ℓ) space. We need this mapping because the paths are represented as (x, y) coordinates in the vehicle frame, but some of the cost functions are defined in terms of station and latitude on the road, for instance, the lane centering cost term, the occupancy grid for moving obstacles, and the distance keeping action.

The mapping XYSL is constructed by sampling a regular grid of discrete (x, y) points, then for each point (x_i, y_i), finding the closest center line sample point $(s_j, x_j, y_j, \ldots)$. The closest point (x_i^c, y_i^c) along the center line is linearly interpolated between adjacent sample points along with the station s_i^c. The latitude of the point (x_i, y_i) is finally calculated as the signed distance to (x_i^c, y_i^c) - negative if to the right of the center line and positive if to the left.

For each point we search recursively for the closest center line sample, using a rough linear search in station increments of 10 meters, then recurse and search exhaustively to find the closest sample.

On the GPU, one thread is allocated per (x_i, y_i) output sample. Each thread is completely independent, and the algorithm requires no special measures be taken for the GPU except accessing the center line through the texture cache.

When XYSL is accessed, the texture hardware is used to linearly interpolate between the sample points. The road shape $r(s)$ is defined for some latitude ℓ, and is transformed to other latitudes using Equation 3.10. Figure 5.22 shows the error that can result from this approach. It is typically below 3cm, which we

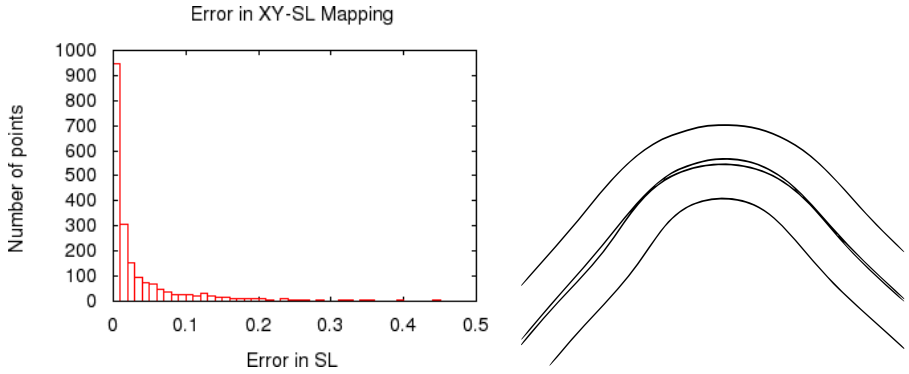


Figure 5.22: Left: Histogram of error magnitude in XYSL for sample points. We sampled points in SL space, mapped them to XY using $r(s)$ to generate ground truth, then compared them to the SL values obtained through the XYSL map. Right: the sample points were taken from a sharply curving road. The highest error points were at the inside of the turn.

consider to be acceptable, except for a few points at the very inside of a sharp turn. At these points, the station coordinate is in error, but the latitude is close to correct. In this case, $r(s)$ was defined as the center of the outside lane, and was transformed to the inner latitudes using Equation 3.10. We could reduce the error by requiring the perception system to provide an independent $r(s)$ for each lane, rather than a single $r(s)$ for the entire road.

5.3.4 Lattice Internal Paths

Paths proceeding from one lattice vertex to another are optimized once the vertices' poses are determined. One thread is allocated per path. The core optimization routine for the paths operates on a large array of inputs, so a dedicated routine sets up the inputs defining the initial guess, the starting states, and the desired final state along each path. Automatic variables used by the optimization routine demand more registers than are available, so scratch space in global memory for the Jacobian, its inverse, and other intermediate results is allocated by the host routine. Going to global memory for these intermediate quantities is acceptable since accesses are infrequent compared to the computational load, so that the ALUs can

be used by some threads while other threads wait for memory.

5.3.5 Internal Static Cost Map

As described in Section 4.4.1, the behavior layer selects a preferred lane to drive along and an optional bias away from the center of the lane. To encourage the planner to generate a plan following these preferences, a cost function $c^{\text{lane}}_{\text{static}}(x, y)$ is defined over the coordinates of the road, with lower values where we prefer to drive.

We implement $c^{\text{lane}}_{\text{static}}$ as a grid of (x, y) points in the vehicle frame. As with the XYSL map, the $c^{\text{lane}}_{\text{static}}$ map is constructed using one thread per (x, y) output sample, and each thread is almost completely independent. The one exception stems from the input data arrays used to describe the latitudes of the lane centers, their base costs, and their directions. Since each array is accessed several times by each thread, it should be accessed through a cache. One could use the texture cache, but since the number of lanes even on a large road is only on the order of tens, we load the lane data into the programmable shared memory cache, which is large enough for this purpose and more convenient to program.

5.3.6 External Static Cost Map

We use a static cost function $c^{\text{obs}}_{\text{static}}$ (Section 4.4.2), constructed from perception data of static obstacles imported from the Tartan Racing (TR) software system. These data form a binary grid G_0 with 0 representing safe areas, and 1 representing perceived lethal obstacles. No part of the vehicle may touch a lethal obstacle. This map is represented separately from the lane cost map $c^{\text{lane}}_{\text{static}}$ because the internal static cost map is expressed in the vehicle frame, while the TR system expresses the external map in a world frame. Rotating or resampling them along the same orientation would introduce error, so instead we represent each map independently.

We represent the static cost function $c^{\text{obs}}_{\text{static}}$ using a table, CESTATIC. This table

approximates the $\mathcal{C}$-space expansion of the obstacles so that the many collision checks done later during path cost evaluation can be done quickly. As discussed in Section 4.4.2, we assume that the vehicle will remain sufficiently aligned with the road, so we compute the expansion in SL space, thereby saving the effort of computing an expansion that supports checking against multiple orientations.

As discussed in Section 4.3, we wish to score a path by sampling (x, y) points and computing a score derived from the proximity of the position to obstacles in the perception map. If the vehicle would overlap an obstacle in the grid, the (x, y) point is given infinite cost so that no plan can be formed that passes through an obstacle. To promote plans that stay a healthy distance from obstacles, the vehicle shape is dilated and tested for overlaps again – those (x, y) points that overlap get a soft penalty - a high cost value that allows paths to pass through if no lower-cost path is available.

The obstacle dilation is implemented using a distance transform in SL space. Since we dilate obstacles separately along the S and L dimensions, we can use the Manhattan metric, which leads to a simpler and faster algorithm than a Euclidean metric. Parallel Euclidean distance transforms for GPUs are an active research topic, see [16] for some recent progress. Since the SL space is a warped version of $x - y$ space, we correct the dilation threshold along S by the local curvature at each cell, $\hat{s} = s(1 - \kappa\ell)$, in order to get correct dilations in XY. The steps of the algorithm are as follows. The algorithm is repeated for the separate soft (finite) and hard (lethal) cost dilations.

- We allocate a grid G_1in SL space with a higher resolution than the input grid, to retain accuracy in the intermediate dilations.
- Using one thread per cell of G_0 in XY space, we use the XYSL map to trace the outline of the SL-dilation of each occupied cell into G_1.
- Due to implementation time constraints, we compute the 1-D distance transform along S on the CPU, although a prefix-sum algorithm could be used on the GPU. The result is a grid G_2.

- On the GPU, we generate a grid G_3 using one thread per station increment of G_2 to compute a thresholded distance transform along L for those cells that are within the dilation threshold along S, as adjusted by $\hat{s} = s(1 - \kappa\ell)$.

Having used the above algorithm twice to generate two G_3 grids, G_3^{soft} representing the soft-cost obstacle dilation and G_3^{lethal} representing the lethal-cost dilation in SL-coordinates, we generate the output grid G_4 with the same scale and dimensions as G_0 by taking

$$G_0[x, y] = \max(G_3^{\text{soft}}[\text{XYSL}(x, y)], G_3^{\text{lethal}}[\text{XYSL}(x, y)]).$$

5.3.7 Dynamic obstacle cost map

Moving obstacles are expressed by the TR system as a set of j time-indexed sets of k_j samples $\{(t_i, x_i, y_i, \theta_i)\}_{i=0}^{k_j}$ giving the predicted future locations of other vehicles detected on the road. Vehicles are assumed to be the same size. We use these to construct a three-dimensional lookup table CDYNAMIC. The GPU supports a three-dimensional texture format we use to represent CDYNAMIC, which we can index using discrete (x, y, t) coordinates. Taking a similar approach to CESTATIC, we assume that both the robot and the detected vehicles are and will remain parallel to the road, and use a simplified $\mathcal{C}$-space expansion method. For each obstacle, we iterate over nearby (x, y) points, look up the corresponding (s, l) value, and compute a penalty based on the station, latitude, time, and velocity. The obstacle shape is dilated more when it is further away in time, thereby diminishing the impact noisy sensor readings may have on tricking the robot into committing to brittle maneuvers near other vehicles several seconds into the future. Faster-moving obstacles are dilated even more. In our implementation, we define a function $D_S^{\infty}(t, v)$ that gives the amount each obstacle sample should be dilated along the station direction to create a lethal cost region as

$$D_S^{\infty}(t, v) = {}^1k_s^{\infty} + {}^2k_s^{\infty} tv.$$

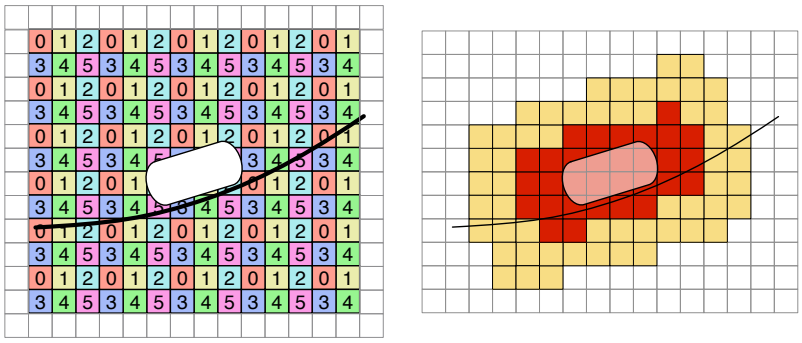


Figure 5.23: Left: Pattern in which threads (numbered) evaluate cells surrounding a single obstacle sample in CDYNAMIC. Referring to Figure 5.24, $\Gamma_x = 3$ and $\Gamma_y = 2$. Right: the resulting CDYNAMIC cost map at a single time slice. The center line (curved) indicates the direction of expansion in S-L space.

The function $D_L^{\infty}(t)$ for the lateral expansion of the lethal-cost region is defined more simply as

$$D_L^{\infty}(t) = {}^1k_s^{\infty} + {}^2k_s^{\infty} t.$$

We dispense with the dependency on velocity since we assume the perception system's estimate of a vehicle's lateral position does not depend on its longitudinal velocity. Functions $D_S^H(t, v)$ and $D_L^H(t)$ for the higher-cost regions surrounding the lethal region are defined similarly.

Each sample set $\{(t_i, x_i, y_i, \theta_i)\}_{i=0}^{k_j}$ must be spaced closely enough to prevent gaps in the map. There is a many-to-many relationship between obstacle samples and the cells they affect. One could choose to

- Use one or more threads per obstacle sample and iterate over nearby cells.
- Use one or more threads per cell and iterate over obstacle samples.

We chose the former after experimenting with both approaches. We use one block of threads for each obstacle sample, and each thread evaluates a subset of cells in CDYNAMIC according to the pattern in Figure 5.23. The procedure is formalized in function INIT-DYNAMIC-MAP, shown in Figure 5.24.

function INIT-DYNAMIC-MAP:

// Obstacle samples

input: $\{(x_i^o, y_i^o, t_i^o, v_i^o)\}$ *// Obstacle samples*
input: XYSL *// Mapping from X-Y to S-L space*
input: Δ *// Size of XY area surrounding obstacle*
// sample in which to limit expansion.
input: $D_S^\infty(t, v)$ *// S-dilation factor for obstacle lethal cost*
// region as function of its time t_i^o and velocity v_i^o.
input: $D_L^\infty(t)$ *// L-dilation factor for obstacle lethal cost region*
// as function of its time t_i^o
input: $D_S^H(t, v)$, $D_L^H(t)$ *// Dilation factors for high-cost region*
output: CDYNAMIC *// Dynamic cost map*
forall obstacle samples $\{(x_o, y_o, t_o, v_o)\}$ **do**
$(s_o, \ell_o) \leftarrow \text{XYSL}(x_o, y_o)$
launch thread block of dimensions (Γ_x, Γ_y)
forall threads (γ_x, γ_y) in thread block **do**
for $\delta_x \leftarrow -\Delta/2 + \gamma_x$ to $\Delta/2$ incrementing by Γ_x **do**
for $\delta_y \leftarrow -\Delta/2 + \gamma_y$ to $\Delta/2$ incrementing by Γ_y **do**
$(x, y) \leftarrow (x_i^o + \delta_x, y_i^o + \delta_y)$
$(s, \ell) = \text{XYSL}(x, y)$
if $|s - s_o| < D_S^\infty(t_i^o, v_i^o)$ or $|l - l_o| < D_L^\infty(t_i^o)$ **then**
$c \leftarrow \infty$
else if $|s - s_o| < D_S^H(t_i^o, v_i^o)$ or $|l - l_o| < D_L^H(t_i^o)$
$c \leftarrow H$
end if
if $c > 0$
$\text{CDYNAMIC}[x, y] \overset{atomic}{\leftarrow} \min(\text{CDYNAMIC}[x, y], c)$
end if
end for
end for
end for
end for
end for

Figure 5.24: Dilate obstacles in the moving obstacle map

5.3.8 Cost Table

The cost table is in Structure-of-Arrays (SoA) format, as per Figure 5.21. The update phase requires special handling for parallel access on the GPU, but in the initialization phase we can simply set the initial values using one thread per entry.

5.3.9 Summary of Initialization Phase

The main effort of initialization is in preparing the XYSL map and static and dynamic cost map lookup tables CSTATIC, CESTATIC, and CDYNAMIC. Once these are completed, the planner can begin generating and evaluating trajectories, as we see in the next section.

5.3.10 Planning Onto The Lattice

For planning onto the lattice, the cost table update is simple since there is no need to coordinate parallel update. Each trajectory has a unique tuple (s, ℓ, a) defining its path's endpoint in the lattice and the acceleration profile over the path. Since these are dimensions in the lattice, each trajectory is guaranteed to land on a unique cost table cell.

Optimizing Initial Paths Onto Lattice

Optimizing paths from the vehicle state onto the lattice proceeds similarly to paths within the lattice, as in Section 5.3.4, but each path has the same starting pose. The fifth-order spline (Section 3.5.1) optimization code is currently implemented only on the CPU.

Index Preparation

Several indices and sizes characterize each path and its family of trajectories. As noted under "Batch processing of concise data items" in Section 5.3.1, it is often better on the GPU to divide a processing pipeline into multiple kernels. Many

of these indices and sizes are used in several of the kernels, and as they may be expensive to recompute in each kernel invocation, we compute them once and store them in SoA format in global memory. This approach works because of the high memory bandwidth and latency hiding of the GPU programming model. These indices and sizes are described in the following:

- Each trajectory is evaluated by one thread, combining a path spline and an acceleration profile. In an array indexed by the thread identifier we store the path spline index and the acceleration profile index.
- Each path is sampled with a number of steps derived from its arc length. This number is computed once and stored in an array indexed by the path spline index.
- The starting time of the trajectory.
- The starting velocity of the trajectory.

Sampling of Paths

Once the initial paths are optimized they are sampled. One thread is allocated per path. The thread loads the spline parameters, the number of samples (which may be different for each path, depending on its length), and the start pose.

As we said in Section 5.3.1, memory accesses are fastest on the GPU when threads access consecutive memory locations. Given N paths, with p_i the ith path and p_{ij} the jth sample along path p_i, the x-coordinate of sample p_{ij} is stored at $x[jN+i]$, the y-coordinate at $y[jN+i]$, and likewise for the θ, κ, and s (arclength) coordinates.

Static Cost Evaluation of Paths

Cost function terms that do not depend on time are evaluated over all paths. The path indexing is as described in the previous item. Samples p_{ij} are loaded and evaluated. The (x, y) coordinates are discretized and indexed into the two static

cost maps CSTATIC and CESTATIC. We do not perform interpolation to smooth the texture.

Monitoring and diagnosing problems with the behavior of the planner is much more difficult when it runs on a GPU, due to the large volume of data and lack of ready access to debugger breakpoints and assertions. To assist in diagnosing, the cost function over the samples is evaluated one step at a time. At each step, it is tested against ∞, and if equal, an array $cause[i]$ is set to an ID unique to the cause. The causes identified are

- NaN - the path parameters diverged and the path did not integrate to a real value
- The curvature limits were exceeded
- The path is out of bounds of the static cost map.
- The CSTATIC entry is a fatal value.
- The CESTATIC entry is fatal.
- No cost term was ∞, but when summed together they overflowed the finite range and are treated as ∞.

These causes are used later in the visualization to show the operator which paths failed and why.

Once computed, the static cost for each path is stored in an array and used later to compute the total cost for each trajectory based on it.

Dynamic Cost Evaluation of Trajectories

Each trajectory is evaluated for just the cost function terms that depend on time, those terms depending on position but not on time having been evaluated in the previous section. A sequential implementation would be able to store the best trajectory as it was generated and evaluated, but with a parallel implementation

the synchronization requirements would be prohibitively expensive. Nor can we store all the trajectories, due to the large volume of trajectory data generated. With a GPU implementation we must be careful to store sufficent data to regenerate the trajectories after the best cost has been determined. Once the dynamic cost has been computed it is stored in an array indexed by trajectory identifier.

5.3.11 Planning Within The Lattice

Planning within the lattice is similar overall to planning onto the lattice - we evaluate the static cost terms on the paths, then the dynamic cost terms on the trajectories. The main difference is that the starting times and velocities for the trajectories must be extracted from the cost table before the dynamic cost terms can be computed.

The trajectories in the lattice are evaluated in order of station. Evaluating them in order of station guarantees that all trajectories incoming to a vertex have been evaluated before the outgoing trajectories, since the vehicle is constrained to always move forward. We could also evaluate in order of increasing time, but this would require moving back and forth between stations. This would reduce performance by requiring the path samples to be regenerated (since there is no room to store all of them), and unlike station-ordering it would not employ the regular memory access patterns that promote performance on the GPU.

The data items needed to initialize each trajectory are:

- Starting time for the trajectory.
- Starting velocity for the trajectory.
- Starting cost for the trajectory.
- Spline index for the trajectory, as in Section 5.3.10.
- Acceleration index for the trajectory, as in Section 5.3.10.

- Cost table index of the search graph vertex from which the trajectory originates, used after all trajectory evaluations have completed to trace back the best trajectory through the graph.

These are extracted in a kernel function that reads entries from the cost table at the desired station and prepares tables of data for the next kernel function.

Each (s, ℓ, a, t, v) vertex originates $n_a n_{sp}$ edges to other vertices, where n_a is the number of acceleration profiles used in the search, and n_{sp} the number of splines. So the number of trajectories outgoing at each station (ignoring splines that would go out of the station or latitude bounds of the lattice) can be up to

$$(n_\ell n_a n_t n_v)(n_a n_{sp}),$$

depending on whether any splines go out of the lattice bounds, the vertex is actually reachable, etc. One thread processes each of the $n_\ell n_a n_t n_v$ cost table entries at the station index, and initializes $n_a n_{sp}$ trajectories according to Table 5.2.

We ensure that consecutively-numbered threads read consecutive cost table entries, and write consecutive outputs. That is, threads are indexed by

$$i_{thr} = i_v + n_v(i_t + n_t(i_a + n_a(i_\ell + n_\ell s))),$$

where the station s is the same for all threads, and the threads range over all combinations of values $i_v \in (0, \ldots, n_v - 1)$, $i_t \in (0, \ldots, n_t - 1)$, and so on for each of the indices. The outputs are indexed similarly,

$$j_{out} = i_{thr} + n_{thr}(k_a + n_a k_{sp}),$$

so that consecutively-numbered threads write consecutive values at the same (j_a, j_{sp}) output pair.

Writing the data in this order ensures that in the subsequent trajectory evaluation phase (Section 5.3.10), consecutive threads will usually evaluate trajectories with the same acceleration profile and path indices, differing only in starting time,

i_{thr}			Thread index/cost table index(identical)															
0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	
k_a			Outgoing acceleration index															
0	0	0	0	0	0	1	1	1	1	1	1	2	2	2	2	2	2	
k_{sp}			Outgoing spline index															
0	0	0	1	1	1	0	0	0	1	1	1	0	0	0	1	1	1	
j_{out}			Outgoing index															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	
			Cycle at which output is written															
0	0	0	1	1	1	2	2	2	3	3	3	4	4	4	5	5	5	

Table 5.2: Sample thread and data indices for preparation of trajectories outgoing from a station after all cost table values at the station have been settled. The threads read consecutive cost table entries (top row) and write consecutive outgoing trajectory entries (second-to-bottom row).

velocity, and incoming accleration profile index. Table 5.2 gives an example of the thread indexing. In this example, three threads set up initial states for trajectories from a cost table with $n_\ell = 1$, $n_a = 3$, $n_{sp} = 2$, $n_t = 1$, and $n_v = 1$. Each thread reads from one cost table entry, and writes the data for all trajectories starting from it. At each iteration, the threads write consecutive entries.

Cost Table Update

The vertices in the search graph are in the form $(s, \ell, a, [t_i, t_{i+1}), [v_j, v_{j+1}))$, indicating the discrete station s and latitude ℓ of the vertex along the road, the acceleration profile index a of the last trajectory segment, and a time range $[t_i, t_{i+1})$ and velocity range $[v_j, v_{j+1})$ of arrival at (s, ℓ).

The cost table stores the best known cost of any trajectory to reach each vertex in the search graph, along with the actual time $t_f \in [t_i, t_{i+1})$ and velocity $v_f \in [v_j, v_{j+1})$ of the trajectory as it reaches the vertex. There is a many-to-one relationship of trajectories to the cost table entries they fall into, and since we are updating the table in parallel, we must take care to maintain consistency. The GPU only provides atomic operations on 4-byte integer quantities. There is no

way to lock a cost table entry and update the several related quantities we require. Therefore we must perform the update in three stages. We use the algorithm UPDATE-COST-TABLE, shown in Figure 5.25 and described in the following.

1. Using one thread per trajectory, each thread writes the cost of its trajectory to the table cell corresponding to its ending state, if its cost is lower. The built-in atomic memory operation `atomicMin()` is used to avoid race conditions. A global barrier then ensures that the table is completely updated before any thread executes the next step. On the GPU a global thread barrier can only be effected by all threads exiting the kernel function and the host invoking a new one.

2. Since multiple trajectories landing at the same cost table cell could have the same value, we must select one of them for its t_f and v_f values. Each thread/trajectory examines the lowest cost written at the previous stage. If it is identical to its own cost, it writes its index into a separate table. In the algorithm we use an `atomicMin()` to ensure that the lowest-index τ is retained when multiple trajectories have the same, lowest, cost. Since writing an integer is an inherently atomic operation, using the atomic minimum is not strictly necessary, and extracts a small cost in performance. However, doing so eliminates the non-determinism inherent in the plain integer store operation that can hamper repeatability useful in debugging. A global barrier ensures that the index of the winning trajectory at each table cell has been written before the next stage.

3. Each thread/trajectory again examines the winning index written in the previous stage, and if it is identical to its own, writes its t_f and v_f values into the appropriate cell, along with other data items used to reconstruct the final trajectory.

function UPDATE-COST-TABLE:
require: $ct[\,]$ – cost table
require: $\tau \in [0 \ldots n_{\mathrm{traj}})$ – indices of trajectories in update
require: $f(\tau)$ – cost table index of final state $(s, \ell, a, [t_i, t_{i+1}), [v_j, v_{j+1}))$ of trajectory τ
require: $c(\tau)$ – cost of trajectory τ
require: $cb[\,]$ – index of trajectory with lowest cost ending at cost table cell
require: $t_f(\tau), v_f(\tau)$ – actual final time and velocity of trajectory τ
require: $cv[\,]$, $ct[\,]$ – actual final time and velocity of lowest-cost trajectory ending at cost table index
forall trajectories τ ending at cost table index $f(\tau)$ **do**
    $ct[f(\tau)] \overset{atomic}{\leftarrow} min(ct[f(\tau)], c(\tau))$
end for
global thread barrier
forall trajectories τ ending at cost table index $f(\tau)$ **do**
    if $ct[f(\tau)] = c(\tau)$ **then**
        $cb[f(\tau)] \overset{atomic}{\leftarrow} min(cb[f(\tau)], \tau)$
    end if
end for
global thread barrier
forall trajectories τ ending at cost table index $f(\tau)$ **do**
    if $cb[f(\tau)] = \tau$ **then**
        $ct[f(\tau)] \leftarrow t_f(\tau)$
        $cv[f(\tau)] \leftarrow v_f(\tau)$
        and additional traceback data to reconstruct the final trajectory...
    end if
end for

Figure 5.25: Algorithm to update cost table from trajectory costs and final states

5.3.12 Extracting the Best Trajectory

Once all trajectories have been evaluated and the cost table contains the best known cost to reach each trajectory cell, we can search the cost table for the lowest cost-to-come plus final-cost and traceback through the parent pointers in the table for the trajectory. Since the cost table is relatively small, for simplicity's sake we copy it and the related data to reconstruct the path exactly back to the host memory, and the remainder of the planning algorithm is performed on the CPU.

5.4 Summary

In this Chapter we presented our planner implementation, describing the non-trivial implementation details needed to turn the theory of Chapter 3 and Chapter 4 into a working planner. We began with the overall system organization and described the order of operations in the planner. We got into the details of how the planner interfaces with the Tartan Racing system, particularly the control system. In this chapter we learned the principles of how to use the GPU effectively, and particularly how to use it for our planning problem. We made sure to describe how to efficiently construct the data structures needed to accelerate the whole planning effort, such as the various spatial and temporal maps.

Chapter 6

Evaluation

In previous chapters we have described the method and operation of our lattice planner, illustrating its features through demonstrations in a variety of situations. In this chapter we undertake a final, systematic evaluation of its capabilities. We evaluate our contribution in two major ways. First, we show that it works as claimed, producing reasonable plans that guide a robot through challenging driving scenarios. We demonstrate the planner working on a real robot and in simulation. Second, we compare the features and capabilities of our planner to similar work, showing that it advances the state of the art.

6.1 Experimental Results

In this section we describe several experiments we conducted with our planner, both on a real robot and in simulation. We used a real robot for normal driving scenarios at low speeds. For normal driving at higher speeds, and for emergency driving scenarios, we used a simulation. We also compare the performance increase of the GPU over the CPU in practice.

Many of our experiments revolve around merging. Merging is one of the most frequently executed complex driving tasks. The causes of a marge may include but are not limited to:

- Slow-moving traffic,
- Obstruction in the current lane,
- An upcoming exit or turn.

There may be plenty of time to prepare for the merge, or a suddenly perceived obstacle may require a rapid reaction. In the following we will demonstrate several scenarios that are variations on this theme.

Parameter	Value
Station increments	6
Latitude increments	14
Accelerations	9
Outgoing paths	$\tilde{4}0$
Time discretizations	1
Velocity discretizations	4
Total trajectories evaluated per cycle (actual)	$\approx 200\ 000$

Table 6.1: Parameters of the lattice used in the experiments

6.1.1 Experimental Configuration

We implemented our planner on an Nvidia GeForce GTX 260 containing 216 computing cores. We installed the Nvidia on a computer with an Intel Core 2 Quad processor. We used this same computer to run simulation experiments and experiments on our robot. In both cases the parameters defining the size of the search lattice were similar, shown in Table 6.1.

Robot Experiments

For robot experiments, the planner was run on Boss, our autonomous SUV. We ran the planner on a 10 Hz update cycle, providing curvature, velocity, and acceleration commands to a controller which converted them into steering and throttle actuation commands to the vehicle. Since Boss cannot be run on public roads, nor at speeds greater than 30 mph, our tests were limited to a small private roads with low speeds and our own traffic vehicles. The values we assigned to parameters controlling the size of the search lattice are given in Table 6.1. We found that these values could generate acceptable plans within the allowed planning latency.

Simulation Experiments

For simulation experiments, our planner ran at 5 Hz. This was necessary because at the higher speeds we used in simulation, we needed to use a larger static occupancy grid than we used for the lower-speed robot experiments. The static occupancy grid needs to be reformatted and copied from the Tartan Racing system to the GPU, which takes more CPU time than the rest of the planning cycle. An enhancement for future work would be to make this transfer more efficient.

6.1.2 High speed evasive maneuvers

A core claim of our work is that our planner has the novel ability to plan evasive maneuvers at high speeds where panic stopping is not viable and a response requires deciding quickly between a rapid merge into a neighboring lane at freeway speeds, or if that is not possible, venturing into an oncoming lane. In this section we show two scenarios illustrating our planner's response to an obstacle suddenly appearing in the robot's lane. The vehicle must judge whether it can complete a merge into the lane to its right or complete a maneuver into the oncoming lane.

In the first scenario, shown in Figure 6.1, the robot and other traffic are traveling initially at 55 mph, or $24.3m/s$. An obstacle suddenly appears in the robot's lane, only 65 meters ahead. There is no room to stop from 55 mph before reaching the obstacle. The planner must either merge into the right lane if it is able, or venture into the oncoming lane if it is open. In this case, a merge is possible, but first the planner must brake to merge into the gap in the next lane. It brakes for one second at $-7ms^{-2}$, reaching $17.3m/s$, and then re-accelerates just before it starts the merge to avoid being rear-ended by the vehicle behind it. It then merges back into its desired lane.

In the second scenario, shown in Figure 6.2, the right-hand lane is too dense with traffic to execute a merge when the obstacle appears, again at only 65 meters away. In this case, however, traffic in the oncoming lane will clear in time to for the robot to swing left around the obstacle. In this case, planner commences

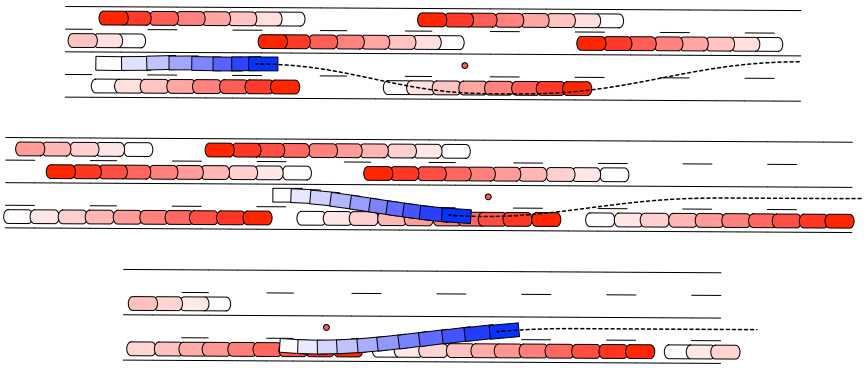


Figure 6.1: An obstacle (red circle) suddenly-appears, causing the planner to plan a merge into a neighboring lane. The thick dashed line is the plan at the final state sampled in each figure. The obstacle is at the same coordinates through all three images.

braking, though in this scenario it need not brake as hard to reach the gap behind the oncoming vehicle. It brakes alternately between $-7ms^{-2}$ (the hard deceleration limit $a^{\text{min}}_{\text{hard}}$) and $-1.5ms^{-2}$ (the soft deceleration limit $a^{\text{min}}_{\text{soft}}$). It slows from $24.3ms^{-2}$ down to $18.4ms^{-2}$ just as the vehicle in the nearest oncoming lane is alongside.

This is a difficult scenario which tests the limits of our planner. While our planner is usually able to find a plan where one would seem to be available, often a much better response is obvious but is outside the planner's search space. For example, the best response may be a long swerve with an initial sharp deceleration followed by an acceleration phase beginning before the swerve has completed. Since we only use constant acceleration profiles (not including the PD-controlled distance keeping acceleration, which is not tuned for emergency maneuvers), our planner's search space does not include any such action. For tight merges at high speeds, it may be necessary to generate an acceleration profile that specifically targets the gap between vehicles in a neighboring lane. Another reasonable response would be for the planner to accept that it will collide with the obstacle, and simply brake as hard as possible to mitigate the impact. Since our implementation assigns obstacles an infinite cost, it currently cannot plan such an outcome.

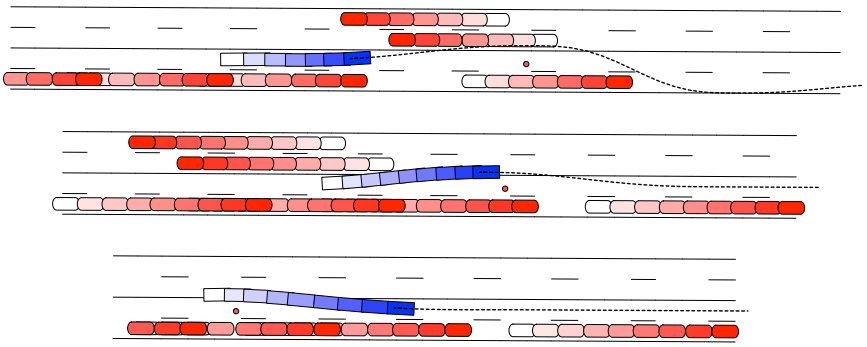


Figure 6.2: A suddenly-appearing obstacle causes the planner to plan to move into the oncoming lane, since there is no room to stop and there is no room to merge into the lane to the right.

Allowing the planner to plan explicitly for unavoidable impacts is a subject for future work. Regardless of these limitations, we are not aware of another planner in the literature that can make any response to these scenarios.

6.1.3 Merging into slower traffic

To test merging into slower traffic on short notice, we ran a simulated scenario on an oval track with two lanes, both in the same direction, and a speed limit of 55 mph. In the right lane we put slow-moving traffic, moving steadily at 26 mph and spaced between 30 and 40 meters apart. At four evenly-spaced points around the oval we placed static obstacles in the left lane. Each set of obstacles blocks more of the left lane than the previous one, requiring the robot to merge further into the right lane. The robot used the lane selection scheme LANE-BEHAVIOR, which uses the algorithm SELECT-LANE in Figure 4.14 to select a value for ℓ^{des}, the desired driving lane, then uses it to construct a lane cost profile like the one illustrated in Figure 4.4. As a result, it preferred to travel in the left lane to avoid being slowed by the traffic in the right lane. It would make deviations into the right lane only to avoid obstacles. Figure 6.3 shows a merge requiring only a small deviation. Figure 6.4 shows a merge requiring a larger deviation into the neighboring lane. Here, the planner has kept a slightly larger gap between itself

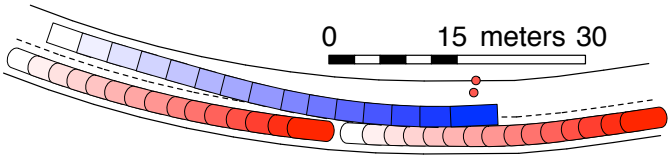
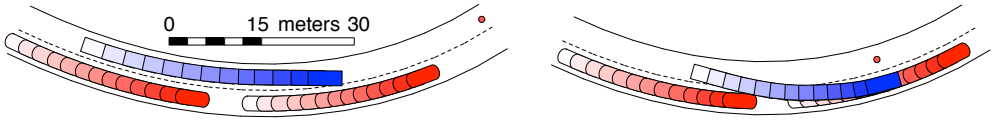


Figure 6.3: Small merge into neighboring lane due to an obstacle intruding 1.25 meters into the lane, with 3 meters remaining. This figure represents 2.8 seconds elapsed time.



(a) Deceleration, 0 - 2.4 s

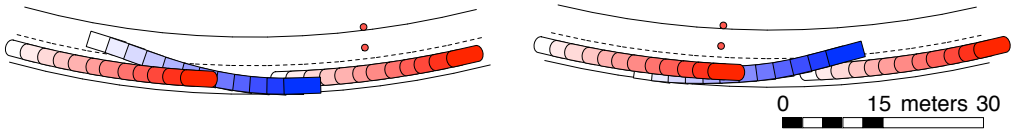
(b) Merge, 2.4 - 4.8 s

Figure 6.4: The obstacles reach 1.9 meters into the left lane, leaving 2.3 meters open. The planner merges all the way into neighboring lane

and the obstacle than the previous case. In Figure 6.5 the obstacle almost completely blocks the left lane, and in Figure 6.6 it blocks the left lane completely and intrudes into the right lane. In these cases the planner uses the shoulder to get around the obstacles. We ran this scenario for fifty minutes with no accidents, covering 45 km with approximately 300 merges.

6.1.4 Freeway lane-changing

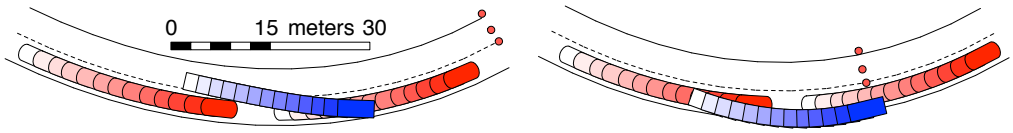
In Section 4.7 we described three policies to effect freeway lane changes. In this section we present a sample scenario using one of these policies, which was



(a) Deceleration, 0 - 2.2 s

(b) Merge, 2.2 - 4.4 s

Figure 6.5: The obstacle leaves 25 cm open in the left lane



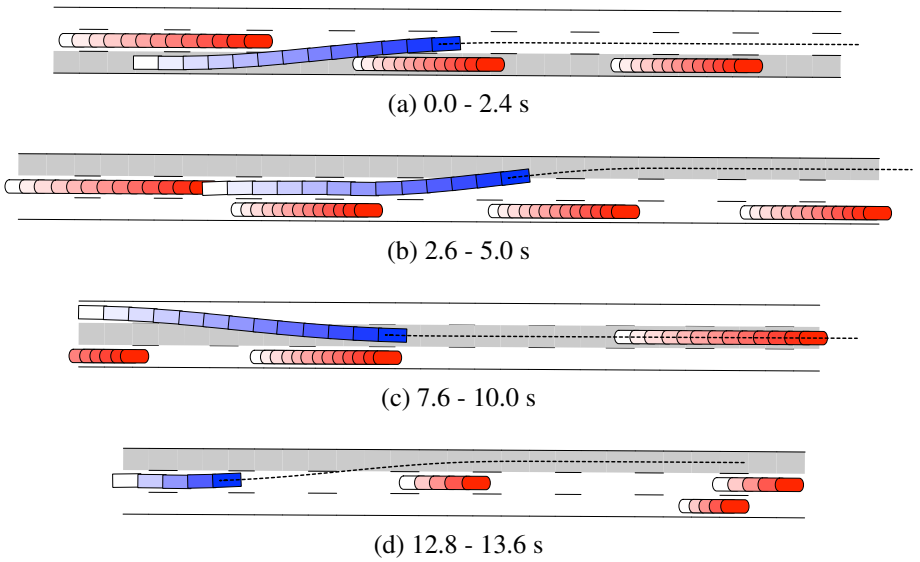
(a) Deceleration, 0 - 2.4 s

(b) Merge, 2.4 - 4.8 s

Figure 6.6: The obstacle blocks the left lane completely and intrudes 40 cm into the right lane

illustrated in Figure 4.17, and which we refer to here as LANE-COMBO. This method uses the behavioral algorithm SELECT-LANE (Figure 4.14) to select a desired driving lane. The chosen lane is assigned the lowest cost in $c^{\text{lane}}_{\text{static}}$, but other lanes are only a small amount higher. The trajectory planner implicitly chooses a lane by balancing between the cost assigned to each lane in $c^{\text{lane}}_{\text{static}}$, the cost of nearby dynamic obstacles in $c^{\text{obs}}_{\text{dynamic}}$, and the final cost Φ (Equation 3.2). There are higher costs in $c^{\text{obs}}_{\text{dynamic}}$ in front of and behind other vehicles to penalize following or leading too close, and Φ penalizes taking longer to reach the end of the planning horizon, for example by staying in a lane with slow traffic. The pattern in $c^{\text{lane}}_{\text{static}}$ allows the lattice planner to drive in a different lane from the lane selected by SELECT-LANE when it is unsuitable for reasons that SELECT-LANE did not take into account.

Figure 6.7 shows a driving scenario using the LANE-COMBO scheme. The lane selected by SELECT-LANE is shown in grey. The speed limit is 24.3 ms^{-1} (55 mph), traffic in the right lane travel at 10 ms^{-1}, and vehicles in the middle lane travel at 15 ms^{-1}. In Figure 6.7a, the vehicle starts in the right lane, as though it has just entered the freeway. Here, we have tuned SELECT-LANE to delay changing lanes when following slower traffic. As a result, the trajectory planner chooses to override the behavioral layer's selected lane, and change into the middle lane. In Figure 6.7b SELECT-LANE chooses the left lane because the trailing vehicle in the middle lane is too close behind. The trajectory planner accedes, and then SELECT-LANE (Figure 6.7c) chooses the middle lane since the trailing vehicle is now far behind. Finally, in Figure 6.7d SELECT-LANE changes



(a) 0.0 - 2.4 s

(b) 2.6 - 5.0 s

(c) 7.6 - 10.0 s

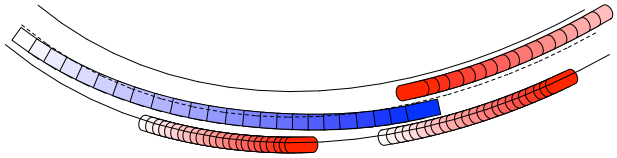
(d) 12.8 - 13.6 s

Figure 6.7: Freeway lane changing using LANE-COMBO. The selected lane is colored in grey.

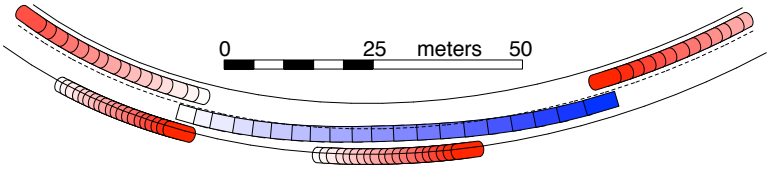
to the left lane to pass the next vehicle ahead – this time early enough to avoid being overridden by the trajectory planner.

We ran 30 minutes of freeway driving experiments for LANE-COMBO covering 30 km with approximately 100 lane changes. We ran an additional hour combined for LANE-BEHAVIOR and LANE-AUTO covering a total of 60 km and approximately 200 lane changes. There were no accidents in any of these experiments, and the planner showed safe behavior throughout.

This example shows that by planning exhaustively out to a long planning horizon and using a complex cost function, our planner can plan and execute complex merge calculations while maintaining safe and robust behavior. Whereas prior works take a brittle top-down approach, with higher-level planners commanding lower levels to change lanes based on limited summary information, our planning is bottom-up, taking all data into account before making a decision.



(a) Pass first bicycle, 0 - 4.7 s



(b) Pass second bicycle, 4.7 - 8.6 s

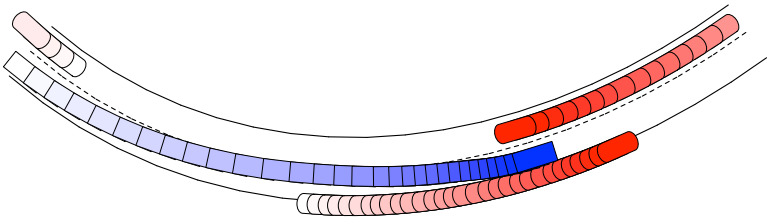
Figure 6.8: The robot passes two bicycles in a row with oncoming traffic, merging between the bicycles before passing the second one.

6.1.5 Merging with oncoming traffic

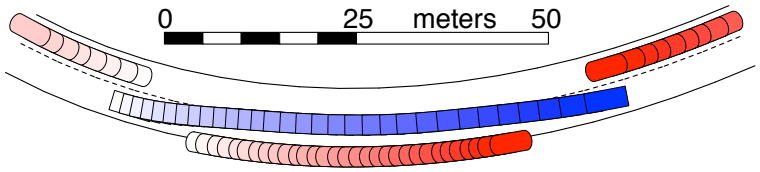
On an oval course with one lane in each direction, we show the planner driving around bicycles in the presence of oncoming traffic. We placed four bicycles around the oval with speeds of 5 m/s, 5.5 m/s, 6 m/s, and 6.5 m/s. The bicycles were represented as normal vehicles displaced laterally 2.5 meters from the lane center. There is room for the robot to pass the bicycles without hitting oncoming traffic, but the lateral obstacle cost dilation is such that the robot prefers to wait until oncoming traffic has cleared and give them a wider berth.

The changing positions of the bicycles and the oncoming traffic led to several unique scenarios. Figure 6.8 shows two bicycles being passed in a row, with a merge between them to avoid a second oncoming vehicle.

Figure 6.9 shows the robot passing one bicycle, but in this case an oncoming vehicle is detected after the robot has already accelerated into the oncoming lane. The robot must slow down, merge back behind the bicycle, wait until the oncoming vehicle passes (Figure 6.9a), and then re-initiate the passing maneuver, merging back in front of the bicycle before the next oncoming vehicle arrives (Figure 6.9b). In these simulations the planner assumes that the oncoming vehicle is



(a) Cancel pass of bicycle, 0 - 6 s



(b) Resume pass of bicycle, 6 - 12 s

Figure 6.9: The robot attempts to pass a bicycle, cancels the attempt when it detects an oncoming vehicle, and then resumes passing when it detects enough open space in the oncoming lane.

on rails and will not slow down or move aside to avoid a collision with the robot.

We ran this scenario for 1.5 hours, covering 65 km. The robot passed a bicycle with oncoming traffic present over 100 times. In some experiments we shortened the robot's perception horizon below what would be prudent for the speed it was traveling. The result was that an oncoming vehicle could appear while the robot was passing a bicycle, leaving no time to merge back into the travel lane. The planner would in that case stop. Since the traffic vehicles are not responsive to the robot's actions, they would continue to travel through the robot. These were the only collisions we observed, but we consider them acceptable since there was plenty of time for the traffic to slow down or possibly move aside, which a real driver would do. In fact, it is what our own planner would do, as we show in the next section.

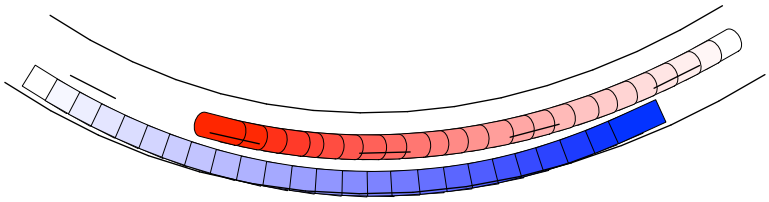
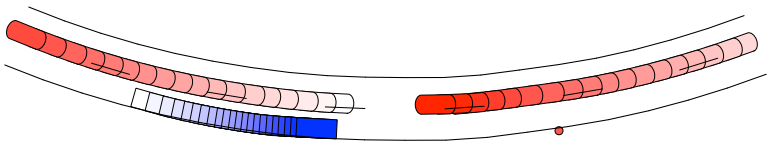
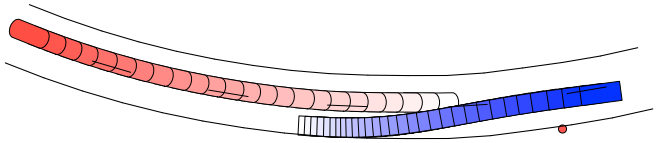


Figure 6.10: The robot moves aside to avoid an oncoming vehicle encroaching into its lane and returns back to the center of the lane after it passes by (5 seconds elapsed).



(a) Move aside and wait, 0 - 4.4 s



(b) Resume circumvention of obstacle, 4.4 - 10.3 s

Figure 6.11: The robot cannot move right to avoid oncoming traffic encroaching into its lane because of a static obstacle in its lane ahead. It slows and waits until the oncoming traffic has passed. The vehicle slows from 30 mph (portion shown: from 21 mph) down to 9 mph and then accelerates back up to speed.

6.1.6 Misbehaving oncoming traffic

We demonstrate our planner's ability to smoothly avoid oncoming vehicles that encroach into its lane. Figure 6.10 shows the robot in a simulated scenario, where a oncoming vehicle is straddling the center line and encroaching into our robot's intended path. The planner smoothly moves aside, driving over the shoulder line, and returns to the center of its lane after the oncoming vehicle passes.

Figure 6.11 shows a more complex scenario with two oncoming vehicles straddling the center line and a static obstacle on the right-hand side of the lane. The robot must slow to wait for the oncoming vehicles to pass before swinging out to avoid the static obstacle. We ran the simulated scenario for an hour with mis-

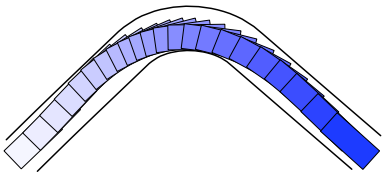


Figure 6.12: With a cost function term penalizing lateral accelerations, the planner automatically slows the vehicle in anticipation of a sharp turn. The distance between successive samples shows the deceleration coming into the turn and acceleration beginning near the apex of the turn. Samples are spaced at 300 ms intervals.

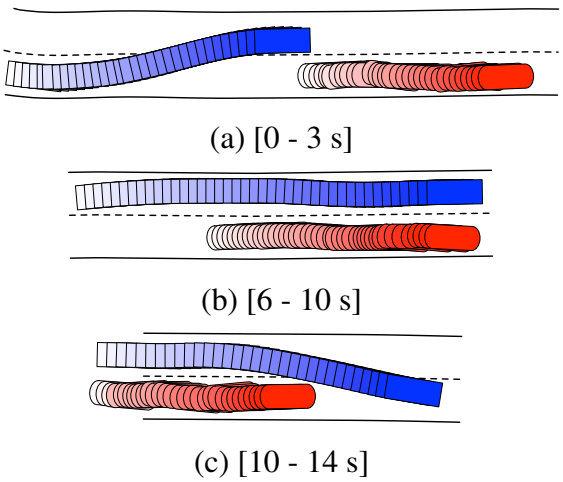
behaving oncoming traffic and static obstacles placed by the side of the road. The robot covered 40 km and moved aside to avoid an oncoming vehicle approximately 400 times, including 200 instances where it had to negotiate around a static obstacle on the right while a vehicle was oncoming. The planner's behavior was consistent, with no accidents or near misses.

6.1.7 Real robot tests

On the robot we performed a more limited range of tests, due to safety and logistical constraints. On the robot, we did experiments in lane changing, distance keeping, and obstacle avoidance. Videos are available from the author for all of these scenarios. In this section we present figures illustrating a few of our tests.

Figure 6.12 demonstrates that the planner can choose an appropriate speed to round a tight corner. In this case the planner penalizes trajectories with higher lateral accelerations up to a hard limit of $c_{\text{dynamic}}^{\text{lataccel}} = 0.3g$. At the first sample in the figure, the robot is traveling at 33 km/h. It slows to 13 km/h with a lateral acceleration of 0.11 g to round the corner at the tenth sample and accelerates back to 32 km/h at the last sample.

Figure 6.13 demonstrates a passing scenario executed on the robot, using the LANE-AUTO lane-changing scheme, that is, the right lane is given a slightly lower cost to traverse than the left lane, encouraging the robot to stay in the right lane. The robot prefers to travel at approximately 35 km/h, but a human-driven vehicle



(a) [0 - 3 s]

(b) [6 - 10 s]

(c) [10 - 14 s]

Figure 6.13: In a test conducted at 30 km/h, the robot passes a slower-moving human-driven vehicle in the right lane by (a) moving into the left lane, then (b) remaining there until it passes the other vehicle, and (c) finally merging in front of it. Samples are spaced at 100 ms intervals.

traveling at 20 km/h is ahead of it in the same lane. Since the bonus awarded for driving further by the final cost coefficient k_s (Equation 3.3) exceeds the slight penalty for driving in the left lane, the planner selects a trajectory that moves into the left lane, passes the slower vehicle, and merges back in front of it.

Over a variety of tests during the development of the planner, we ran the robot in autonomous mode 73 separate times, for a total of 2 hours 7 minutes, covering 49 km and reaching a maximum speed of 44.8 km/h.

The safety and robustness of a system is partially demonstrated by consistent behavior that does not require a human to intervene. The safety protocol used by Tartan Racing was to have two people in the vehicle. The "safety driver" sat in the driver's seat and was prepared to take control of the vehicle instantly. The "software operator" watched the planner's status on a laptop and was able to anticipate either that the robot would be able to execute an upcoming higher-risk maneuver safely, or that it was about to do something dangerous.

Due to logistical constraints, in our experiments we employed only a safety driver who had to take control during higher-risk maneuvers where a software op-

erator would have been needed to watch the vehicle's perception state and plans closely and tell the driver that a takeover was not needed. Out of those experiments, we took control of the robot for safety reasons 9 times. Later review showed that most of these were unnecessary.

- In early testing the vehicle habitually weaved back and forth about 1.5 meters across the road due to a software flaw. In one instance the driver took control when approaching a pole close to the side of the road while the vehicle was weaving to the right. Reviews of the logs later showed that the vehicle would not have come close to hitting the pole.

- In early testing the planning task crashed and was automatically relaunched, leaving the robot without a plan for about one second. The bug was fixed.

- While tuning system control delays in early experiments, weavy runs were terminated twice when the safety driver could not be sure that the vehicle was not diverging. Later reviews should the vehicle remained under control.

- A police cruiser showed up on the course, and the safety driver took control out of caution.

- Three times during passing experiments the robot was passing a vehicle while taking a tight corner. Since a software operator was not available to confirm that the neighboring vehicle was properly perceived and the plan was safe, the safety driver took over.

- The safety driver took over when a goose aggressively protecting its nest came onto the road. The perception system is not well able to perceive small animals as lethal obstacles, nor for that matter to identify them as animals at all. Our cost function does not include a prediction of the likely motion of an animal, nor any terms to shape desirable responses when an animal is on the road.

Examining the above list, we conclude that after early testing phases to flush out bugs and tune the control system, the vehicle did not perform any unsafe maneuvers that required a human to intervene. All takeovers were precautionary and could have been avoided given the manpower to follow the Tartan Racing safety protocol.

6.1.8 Tuning Parameters

In some of the preceding experiments, the cost function weightings had to be tuned in order to demonstrate the desired behavior. These tuning changes were small in number and size, and a future high-level behavior could tune them depending on its assessment of the traffic situation. Aside from the various lane-changing behavior parameters for the simulation experiments in Section 6.1.4 and the robot experiments in Section 6.1.7, we only changed a few cost function parameters for the experiments presented in this chapter.

The first set of weights we tuned related to lateral acceleration (Section 4.5.2). These were the lateral acceleration soft limit $k_{a\perp}^{\text{soft}}$, which we tuned lower in robot experiments as a precautionary measure, and the linear penalty $c_{a\perp}^{\text{max}}$, which we lowered for the evasive maneuvers we presented in Section 6.1.2. We believe it would have been easier to tune a single value of $c_{a\perp}^{\text{max}}$ to work in multiple situations had we used a quadratic rather than linear penalty, which more closely reflects the actual constraints we wish to apply to lateral accelerations. We used lower-magnitude values of the longitudinal acceleration soft limits $a_{\text{soft}}^{\text{max}}$ and $a_{\text{soft}}^{\text{min}}$ (Section 4.5.2) on the robot, to save on fuel and brake wear.

The second set of weights we tuned were related to the lane cost potential function $c_{\text{static}}^{\text{lane}}$ (Equation 4.2). Specifically, for the bicycle-passing scenario (Section 6.1.5) we lowered c_a^{opp}, the constant cost term for opposing lanes, and c_b^{trav}, the linear cost term for travel lanes. These changes encouraged the planner not to stay behind bicycles, but rather to perform passes that would require it spend relatively long periods away from the lane center or encroach into the oncoming lane. A high-level behavior could be coupled with a bicycle-detecting perception

Search Phase	GPU Time	CPU Time	Speedup
Plan trajectories from source pose onto lattice	2 ms	12 ms	6
Update all paths in lattice	0.1 ms	4 ms	40
Plan all trajectories coming out of a single station	2 ms	42 ms	21
Whole planning cycle	45 ms	650 ms	15

Table 6.2: Time taken on the CPU and GPU for stages of the planning cycle

system to make this adjustment on a case-by-case basis.

In summary, while most cost function weights changed from their initial assignments during the development of the planner, over repeated experiments and adjustments we settled on values that worked in a wide variety of scenarios, with just a few minor changes needed to evoke specific desired responses for demonstration purposes. Even these few adjustments could be eliminated by adding new high-level behaviors and additional refinements to the form of the cost function.

6.1.9 Performance

To compare the performance improvement gained by using the GPU, we also implemented the planner on the CPU only and run it in simulation on a single core. Table 6.2 displays the time taken in each phase of the search by each of the GPU and CPU. The GPU provides a considerable speedup overall even accounting for the fact that only one core is used in the CPU implementation, although the GPU is much faster at certain tasks. Both implementations are reasonably well-optimized, so this comparison is strongly suggestive of the relative merits of the platforms for our algorithm.

6.1.10 Summary

In this section we have shown that our planner works in practice as in theory, generating reasonable plans in a variety of scenarios. In the next section we will

compare our planner to previous works and show that its dense sampling of trajectories in space and time advances the state of the art, particularly in emergency evasive maneuvers.

6.2 Comparison to the State of the Art

Having shown in the previous section that the planner performs as desired, we turn to comparing our planner to other works. In most cases we do not have access to implementations of other planners for use even in simulation, let alone to run on our robot. Therefore we cannot run head-to-head comparisons of our planner against others in order to see which performs best in a variety of scenarios. Instead, we identify planner features that are necessary to robustly handle several critical driving scenarios, and compare other planners based on their exhibition of said features. We will show that our planner exceeds other works on features vital to robust driving performance.

Figure 6.14 lists features we have identified as necessary for a robust driving planner. Each column indicates how well various related works provide the features we have identified.

6.2.1 Description of Planner Features

In the following we expand on the features listed in Figure 6.14. We have chosen these features to conduct our comparative evaluation because we believe they are necessary to a robust planner. In the following we describe these features and justify why we believe they are important.

Hard real-time response

In a dynamic and hazardous environment, a planner must have the ability to generate a set of candidate trajectories, and this set must be very likely to contain a safe alternative. Further, it must do so in an amount of time guaranteed to be small. A

	Deliberative; explicitly evaluate candidate plans	Contain multiple lateral shifts in one trajectory	Scale performance w/ bigger parallel computers	Simultaneous planning in latitude/longitude/time	Hard real-time response	Feasible with limited vehicle acceleration	Real-robot tests; consistent performance
Proposed lattice planner	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
Rule-based/ command-fusion	No	No	No	Some	Yes	Yes	No
CMU UC	Yes	No	No	Yes	Yes	Yes	Yes
Stanford UC	Yes	No	No	Some	Yes	Yes	Yes
MIT UC	Yes	Yes	Some	Yes	No	Yes	Some
Karlsruhe/ Werling	Yes	No	Some	Yes	Yes	Yes	Some
Karlsruhe/ Ziegler	Yes	Yes	Some	Yes	Yes	No	No

Figure 6.14: Comparison of our spatiotemporal lattice planner (first column) features and capabilities against related work. From left: "Rule-based/command-fusion" is a class of driving systems including Navlab-related work[97] through to systems fielded at the DARPA Urban Challenge[86, 104]; "CMU UC"[45] is the planning stack used for on-road driving by the Tartan Racing team[28] in the 2007 DARPA Urban Challenge(DUC); "Stanford UC"[71] is the Stanford DUC entry; "MIT UC"[53] is the DUC entry from MIT; "Karlsruhe/Werling"[105] and "Karlsruhe/Ziegler"[106] are recent contributions similar in spirit to our lattice planner.

	Emergency stop behind stationary obstacle, no room to pass	Merge into traffic to go around slower vehicle	Merge into traffic to go around slower vehicle, then merge back in front to make exit	Double lane change with merge required to avoid sudden appearance of obstacle
Hard real-time response	Safety critical	Helpful	Helpful	Safety critical
Contains multiple lateral shifts in one trajectory	Not necessary	Not necessary	Helpful	Safety critical
Feasible with limited vehicle acceleration	Safety critical	Helpful	Helpful	Safety critical
Deliberative approach; explicitly evaluate candidate plans	Not necessary	Helpful	Helpful	Safety critical
Simultaneous planning in latitude/longitude/time	Not necessary	Helpful	Helpful	Safety critical

Figure 6.15: Traffic scenarios that justify the features we use to evaluate our planner against the state of the art in Figure 6.14.

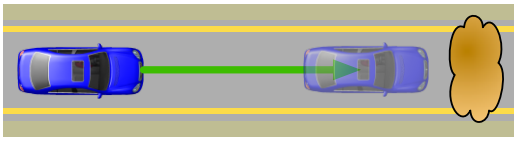


Figure 6.16: An emergency scenario where the robot must come to a complete stop. A planner must have a hard real-time guarantee to respond correctly to this situation.

soft real-time guarantee, where the planner must produce satisfactory results in a small amount of time *on average*, is insufficient. Some authors, such as [53], have attempted to finesse the issue, but we do not believe it can be avoided. Figure 6.16 and Figure 6.17 illustrate emergency scenarios where the planner must be able to guarantee a hard real-time response to obtain a safe outcome.

Contain multiple lateral shifts in one trajectory

In complex emergency scenarios, making the correct high-level planning decision can require a detailed evaluation of all possible maneuvers, one of which may require at least a double lane change. Figure 6.17 illustrates an emergency scenario

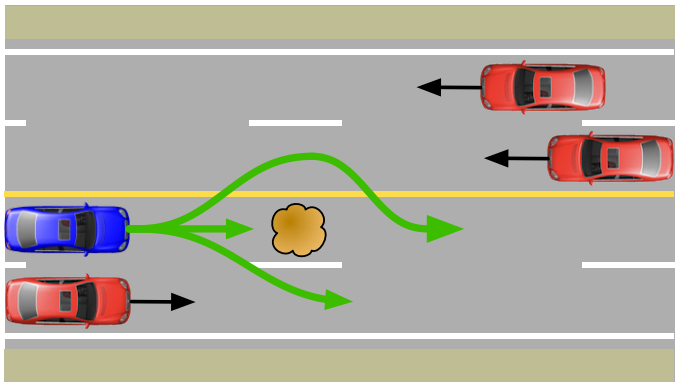


Figure 6.17: An emergency scenario that would require a double lane change to plan optimally, as well as a hard real-time response. The robot (blue) is at the left, heading right, in the inner lane. An obstacle has suddenly appeared on the roadway and there is no time to stop. A plan to swerve right must verify that there is time to brake in order to merge behind the vehicle(red) next to it. A plan to swerve left must verify that there is time to vacate the opposing lane before hitting one of the oncoming vehicles. If neither swerve can be performed safely, it may be better to brake as hard as possible before hitting the obstacle.

that requires a full evaluation of a double lane change maneuver in order to determine the best course of action. Our planner can also generate trajectories with additional consecutive lateral maneuvers if necessary.

Feasibility with limited vehicle acceleration

We have mentioned (Section 3.4.1) that grid-based approaches to constructing a search space can fail when nonholonomic constraints are present. Some authors, e.g. [106] finesse this problem by assuming that the vehicle will be able to reach grid points by using high accelerations, or by using widely spaced grid points and limiting their experiments to scenarios where rapid maneuvering is not required. We believe that an effective planner must be able to function on a vehicle with low accelerations even when undertaking challenging maneuvers.

Deliberative planning approach

We maintain that a *deliberative* approach, one that explicitly and realistically simulates the likely outcome of a plan before committing to it, is necessary to a successful autonomous driving robot. Deliberative approaches stand in contrast to *reactive* approaches like the subsumption architecture[15] that wire sensor readings directly to actuators, taking actions without explicitly predicting the outcome in a world model. Some authors[49][105] describe a deliberative approach as "reactive" if it is performed rapidly enough and on a relatively limited horizon, but we are careful to make the distinction between the two. We propose that the most effective and reliable planner is the one that explicitly evaluates the greatest number of plans in the most thorough and realistic way. Figure 6.17 shows that in complex emergency situations, a planner that can consider a wide variety of plans and reason explicitly about the likely outcome is better able to pick the safest response.

Simultaneous planning in spatial and temporal dimensions

Constraints on computing drive some authors to pursue a staged approach to planning, where for example the planner will first generate and commit to a path calculated to avoid static obstacles, and then search for a velocity profile over the path that will also avoid dynamic obstacles, for example [43]. With such an approach there will be scenarios where a feasible plan exists but will be discarded early. Increasing the coupling between spatial and temporal dimensions helps the planner accurately plan maneuvers like merging into moving traffic Figures 6.18 and 6.19.

Scale performance with bigger parallel computers

We believe, as mentioned in the section justifying a deliberative planning approach, that the more plans evaluated by the planner, the better. As we showed in Chapter 1, recent performance increases in computing operations per second

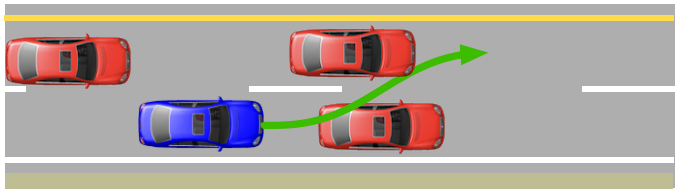


Figure 6.18: The robot must be able to merge into traffic to go around a slow-moving vehicle. This scenario is not safety-critical, but it illustrates the benefits of a planner with the features we propose.

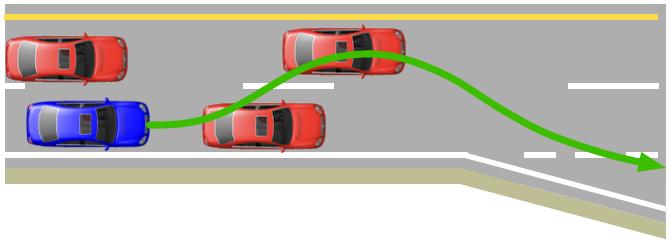


Figure 6.19: The robot must merge into traffic to go around a slow-moving vehicle, as in Figure 6.18, but this time mindful that it must merge back soon to make an exit.

per dollar have come primarily from increasing parallelism, and this trend will continue into the foreseeable future. Therefore, a planner that is ready to translate increasing parallelism into increasing plan quality will be better able to meet new challenges in the future.

Real-robot tests

Testing a planner on a real robot in a variety of scenarios reveals weaknesses that could remain hidden if it were only tested in simulation. Even when simulation test scenarios have been carefully selected to uncover failure modes, significant flaws that can only be revealed in an encounter with the real world may still lurk within. Significant re-thinking may be required once a planner has left the lab and suffered the shock of an encounter with the real world. Access to a robot with sufficient perception capabilites and the space to run it is a significant obstacle to mounting a true demonstration of a planner's effectiveness. For that reason

planners that have only be tested in simulation must be taken seriously, but with caution.

Summary

We have described several of the planner features that we believe are important to focus on when evaluating competing approaches to planning for autonomous driving. In the following we use these features to compare our planner to the representatives of the state of the art.

6.2.2 Comparison to Rule-based Approaches

Compared to unstructured driving tasks such as roving around a desert looking at rocks, driving on public roads requires adherence to a complex set of rules. The rules of the road must be in some way reflected in the planner so that it generates compliant behavior.

This need for the planner to follow the rules of the road has led some researchers[9, 36] to advocate that the planner should be organized around a decision tree of rules that analyze the current road situation into a taxonomy of perhaps thousands of individual cases, with a prescribed response for each. To these authors, the complete taxonomy of driving scenarios serves not only as a road map for an implementation, but a guarantee of correct behavior as well. We do not make a detailed evaluation of our work against this kind of planner since we are not aware of a reasonably fleshed out implementation that embodies this approach.

Rule-based planning can also be seen as a way of coping with the apparent intractability of the search space compared to the available computational resources. When it is not possible to evaluate a large variety of plans, rule-based approaches use heuristics to narrow down the search space, discarding large swathes of the space so that only a small set of plans needs to be evaluated in any detail. Sukthankar's MonoSAPIENT[97] is an example of this rationale for taking a rule-based approach. All planners apply some degree of *a priori* narrowing of

the search space to turn the continuum of plans into a finite set of discrete options. MonoSAPIENT is a finite state machine with major driving modes such as following a vehicle or changing lanes, with hand-coded case analysis to transition between states while considering safety and the rules of the road. It does not explicitly anticipate the outcome of a planning decision, rather, the rules are carefully designed so that the outcome should be safe. By contrast, our planner explicitly evaluates the outcomes of all candidate plans and so offers a stronger guarantee of safety.

6.2.3 Comparison to Command Fusion Approaches

A *command fusion* approach identifies an abstract set of actions and deploys independent reasoning objects to assess each possible action. Each reasoning object assigns a weight to each action, and an arbitrator sums the weights and picks the best action. The essential proposition of an arbitration approach is that it is possible to abstract a small set of local actions out of the whole space of possible plans, such that each action can be evaluated by looking at just a few aspects in isolation.

Sukthankar's PolySAPIENT[97] is an arbitration framework that begins with an abstract set of actions such as *speed-up*, *slow-down*, *change-left*, *change-left-and-slow-down*, etc. A set of reasoning objects independently evaluate each action according to its own understanding of what the action means and issue a numerical vote indicating the desirability of each action. Reasoning objects might include a reasoner that pays attention only to the car in front, voting on all actions based only on the likelihood of a collision with that vehicle - voting to slow down if the vehicle ahead is slowing down, or even to change lanes if there is no room to slow down enough. Another reasoning object tracks the car ahead in the right lane, another to the car behind in the right lane, and likewise for the left lane. An arbiter tallies the votes from all the reasoning objects and picks the action with the highest score. An action can be effectively vetoed with a score of $-\infty$. As with MonoSAPIENT, the reasoning objects do not explicitly evaluate the outcome of each action. After an action has been selected by the arbiter, a trajectory generator

turns the action selection into an actual vehicle motion. To achieve robustness with this approach it is vital to ensure that the generated trajectory is consistent with the implicit expectation each reasoning object has about what an abstract action like "change-left" means and how it will actually be carried out.

Complex plans such as the double lane change shown in Figure 6.17 are difficult to address with the arbitration-based approach, since it requires very precise evaluation of all plans ahead of time, while the reasoners deal with only abstract actions that are not completely spelled out before they are decided upon. Sukthankar's PolySAPIENT, for example, reaches the pinnacle of its performance with a single lane-change emergency maneuver that requires a merge into traffic ([97] Section 7.3.5). Voting weights and tuning parameters had to be carefully set to obtain this behavior.

More realistic action sets have been used in open-country navigation applications without dynamic obstacles, such as constant steering angles[89, 50]. At the DARPA Urban Challenge, the Team AnnieWAY and CarOLO entries[104, 86] also used a command-fusion approach based on constant curvature arcs. These required complex heuristics to regulate the interpretation of the arc paths in the context of driving on a narrow road with moving traffic.

Although command-fusion approaches are evidently more capable than rule-based approaches, complex emergency maneuvers at high speeds and in the presence of moving obstacles seem to be an insurmountable challenge for them as well. None of the command-fusion based systems we have examined have demonstrated an ability to handle the double-lane change maneuver illustrated in Figure 6.17. They would also scale poorly with increasing parallelism, since the number of practically distinguishable constant steering commands is relatively low compared to the degree of parallelism readily available even today.

6.2.4 Comparison to the CMU Urban Challenge Planner

The Carnegie Mellon team (also known as the Tartan Racing Team[101]) won the DARPA Urban Challenge. The work presented in this thesis is inspired by their

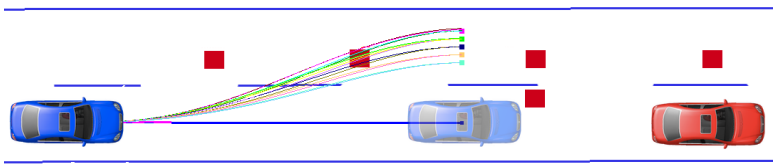


Figure 6.20: The Tartan Racing lane changing method is unable to avoid obstacles in its current lane while a change is pending.

lane planner. They sample states along the roadway, as our planner does, and used a cubic polynomial spiral to solve a boundary value problem connecting the current vehicle state to each sampled state[45]. They ran several acceleration profiles over the path to generate multiple trajectories, each of which was evaluated, with the best one chosen.

In this sense their planner is similar to phase one of our planner, where paths are generated to go from the vehicle's current position onto the planning lattice. It is there that they stop, and where our approach continues to generate more complex plans. The Tartan Racing planner offers a real-time guarantee, since it evaluates a discrete and limited number of plans with predictable computational complexity. It would not scale well with additional parallel computing. While each candidate path and trajectory can be evaluated independently, the existing path endpoint sampling strategy already covers the roadway fairly well. There is not much benefit to sampling even more lateral points, except when changing lanes.

We had access to the Tartan Racing planner, and performed experiments to highlight its limitations compared to our planner. Figure 6.20 shows a scenario where the vehicle must swerve around in an obstacle in the current lane while a lane change is pending. Since only one path sample is generated for the current lane in this case, the vehicle must come to a full stop and revert to its unstructured planning mode to get around the obstacle. Our planner has no problem with this scenario.

Figure 6.21 shows that the short planning horizon of the Tartan Racing planner can cause it to make suboptimal decisions. Much as our lane selection algorithm

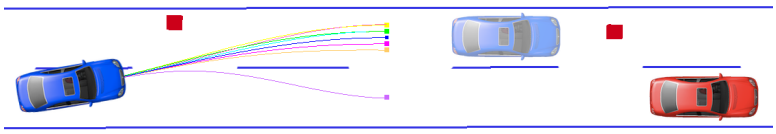


Figure 6.21: The Tartan Racing lane changing method changes lanes based on the locations of other vehicles, but does not account for static obstacles beyond the stopping distance.

SELECT-LANE (Figure 4.14) does, the Tartan Racing planner ignores static obstacles when planning a lane change, attending only to the locations and velocities of moving obstacles. In the scenario of Figure 6.21, a static obstacle is beyond the planning horizon, so the vehicle initiates a lane change only to come to a full stop in the new lane. Our planner is able to plan out to a much longer horizon. While it may change into a lane containing an obstacle beyond the planning horizon, in practice the planning horizon is so long that it has much more time to change back without coming to a stop.

6.2.5 Comparison to the Stanford Urban Challenge Planner

The lane planner for the Stanford Urban Challenge entry was similar in spirit to the CMU entry. A set of trajectories parallel to the lane, each at different lateral offsets from the road center line, are rolled out from the current vehicle state. This allows moving around small obstacles, but it cannot handle complex emergency maneuvers. As with the Tartan Racing planner, increased parallel computation would not add much benefit since athere are only so many usefully distinct trajectories that swerve once at the beginning and then run out at the same offset along their entire length.

6.2.6 Comparison to the MIT Urban Challenge Planner

MIT's entry[61, 53] into the Urban Challenge was unique in their use of a rapidly-exploring random tree (RRT) approach for all planning tasks, not only in unstruc-

tured environments such as parking lots, but also in simply driving straight down a lane.

The RRT is very versatile and would appear to be promising. Any trajectory plan can theoretically be generated by an RRT. Further, as one allows the algorithm to run for unbounded time, the probability that a valid plan will be found converges to 1, if a feasible plan exists at all.

In practice there are a few problems with using RRTs for a real-time application in a relatively low-dimensional environment. While the MIT authors take pains to ensure that the planner always maintains a valid plan with a safe final state, the RRT cannot guarantee that it has examined a well-dispersed set of trajectories. Experimental results reported in [53] seem to show that the planner was unable to drive at a consistent speed along a softly curved road with no traffic.

6.2.7 Comparison to other regular sampling approaches

The "Karlsruhe/Werling"[105] and "Karlsruhe/Ziegler"[106] columns in Figure 6.14 have been published since the DUC by researchers affiliated with Team AnnieWAY, though not based on the approach used by that team in the DUC.

The "Ziegler"[106] approach is closest in spirit to our own - the authors use a regular lattice defined in SL-coordinates along the road center line. In a first stage they sample plans going from the vehicle onto the lattice, and then complete the plan using regularly sampled transitions within the lattice. In this respect their approach is identical to ours. Where it differs is that they sample time and velocity as discrete points rather than intervals as we do. As we noted in Section 3.4.1 this places undue strain on the system: either the vehicle must be capable of large accelerations to make the transition from one grid point to another, or else there must be an infeasibly dense state sampling and large action set joining them in order to ensure a feasible trajectory can be found. They have demonstrated their planner in a few simulated merging and passing scenarios but we do not know of a demonstration on a real robot.

The "Werling"[105] approach resembles the first stage of our planner. Our

approach is to sample paths terminating in a set of regularly sampled endpoints, then traverse each path with a variety of acceleration profiles to produce trajectories. In contrast, the Werling approach generates a set $\mathcal{T}_{\text{lat}}$ of samples of the lateral position of the vehicle with respect to the road center line as a function of time, and independently samples a set $\mathcal{T}_{\text{lon}}$ of the longitudinal position also as a function of time. All pairs of lateral and longitudinal samples with matching end time T are then paired to generate a set of trajectories $\mathcal{T}_{\text{lat}} \times \mathcal{T}_{\text{lon}}$. The result is a set of single swerve actions that can be evaluated for comfort, adherence to the center line, and collision with other vehicles. Unlike our approach, theirs does not append additional swerve actions and so cannot generate a double lane-change evasive maneuver. However, they can sample the $\mathcal{T}_{\text{lon}}$ endpoints in order to promote specific behaviors such as distance keeping, merging, and velocity keeping. They also have a scheme to promote stability across replan cycles by ensuring that the search space always includes the previous cycle's plan. Our planner has a similar feature enabled by the static global pose of the lattice, which we described in Section 4.2. They report having used the planner on their autonomous vehicle JUNIOR with no traffic, and simulation results showing robust behavior driving in well-behaved traffic. They do not report real or simulated vehicle speeds, so we cannot fully evaluate the robustness and performance of their approach.

6.3 Summary

In this chapter we have demonstrated not only that our planner works in a variety of normal and emergency driving scenarios, but that it advances the state of the art, handling emergency situations better than other planners in the literature.

In the next chapter we draw final conclusions, summarize the contributions of this thesis, and discuss directions for future work.

Chapter 7

Conclusions

> Waste makes haste. For time is fleeting. A rolling stone is worth two in the bush.
>
> Robocop

7.1 Conclusions

Our planner is able to generate reasonable plans in real-time for a variety of traffic situations. The planner can use a GPU to exploit the parallel structure of the search space and significantly reduce the planning latency.

The cost function plays an important role in shaping the final behavior, and more work is necessary to develop cost functions for more complex behaviors, such as cruising outside of other vehicles' blind spots, moderating distance from obstacles based on velocity, or passing on the highway in the presence of oncoming traffic. We may also need the cost function to distinguish between emergency maneuvers and normal maneuvers, so that an emergency maneuver, however brief, would only be selected when no normal maneuver is available, however long.

An obvious next step in the development of the planner is to investigate more sophisticated acceleration profiles while maintaining execution efficiency. Using a constant acceleration over the course of the path can lead to execution errors, since vehicles typically cannot change acceleration abruptly. At low speeds, constructing paths using a cubic polynomial spiral while staying strictly within steering rate constraints overly limits the maneuverability of the vehicle. A refinement to the types of actions considered may be necessary at low speeds.

We have not yet looked at reachability in the lattice. If there are large portions of the lattice which are never part of the solution, they may be trimmed in order to increase trajectory density in other parts.

7.2 Contributions

Before an autonomous passenger vehicle can drive safely and efficiently in traffic, we must overcome many difficult motion planning challenges. A planner for an autonomous passenger vehicle must generate a variety of complex behaviors executed over varying time scales while maintaining safety and comfort for the passengers and other users of the road. In this thesis we present a novel combina-

tion of planning ideas into a low-level trajectory planner for autonomous highway driving. The low-level planner is responsible for making planning decisions at a frequency on the order of 10 Hz, with a planning horizon on the order of 10 seconds. Our approach increases the range of behaviors that the low-level planner can intentionally produce over the state of the art, as well as the robustness of its interaction with higher levels of the planning stack.

7.2.1 Search Space Decomposition

We contribute a decomposition of the search space for on-road driving into a planning graph that uses a resolution-equivalent grid approach to include both spatial and temporal dimensions in the search graph while respecting the kinematic and dynamic constraints on a typical automobile. The structure of our graph allows us to sample the search space with enough coverage to generate acceptable plans over a wide variety of driving scenarios, while sampling sparsely enough to generate plans in real-time.

Our method overcomes two hurdles. First is the usual curse of dimensionality, where the number of paths within a search space increases exponentially with the number of dimensions in the space. The second problem is that the presence of nonholonomic constraints in the domain necessitate that when an action changes one dimension of a state, other state variables must also change as a side effect. The use of a standard fixed grid imposes additional constraints on the values of state variables along the path that are hard to satisfy without greatly increasing the resolution of the grid. To resolve nonholonomic constraints without increasing the grid resolution, we adapt the locations of grid points on the fly, similar to the concept of resolution-equivalence enunciated in the Incremental Search Engine[99], used in Hybrid A*[25], and the anonymous method by Barraquand and Latombe[11].

7.2.2 Cost Scheduling

It is a well-known problem in planning that the models used by hierarchies of planners tend to be inconsistent with one another, leading to undesirable behavior. In this thesis we contribute a solution to this problem. Our planning framework assumes more responsibility at the lower levels of the planning hierarchy by increasing the scope and number of plans that are evaluated at that level. Higher-level planners responsible for making decisions with longer-term scope, such as selecting which lane to be in for an upcoming turn or exit, can communicate with the low-level planner by modifying the cost function in order to encourage desired behavior, rather than directly specifying goals that may be impossible to attain.

Our planner searches a richer space of possible plans than previous planners, and by doing so is able to support a richer language of communication with higher-level planners through the cost function it uses to select trajectories. Not only does our planner mitigate the problem of infeasible commands being issued by higher levels by taking on more responsibility, it further reduces this problem by changing the way the layers interact. Our planner does not directly accept commands from higher levels for it to implement. Rather, higher levels simply manipulate the cost function in order to nudge our planner towards the desired result, for example, changing into a specific lane when feasible.

7.2.3 Parallelization

The sequential planners typically used in search-based planning, such as A*, are not readily parallelizable. A problem-specific analysis of the search space is necessary to reveal opportunities for parallel evaluation of search nodes. Our decomposition of the search space for driving exposes considerable scope for parallel computation. We show how our planner can be implemented on a graphics processing unit (GPU) vector-parallel processor to achieve a significant parallel speedup.

7.2.4 Implementation

We contribute an implementation of our planner in an autonomous passenger vehicle. The planner is implemented using an Nvidia GPU and integrated into the Tartan Racing software system developed for the 2007 DARPA Urban Challenge. An important part of our implementation is a cost function that balances the competing desires for brisk driving, comfort, and safety. The cost function must contend with the phenomenon of *relentless optimization* whereby a planner tends to select plans that fail in the presence of errors in modeling or execution. We validate our implementation with real-world experiments.

7.2.5 Virtual Terminal State

We contribute a method for selecting the terminal state of the plan in a repeated planning task with no fixed endpoint. Whereas other driving planners choose a fixed goal distance or time at each planning cycle[105, 29], we use an adaptive planning horizon that minimizes current costs combined with an estimate of future costs. This allows our planner to adapt to changing road conditions by changing the length of the plan.

We adapt the standard formulation to evaluate a final state for a control or planning problem, which is to sum a cost functional over a trajectory and a cost function for the state at the final time, as in [56] Chapter 10.6:

$$L(\tilde{x}_{t_F}, \tilde{u}_{t_F}) = \int_0^{t_F} l(x(t), u(t))dt + l_F(x(t_F)).$$

7.3 Future Work

In the course of this research project we have identified many directions for future work. Some are particular to our approach, and some are broader in scope.

7.3.1 Search Space Improvements

First we review directions for future enhancements of the structure of the search space to speed the planner or improve the quality of its plans.

Planning at the Limits

The evasive maneuvers described in Section 6.1.2 need to be more drastic. Our approach generates trajectories by combining polynomial spirals and constant accelerations, then filters out those that the vehicle cannot execute, for example due to steering rate limits and limits on lateral acceleration. However, the generated trajectories do not fully exploit the abilities of the vehicle. For example, a trajectory that is just within the lateral acceleration limits of the vehicle at one portion is very likely to exceed them at some other points and to fall short of them at other points. The net effect is that trajectories which are feasible throughout their full length may fall very short of the vehicle's ability at some points. In an emergency situation, we need the planner to exploit the full abilities of the vehicle. Future work should look at ways to solve this problem, perhaps by planning directly in terms of the performance envelope of the vehicle, that is, the dynamic, or acceleration constraints.

Our planner is not effective at making sharp swerves at low speeds, for example when changing lanes from a stopped position immediately behind another stopped car. This limitation is due to similar causes to those that limit our ability to fully exploit the vehicle's performance abilities for high speed evasive maneuvers. At low speeds we need to plan in terms of the relevant performance limits of the vehicle, i.e., the speed at which the vehicle can turn its steering wheel.

Finally, the use of different trajectory forms at different speeds raises the question of how to smoothly switch between plans made in different forms as the vehicle changes speed.

Number of Swerves

The planner currently demands a lot of computational power to explore a large range of candidate plans at each cycle. Future increases in computational power will mitigate the impact of these demands. However, more work will be needed to reduce the total amount of computation done by the planner. The planner as currently constituted can plan multiple swerves in a single trajectory. It is arguable that this many swerves are never necessary. More work should be done to determine whether limiting the plan complexity to two swerves would be sufficient for all conceivable situations. If so, the planner could do less work.

Long acceleration profiles

The cost function 3.2 favors higher-speed trajectories than lower ones, so we include the acceleration policy index as a dimension in the search space in order to maintain plan diversity (Section 3.4.5. Despite this measure, long emergency braking maneuvers that start from high speeds and require more than two consecutive trajectory segments through the lattice to come to a complete stop may be eliminated from the search. This and similar special cases yet to be identified need to be addressed in a principled way.

7.3.2 Behavioral Improvements

To demonstrate the feasibility of our planning approach, we contributed an implementation and a sample cost function to rank candidate plans. The cost function is largely independent of the search method, although some implementation details may need to be taken into account to ensure efficiency. Since the cost function encodes our preferences for how the vehicle should behave in all conceivable situations, much more experimentation in a variety of scenarios, and a great deal of cost function tuning is needed. Methods for automated parameter tuning from expert examples, such as the LEARCH algorithm[95], may be a useful tool.

Automated parameter tuners do not answer the question of what parameters

should exist. For example, are binary cost penalties on proximity to obstacles, speed limits, etc., sufficient, or should we use more intermediate levels? Should we add additional penalties for passing near an obstacle at higher velocities? These and similar questions need to be identified and answered.

The point of diminishing returns on increasing the size of the search graph is yet to be determined. How many more acceleration profiles should we add, if computational resources are available? Should we change the size of the lattice based on the speed of the vehicle? If so, how? How small should we make the lateral discretizations?

In order to continue making definite progress, future work should catalog traffic scenarios in more detail and outline an acceptable range of desired responses. This research area has matured to the point where it is necessary for new contributions to be evaluated against an objective list of agreed-upon performance criteria.

Our planner treats other traffic as though it were on rails, unable to change velocity, and completely unyielding. This applies even to traffic following our own vehicle, and for future states several seconds away. In real life, vehicles react to each other's movements. A full multi-agent search seems intractable, but we suspect that simple heuristics expressed through the cost function might be sufficient. In this work, for example, moving obstacles project an infinite cost into the dynamic cost map for the full planning horizon several seconds into the future. Lowering that cost to a finite level after a few seconds, and predicting that other vehicles will veer to the right and slow down rather than travel straight at the same speed, would allow the planner to generate evasive maneuvers that nudge into oncoming lanes while oncoming traffic is still several seconds away and would have time to react, in essence creating a self-fulfilling prophecy for the behavior of other vehicles.

7.3.3 Planner Architecture

The control approach described in Section 5.2.1 introduces a discontinuity in the control loop by generating new plans from the perceived vehicle state. An approach that some have found to be more effective is to "plan from the plan"[102]. That is, initialize the planner state from the vehicle state at planner startup and from then on ignore the actual vehicle state in the planner, depending on a path tracker such as one of those described by [96] to keep the actual vehicle state close to the planned state.

Appendix A

Overview of the GPU Architecture

Multi-core graphics processors(GPUs), developed most notably by Nvidia and AMD, are very different from CPUs. While both GPUs and contemporary CPUs are multi-core computers, and both are equally capable of solving decidable problems, they are designed to solve different types of problems quickly.

In this appendix we describe the major architectural features of the GPU, and most importantly, why some of the peculiarities of the GPU architecture will continue to be common, so that developing algorithms that take advantage of them is likely to be a worthwhile effort. We first contrast the two architectures before delving into the details of the GPU.

A.1 Major Differences between CPUs and GPUs

Contemporary personal computers are typically available with Intel or AMD processors containing between two and six traditional processor cores. These cores each typically run one sequential thread of execution at a time, and devote large amounts of chip space to running that single sequential thread of execution as quickly as possible. Speculative execution, branch prediction, a provision of excess functional units, analysis of instruction-level parallelism, and code translation are a few of many optimizations developed to squeeze performance out of a sequential thread of execution. Each of these consumes chip space. The functional units are designed to compute complex operations quickly, with diminishing returns in performance for each additional transistor. As opportunities to increase performance in sequential execution by adding transistors have dwindled, chip designers have resorted to utilizing the space available for additional transistors on a die by adding copies of the entire CPU.

GPUs also use many cores, numbering in the dozens or even hundreds. In contrast to contemporary CPUs, GPUs make no effort to execute any single thread quickly. Rather, they attempt to maximize the aggregate throughput of many threads. Hundreds or thousands of threads running the same algorithm must be available to execute in parallel for this strategy to be effective.

Figure A.1: Comparison of transistor allocation to functional components in traditional CPU architecture vs. Nvidia GPU, reproduced from [75]

Figure A.1 illustrates the broad design features that CPUs and GPUs incorporate to reach their respective performance goals. Whereas the CPU allocates large numbers of transistors to on-chip cache and control functions, the GPU allocates a relatively small number of transistors to cache and control, leaving the majority of chip space available for arithmetic logic units(ALUs), which actually compute the results. To achieve the maximum theoretical benefits requires an understanding of the architectural features of the processor.

A.2 Chip Architecture

Figure A.2 illustrates the architecture of the Nvidia GPU. The processor is composed of a variable number of *multiprocessors*, each of which is composed of exactly eight *scalar cores*. Each scalar core can execute one instruction per clock cycle, though complex instructions such as division and built-in transcendentals may take longer. The eight scalar cores within each multiprocessor share a small cache of programmer-addressable local memory and each scalar core has a large number of registers to support the private storage needs of hundreds of threads. The multiprocessor has just one instruction issue unit, so each of the eight scalar cores execute the same instruction at the same time, albeit on different data. Multiprocessors can only share data by storing to and reading from the global device memory. Higher performance GPUs are obtained mainly by increasing the num-

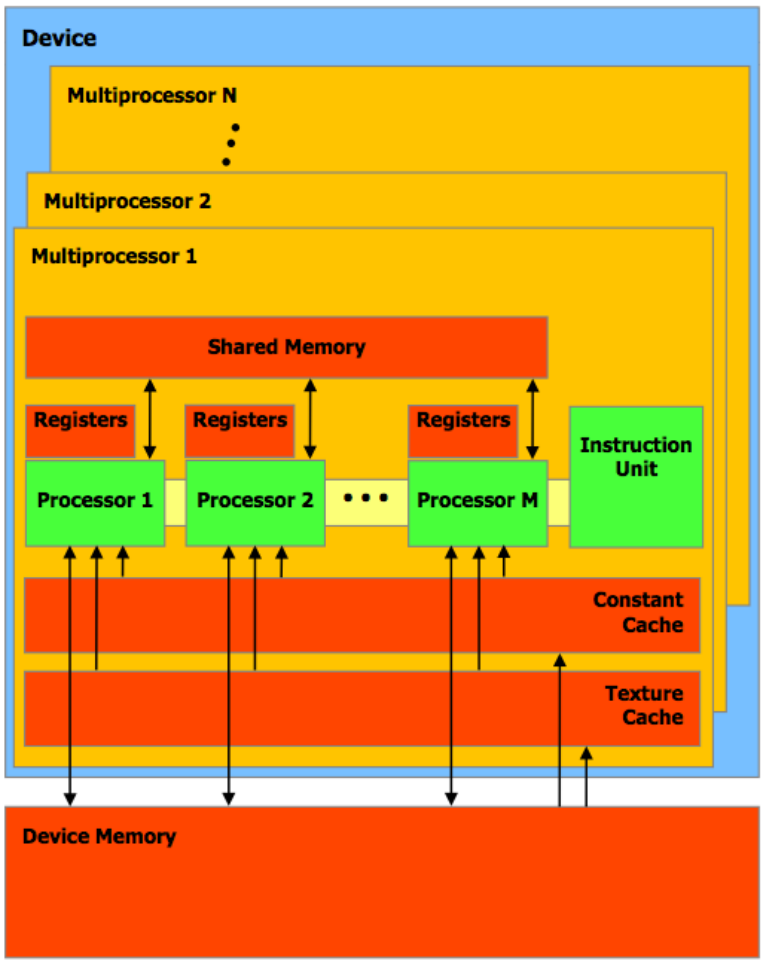


Figure A.2: Architecture diagram of nvidia GPU, reproduced from [75]

ber of multiprocessors on the chip.

A.3 Compute Unified Device Architecture

Early tools for employing graphics processors in general-purpose computing, an effort broadly termed General-Purpose GPU(GPGPU), were difficult to use and came with many restrictions, requiring the programmer to frame their algorithms as special programs for shading pixels, with debilitating restrictions on access to data and especially a lack of ways to synchronize between different operations. GPU manufacturers have been developing their graphics processors into more capable, general, and accessible parallel computing platform. One of these efforts is Nvidia's Compute Unified Device Architecture(CUDA).

Several multi-core processors share important architectural features with the Nvidia GPU, so we shall describe the particulars of the Nvidia architecture as concretely as possible in order to make it easier to understand, and then show the similarities it shares with other multi-core platforms.

CUDA consists of syntax enhancements to C++, with an accompanying compiler and a runtime library. It works beneath the graphics layer and leaves the programmer to program in C with a few simple additions to the language to utilize the parallel features of the processor. AMD, another supplier of GPUs, is developing a similar effort using their own Stream SDK[4]. Additionally, a portable language for heterogeneous computing using CPUs and GPUs known as OpenCL[39] is newly available. At the moment, CUDA is the most accessible and the most advanced, so we will be demonstrating results using CUDA.

In CUDA, threads are organized into *blocks* which may contain up to 768 threads. All threads in the same block may synchronize with each other using a barrier operation, but threads from different blocks may not synchronize. Each of the threads within a block is addressed with a unique three-dimensional index (x, y, z). Threads can retrieve their indices using a hardware instruction. Blocks are organized into a grid, and like threads, are uniquely numbered by the schedul-

ing hardware. Exactly one grid of thread blocks is run on the GPU per *kernel* invocation from the host CPU. Figures A.3 and A.4 show the core syntax of CUDA. Code running on the CPU invokes parallel functions called *kernels* to be run on the GPU. Each thread is designated to handle a small piece of the work. The one-dimensional convolution code shown in these figures illustrates the use of the programmer-controlled cache memory, also called *shared memory*. It also demonstrates the synchronization of threads within a block. The function shown in Figure A.3 runs on the GPU. Threads with the same ***blockIdx*** can access the same shared memory locations and synchronize with the built-in __syncthreads() barrier primitive. Since each element of the kernel and image arrays will be accessed many times and by different threads, they are loaded into the shared memory. Then, each thread computes one element of the result and stores it. Figure A.4 shows the code run on the host CPU. This is normal C++ code that runs in the usual way. The programmer must explicitly schedule data to be copied between buffers allocated on the host and GPU. The special ⋘⋙ syntax is used to invoke a GPU "kernel" function with the grid and block size layout within the brackets.

A.4 Features Common to GPUs

In the previous section we described the Nvidia GPU and the CUDA programming model. In this section we argue that the interesting features of the Nvidia GPU are also present in other platforms, such as AMD's Radeon graphics processors and Intel's upcoming Larrabee[93], and that their architectures are driven by the same design forces. The Nvidia GPU with its CUDA language extension is currently the most advanced and accessible platform for general-purpose parallel computing, but efforts made to develop algorithms and approaches will generalize to future platforms.

```
// 1D convolution with CUDA.
// This function runs on the GPU.
__global__ void convolve1d( float *image, int imgsize,
    float *kernel, int ksize, float *result ) {
    const int ksize2 = ksize / 2;

    // Index of this thread block's portion of the input image.
    const unsigned int img0 = blockIdx.x * (blockDim.x - ksize2*2);

    // Cache entire kernel and portion of image in block-shared memory.
    __shared__ float kcache[ MAXKSIZE ], imgcache[ MAXTHREADS ];

    // Load image cache, one thread per element, including padding.
    const int imgidx = img0 + threadIdx.x - ksize2;
    if( imgidx < 0 || imgidx >= imgsize ) imgcache[ threadIdx.x ] = 0;
    else imgcache[ threadIdx.x ] = image[ imgidx ];

    // Load kernel from global memory into shared cache.
    if( threadIdx.x < ksize ) kcache[threadIdx.x] = kernel[threadIdx.x];

    // Synchronize all threads in block; ensures data loaded into cache
    __syncthreads();

    // One thread responsible for each cell of result.
    // Some threads do nothing here; they exist only to load data above
    if( threadIdx.x >= ksize2 && threadIdx.x < blockDim.x - ksize2 ) {
        float r = 0;
        for( int i = -ksize2; i <= ksize2; ++i )
            r += kcache[i + ksize2] * imgcache[i+threadIdx.x];
        result[img0 + threadIdx.x - ksize2] = r;
    }
}
```

Figure A.3: First part of the CUDA code sample showing a parallel one-dimensional image convolution operation. This "kernel" runs on the GPU.

```
int main( int argc, char *argv[] ) {
    // input: img, imgsize, kernel, ksize

    const int blocksize = MAXTHREADS, cells_per_block = blocksize - ksize - 1;
    const int gridsize = (imgsize + cells_per_block - 1) / cells_per_block;

    // Allocate room on the graphics card to store a copy of the image.
    void *img_device; cudaMalloc( &img_device, imgsize*sizeof(float) );
    // Copy img from host (CPU) to device (GPU).
    cudaMemcpy( img_device, img, imgsize * sizeof(float), cudaMemcpyHostToDevice );
    // Similarly: allocate space for result and kernel on GPU, copy from host...
    void *result_device; cudaMalloc( &result_device, imgsize*sizeof(float) );
    void *kernel_device; cudaMalloc( &kernel_device, ksize*sizeof(float) );
    cudaMemcpy( kernel_device, kernel, kernelsize * sizeof(float), cudaMemcpyHostToDevice );

    // Call CUDA convolution code.
    // gridsize blocks will be run, of blocksize threads each.
    convolve1d<<< gridsize, blocksize >>>
        ( img_device, imgsize, kernel_device, ksize, result_device );

    // Allocate space on the host for the result, and copy from GPU.
    float *result_host = malloc( imgsize*sizeof(float) );
    cudaMemcpy( result_host, result_device, imgsize * sizeof(float), cudaMemcpyDeviceToHost );
    // output: result_host
}
```

Figure A.4: Second part of the CUDA code sample showing a parallel one-dimensional image convolution operation. This code runs on the "host", or CPU, and interacts with kernel code (Figure A.3) through the CUDA API.

Data-Parallel Operations

The performance achieved by GPUs comes from their devotion of large numbers of transistors to computational rather than control elements, as illustrated in Figure A.1. The control elements are responsible for instruction decoding, handling branches in the code, calculating addresses for operations on memory, analyzing code for instruction-level parallelism, and more. In order for the GPUs to keep these computational elements, or ALUs, supplied with work, the control elements must accomplish more with less. The simplest way to do this is by having each instruction cause the same operation to be performed simultaneously on multiple data elements. This is easy to do in many graphics applications, where rendering an image involves performing the same operations on many pixels.

An operation that performs the same computation on several pieces of data at once is termed *data-parallel*. Data-parallelism stands in contrast to *task-parallelism* where multiple threads of control may simultaneously perform entirely different functions. In order to gain the benefits of the GPUs for motion planning, it is necessary to develop algorithms that can use data-parallel operations.

Data-parallel computers typically organize data elements into fixed-size vectors. In CUDA for example, data-parallel operations are performed on vectors of 32 elements. Figure A.5 illustrates. Part (a) of the figure shows a normal program using scalar variables. Part (b) shows the same program running simultaneously on several copies of the variables organized into vectors, denoted `<v>`. Control flow is accomplished by creating a new vector variable that contains the results of a logical test performed on independent vector elements. Computations for elements corresponding to positions that tested false are still performed, but their results are not stored. With appropriate hardware support, the program could skip lines 3 or 4 entirely if `mask1` had all the same value.

In CUDA, the operations appear to be computed by independent threads. However, threads are grouped into *warps* of 32 elements. Threads within a warp execute the same instruction at the same time, but warps may run completely independently of each other. Figure A.6 illustrates this idea with an abbreviated

```
1: a = C[d];
2: if( a == 0 )
3:   d++;
4: else e--;
5: f += b * e;
```

(a)

```
1: <a> = C[ <d> ];
2: <mask1> = (<a> == 0);
3: <d> = ( <mask1> & (<d>+<1>))
         | (~<mask1> & <d>);
4: <e> = (~<mask1> & (<e>-<1>))
         | ( <mask1> & <e>);
5: <f> += <b> * <e>;
```

(b)

Figure A.5: Comparison of (a) a program using familiar scalar variables with (b) explicit use of vector-parallel operations. Each operation is performed element-wise on the vector variables, and constants are also vectors. Note that identifier `C` is an array, but not a vector variable.

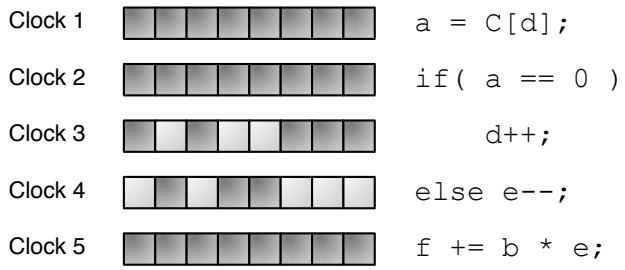


Figure A.6: Serialized execution of threads in an eight-thread warp on the GPU

warp of 8 elements. Each vector element is represented as a local variable for one thread, but threads in the same warp run the same instruction. When the code branches, only the threads that take the "true" branch run, simultaneously. When those threads reach the end of their branch, the threads that take the "false" branch run until the branches merge, when all threads can run again. If all threads were to take the same branch, then no time would be wasted with the alternate branch, an improvement over the mask-vector concept of Figure A.5.

Modern CPUs typically have vector instruction sets and vector registers in addition to their traditional scalar instructions and registers. For example, the Intel

SSE and PowerPC Altivec instruction sets can operate on vectors of 4 elements. However, the CPU vector instructions cannot perform operations on vector elements conditionally, nor can they use a vector element as an index for a memory access. That is, traditional CPU SIMD instructions lack support for the instruction on line 1 of Figure A.5(b), where the vector elements of `<d>` are used as indices into the array `C[]`. They also lack support for line 2, where the results of a conditional expression evaluated on each element of a vector are stored in another vector. Nor do they have an efficient built-in way of performing the mask operation expressed in lines 3 and 4.

Note that CUDA's warps are equivalent to the explicit use of operations conditioned on mask registers shown in Figure A.5. AMD GPUs, IBM's Cell[81], and Intel's Larrabee all depend for their high performance compared to CPUs on the conditional vector-parallel operations just described. The motivation for vector-parallel operations from a performance perspective is that they allow more work to be done with the same number of clock cycles and transistors devoted to instruction decoding, out-of-order instruction issue, and address translation. The challenge for the software developer and algorithm designer is to find ways to compose algorithms using vector-parallel operations. To achieve maximum performance using vector operations, we must be careful to organize our computation to ensure that all elements of each vector branch in the same direction as often as possible.

Memory Architecture

In this section we contrast the memory architecture of contemporary GPUs and CPUs and their effect on program performance. We then attempt to anticipate which aspects of the GPU architecture are likely to persist.

In traditional CPU architectures, cache misses are extremely detrimental to performance. All execution stops until the desired information is retrieved from memory further away from the CPU. The most common way to mitigate this problem is by using larger caches. Another way is to build hardware that can switch

instantly to another runnable thread when one thread is stalled. For example, Intel's Hyper-Threading feature allows one secondary thread to be run simultaneously with the primary thread. When the primary thread is forced to wait, the secondary thread may continue running. This can provide a speed boost to some applications.

GPUs use a comparatively small amount of cache memory, and rely on the programmer to write algorithms that use hundreds of threads. The GPU schedules threads in hardware at a fine grain, so that when a thread stalls waiting for a memory access to complete, another thread can be run immediately. If enough threads are available, then the ALUs can be kept working even when threads must frequently wait for memory operations to complete. This is the major rationale for the GPU architecture - if an algorithm can muster enough threads to keep the ALUs busy, then the work done may be maximized, even though individual threads may have a higher latency than they would on a sequential processor.

Memory bandwidth and latency limitations require many operations to be performed on each piece of data loaded from memory in order to achieve peak performance. Algorithms that run best on the GPU therefore must intersperse memory accesses with a significant amount of work. Algorithms that frequently permute global memory are not suitable for the GPU.

CPUs use a cache memory structure that automatically stores the values in recently-accessed memory close to the processor in anticipation that they will be used again soon. In place of these, the *shared memory* of the Nvidia GPUs requires the programmer to explicitly determine when data will be transferred between the large, global memory from the small, fast memory near the processor.

Memory bandwidth is the aggregate amount of data that can be transported between main memory and the processor per unit time. GPUs have a considerably higher memory bandwidth than CPUs. The Intel Core 2 Duo, for example, has a theoretical bandwidth of 8.5 GB/s, whereas the Nvidia GTX 260 has a memory bandwidth of 112 GB/s. The higher memory bandwidth of the GPU comes as a result of a wider data path. In current designs, the peak bandwidth is only achieved

when vector operations access consecutive memory locations. The higher bandwidth and lack of cache memory are complementary tradeoffs, in that multiple loads of the same data from main memory can be tolerated because of the high bandwidth available. Algorithms that must operate on large amounts of memory are good candidates for the GPU.

Graphics applications demand high memory bandwidth because of the numerous large images used to texture 3D objects, the large numbers of triangles used to represent the shapes of the objects, the several stages in the rendering process and the dozens of frames that must be rendered per second in games, their major market. Since high memory bandwidth is vital to graphics, future generations of GPUs are likely to retain this advantage over CPUs. However, the reliance on a programmer-controlled shared memory in favor of an automatically managed cache may not persist. The designers of Intel's Larrabee, for example, chose the latter.

Summary

We have described some of the major architectural features of GPUs that determine which algorithms they run most efficiently. These are the SIMD operations with a related threading model, the high bandwidth and high latency of memory operations, limited communications abilities between multiprocessors, and small, manually allocated cache memory. We believe that with the possible exception of the latter, these architectural features are likely to persist through future generations of GPUs and that therefore they should be considered when designing and assessing algorithms that are likely to retain their value.

Bibliography

[1] National Highway Traffic Safety Administration. 2007 traffic safety annual assessment – highlights, August 2008. DOT HS 811 017.

[2] National Highway Traffic Safety Administration. Traffic safety facts: Older population, 2008. DOT HS 811 161.

[3] James S. Albus. 4d/rcs a reference model architecture for intelligent unmanned ground vehicles. In *Proceedings of the SPIE 16th Annual International Symposium on Aerospace/Defense Sensing, Simulation and Controls*, 2002.

[4] AMD Corporation. *Technical Overview: ATI Stream Computing*, 2009.

[5] R.L. Andersson. Aggressive trajectory generator for a robot ping-pong player. *Control Systems Magazine, IEEE*, 9(2):15–21, Feb 1989.

[6] American Medical Association. Physician's guide to assessing and counseling older drivers, 2nd edition, 2010.

[7] Andrew Bacha, Cheryl Bauman, Ruel Faruque, Michael Fleming, Chris Terwelp, Charles F. Reinholtz, Dennis Hong, Al Wicks, Thomas Alberi, David Anderson, Stephen Cacciola, Patrick Currier, Aaron Dalton, Jesse Farmer, Jesse Hurdus, Shawn Kimmel, Peter King, Andrew Taylor, David Van Covern, and Mike Webster. Odin: Team victortango's entry in the darpa urban challenge. *J. Field Robotics*, 25(8):467–492, 2008.

[8] Christopher R. Baker, David Ferguson, and John Dolan. Robust mission execution for autonomous urban driving. In *10th International Conference on Intelligent Autonomous Systems (IAS 2008)*, pages 155–163, July 2008.

[9] Anthony J. Barbera, John A. Horst, Craig I. Schlenoff, and David W. Aha. Task analysis of autonomous on-road driving. In Douglas W. Gage, editor, *Proc SPIE 5609*, pages 61–72, 2004.

[10] T. Barbera, J. Albus, E. Messina, C. Schlenoff, and J. Horst. How task analysis can be used to derive and organize the knowledge for the control of autonomous vehicles. *Robotics and Autonomous Systems*, 49(1-2):67 – 78, 2004. Knowledge Engineering and Ontologies for Autonomous Systems 2004 AAAI Spring Symposium.

[11] Jérôme Barraquand and Jean Latombe. Nonholonomic multibody mobile robots: Controllability and motion planning in the presence of obstacles. *Algorithmica*, 10:121–155, 1993. 10.1007/BF01891837.

[12] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, Liewei Bao, J. Brown, M. Mattina, Chyi-Chang Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. Tile64 - processor: A 64-core soc with mesh interconnect. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 88–598, Feb. 2008.

[13] Jonathan Bohren, Tully Foote, Jim Keller, Alex Kushleyev, Daniel D. Lee, Alex Stewart, Paul Vernaza, Jason C. Derenick, John R. Spletzer, and Brian Satterfield. Little ben: The ben franklin racing team's entry in the 2007 darpa urban challenge. *J. Field Robotics*, 25(9):598–614, 2008.

[14] O. Brock and O. Khatib. Real-time re-planning in high-dimensional configuration spaces using sets of homotopic paths. In *Robotics and Automation, 2000. Proceedings. ICRA '00. IEEE International Conference on*, volume 1, pages 550–555 vol.1, 2000.

[15] R. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, Mar 1986.

[16] Thanh-Tung Cao, Ke Tang, Anis Mohamed, and Tiow-Seng Tan. Parallel banding algorithm to compute exact distance transform with the gpu. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, I3D '10, pages 83–90, New York, NY, USA, 2010. ACM.

[17] Yi-Liang Chen, Venkataraman Sundareswaran, Craig Anderson, Alberto Broggi, Paolo Grisleri, Pier Paolo Porta, Paolo Zani, and John Beck. Terramax^{tm}: Team oshkosh urban robot. *J. Field Robotics*, 25(10):841–860, 2008.

[18] Intel Corporation. Excerpts from a Conversation with Gordon Moore: Moore's Law. `ftp://download.intel.com-/museum/Moores_Law/Video-Transcripts-/Excepts_A_Conversation_with_Gordon_Moore.pdf`.

[19] Intel Corporation. Microprocessor quick reference guide. `http://www.intel.com/pressroom/kits/quickreffam.htm`

[20] Intel Corporation. Sspec/qdf reference. http://ark.intel.com/SSPECQDF.aspx.

[21] R. Craig Coulter. Implementation of the pure pursuit path tracking algorithm. Technical Report CMU-RI-TR-92-01, Robotics Institute, Pittsburgh, PA, January 1992.

[22] Cray Inc. *Cray XMT Programming Environment User's Guide S247913*, 1.3 edition. http://docs.cray.com/books/S-2479-13/S-2479-13.pdf.

[23] DARPA. Urban challenge. competition, Nov 2007. `http://www.darpa.mil/grandchallenge/`

[24] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Sixth Symposium on Operating System Design and Implementation (OSDI)*, San Francisco, CA, October 2004.

[25] Dmitri Dolgov, Sebastian Thrun, Michael Montemerlo, and James Diebel. Practical search techniques in path planning for autonomous driving. In *Proceedings of the First International Symposium on Search Techniques in Artificial Intelligence and Robotics (STAIR-08)*, Chicago, USA, June 2008. AAAI.

[26] L. E. Dubins. On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents. *American Journal of Mathematics*, 79(3):pp. 497–516, Jul 1957.

[27] Michael Erdmann and Tomas Lozano-Perez. On multiple moving objects. In *IEEE International Conference on Robotics and Automation, San Francisco, April 7-10, 1986*, pages 1419–1424, 1986.

[28] C. Urmson et al. Autonomous driving in urban environments: Boss and the urban challenge. *J. Field Robotics*, 25(8):425–466, 2008.

[29] Dave Ferguson, Thomas M. Howard, and Maxim Likhachev. Motion planning in urban environments. *Journal of Field Robotics*, 25(11-12):939–960, 2008.

[30] David Ferguson, Christopher R. Baker, Maxim Likhachev, and John Dolan. A reasoning framework for autonomous urban driving. In *Proceedings of the IEEE Intelligent Vehicles Symposium (IV 2008)*, pages 775–780, Eindhoven, Netherlands, June 2008.

[31] David Ferguson, Thomas Howard, and Maxim Likhachev. Motion planning in urban environments: Part ii. In *Proceedings of the IEEE/RSJ 2008 International Conference on Intelligent Robots and Systems*, September 2008.

[32] Paolo Fiorini and Zvi Shiller. Motion planning in dynamic environments using velocity obstacles. *The International Journal of Robotics Research*, 17(7):760–772, July 1998.

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

[34] Thierry Fraichard and Hajime Asama. Inevitable collision states: A step towards safer robots? *Advanced Robotics*, 18(10):1001–1024, 2004.

[35] Thierry Fraichard and Vivien Delsart. Navigating dynamic environments with trajectory deformation. *Journal of Computing and Information Technology*, 2009.

[36] A. Furda and L. Vlacic. Towards increased road safety: Real-time decision making for driverless city vehicles. In *Systems, Man and Cybernetics, 2009. SMC 2009. IEEE International Conference on*, pages 2421 –2426, Oct 2009.

[37] Russell Gayle, Paul Segars, Ming C. Lin, and Dinesh Manocha. Path planning for deformable robots in complex environments. In *Proceedings of Robotics: Science and Systems*, Cambridge, USA, June 2005.

[38] D.N. Godbole, V. Hagenmeyer, R. Sengupta, and D. Swaroop. Design of emergency manoeuvres for automated highway system: obstacle avoidance problem. In *Decision and Control, 1997., Proceedings of the 36th IEEE Conference on*, volume 5, pages 4774–4779 vol.5, Dec 1997.

[39] Khronos Group. OpenCL. Computing Standard. `http://www.khronos.org/opencl/`

[40] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, July 1968.

[41] Jim Held, Jerry Bautista, and Sean Koehl. From a few cores to many: A tera-scale computing research overview. Intel Corporation. White paper, 2006.

[42] Dominik Henrich. Fast motion planning by parallel processing – a review. *Journal of Intelligent and Robotic Systems*, 20(1):45–69, 1997.

[43] J. Horst and A. Barbera. Trajectory generation for an on-road autonomous vehicle. In *Proceedings of the SPIE: Unmanned Systems Technology VIII*, volume 6230, 2006.

[44] Thomas Howard. *Adaptive Model-Predictive Motion Planning for Navigation in Complex Environments*. PhD thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, August 2009.

[45] Thomas M. Howard, Colin J. Green, Alonzo Kelly, and Dave Ferguson. State space sampling of feasible motions for high-performance mobile robot navigation in complex environments. *Journal of Field Robotics*, 25(6-7):325–345, 2008.

[46] H. Jula, E.B. Kosmatopoulos, and P.A. Ioannou. Collision avoidance analysis for lane changing and merging. *Vehicular Technology, IEEE Transactions on*, 49(6):2295–2308, Nov 2000.

[47] A. Kanaris, E.B. Kosmatopoulos, and P.A. Loannou. Strategies and spacing requirements for lane changing and merging in automated highway systems. *Vehicular Technology, IEEE Transactions on*, 50(6):1568–1581, Nov 2001.

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

[49] Alonzo Kelly and Bryan Nagy. Reactive nonholonomic trajectory generation via parametric optimal control. *The International Journal of Robotics Research*, 22(1):583 – 601, July 2003.

[50] Alonzo Kelly and Anthony Stentz. Rough terrain autonomous mobility – part 2: An active vision, predictive control approach. *Autonomous Robots*, 5:163–198, 1998. 10.1023/A:1008822205706.

[51] J.T. Kider, M. Henderson, M. Likhachev, and A. Safonova. High-dimensional planning on the gpu. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 2515 –2522, May 2010.

[52] Bruce Krogh and Chuck Thorpe. Integrated path planning and dynamic steering control for autonomous vehicles. In *Proceedings of 1986 IEEE International Conference on Robotics and Automation (ICRA '86)*, pages 1664 – 1669, April 1986.

[53] Y. Kuwata, S. Karaman, J. Teo, E. Frazzoli, J.P. How, and G. Fiore. Real-time motion planning with applications to autonomous urban driving. *Control Systems Technology, IEEE Transactions on*, 17(5):1105 –1118, sept 2009.

[54] A. Lacaze, Y. Moscovitz, N. DeClaris, and K. Murphy. Path planning for autonomous vehicles driving over rough terrain. In *Intelligent Control (ISIC), 1998. Held jointly with IEEE International Symposium on Computational Intelligence in Robotics and Automation (CIRA), Intelligent Systems and Semiotics (ISAS), Proceedings*, pages 50 –55, September 1998.

[55] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast bvh construction on gpus. *Computer Graphics Forum*, 28(2):375–384, 2009.

[56] S. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.

[57] S. M. Lavalle. Rapidly-exploring random trees: A new tool for path planning. Technical Report 98-11, Computer Science Dept., Iowa State University, October 1998.

[58] J.-W. Lee and B. Litkouhi. Control and validation of automated lane centering and changing maneuver. In *ASME Dynamic Systems and Control Conference*, 2009.

[59] Brett M. Leedy, Joseph S. Putney, Cheryl Bauman, Stephen Cacciola, J. Michael Webster, and Charles F. Reinholtz. Virginia tech's twin contenders: A comparative study of reactive and deliberative navigation. *J. Field Robotics*, 23(9):709–727, 2006.

[60] John Leonard, Jonathan How, Seth Teller, Mitch Berger, Stefan Campbell, Gaston Fiore, Luke Fletcher, Emilio Frazzoli, Albert Huang, Sertac Karaman, Olivier Koch, Yoshiaki Kuwata, David Moore, Edwin Olson, Steve Peters, Justin Teo, Robert Truax, Matthew Walter, David Barrett, Alexander Epstein, Keoni Maheloni, Katy Moyer, Troy Jones, Ryan Buckley, Matthew Antone, Robert Galejs, Siddhartha Krishnamurthy, and Jonathan Williams. A perception-driven autonomous urban vehicle. *J. Field Robot.*, 25(10):727–774, 2008.

[61] John Leonard, Jonathan How, Seth Teller, Mitch Berger, Stefan Campbell, Gaston Fiore, Luke Fletcher, Emilio Frazzoli, Albert Huang, Sertac Karaman, Olivier Koch, Yoshiaki Kuwata, David Moore, Edwin Olson, Steve Peters, Justin Teo, Robert Truax, Matthew Walter, David Barrett, Alexander Epstein, Keoni Maheloni, Katy Moyer, Troy Jones, Ryan Buckley, Matthew Antone, Robert Galejs, Siddhartha Krishnamurthy, and Jonathan Williams. A perception-driven autonomous urban vehicle. *J. Field Robotics*, 25(10):727–774, 2008.

[62] M. Likhachev and D. Ferguson. Planning long dynamically-feasible maneuvers for autonomous vehicles. In *Proceedings of Robotics: Science and Systems IV*, Zurich, Switzerland, June 2008.

[63] Maxim Likhachev and Anthony Stentz. R* search. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 2008.

[64] Tomás Lozano-Pérez and Michael A. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Commun. ACM*, 22:560–570, October 1979.

[65] L. Martinez-Gomez and T. Fraichard. An efficient and generic 2d inevitable collision state-checker. In *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, pages 234–241, Sept. 2008.

[66] Isaac Miller, Mark Campbell, Dan Huttenlocher, Frank-Robert Kline, Aaron Nathan, Sergei Lupashin, Jason Catlin, Brian Schimpf, Pete Moran, Noah Zych, Ephrahim Garcia, Mike Kurdziel, and Hikaru Fujishima. Team cornell's skynet: Robust perception and planning in an urban environment. *J. Field Robot.*, 25(8):493–527, 2008.

[67] Javier Minguez and L. Montano. Nearness diagram (nd) navigation: collision avoidance in troublesome scenarios. *Robotics and Automation, IEEE Transactions on*, 20(1):45–59, Feb. 2004.

[68] Michael Montemerlo, Jan Becker, Suhrid Bhat, Hendrik Dahlkamp, Dmitri Dolgov, Scott Ettinger, Dirk Haehnel, Tim Hilden, Gabe Hoffmann, Burkhard Huhnke, Doug Johnston, Stefan Klumpp, Dirk Langer, Anthony Levandowski, Jesse Levinson, Julien Marcil, David Orenstein, Johannes Paefgen, Isaac Penny, Anna Petrovskaya, Mike Pueger, Ganymed Stanek, David Stavens, Antone Vogt, and Sebastian Thrun. The stanford entry in the urban challenge. *Journal of Field Robotics*, 2008.

[69] Michael Montemerlo, Jan Becker, Suhrid Bhat, Hendrik Dahlkamp, Dmitri Dolgov, Scott Ettinger, Dirk Haehnel, Tim Hilden, Gabe Hoffmann, Burkhard Huhnke, Doug Johnston, Stefan Klumpp, Dirk Langer, Anthony Levandowski, Jesse Levinson, Julien Marcil, David Orenstein, Johannes Paefgen, Isaac Penny, Anna Petrovskaya, Mike Pflueger, Ganymed Stanek, David Stavens, Antone Vogt, and Sebastian Thrun. Junior: The stanford entry in the urban challenge. *Journal of Field Robotics*, 25(9):569–597, 2008.

[70] Michael Montemerlo, Jan Becker, Suhrid Bhat, Hendrik Dahlkamp, Dmitri Dolgov, Scott Ettinger, Dirk Hähnel, Tim Hilden, Gabe Hoffmann, Burkhard Huhnke, Doug Johnston, Stefan Klumpp, Dirk Langer, Anthony Levandowski, Jesse Levinson, Julien Marcil, David Orenstein, Johannes Paefgen, Isaac Penny, Anna Petrovskaya, Mike Pflueger, Ganymed Stanek, David Stavens, Antone Vogt, and Sebastian Thrun. Junior: The stanford entry in the urban challenge. *J. Field Robotics*, 25(9):569–597, 2008.

[71] Michael Montemerlo, Jan Becker, Suhrid Bhat, Henrik Dahlkamp, Dmitri Dolgov, Scott Ettinger, and Dirk Haehnel. Junior: The stanford entry in the urban challenge. *Journal of Field Robotics*, 25(9):569–597, 2008.

[72] Intel Corporation Museum. Product category list. `http://download.intel.com-/museum/research/arc_collect/timeline-/TimelineProductTypeSort7_05.pdf`.

[73] Wassim G. Najm, John D. Smith, and Mikio Yanagisawa. Pre-crash scenario typology for crash avoidance research. Technical Report DOT HS 810 767, National Highway Traffic Safety Administration, April 2007.

[74] N. J. Nilsson. *Principles of Artificial Intelligence*. Morgan Kaufmann Publishers, Inc., 1980.

[75] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*, version 2.0 edition, June 2008.

[76] Office of Highway Policy Information. Traffic volume trends: December 2010. Technical report, Federal Highway Administration, US DOT, Feb 2011.

[77] Jia Pan, C. Lauterbach, and D. Manocha. Efficient nearest-neighbor computation for gpu-based motion planning. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 2243 –2248, Oct 2010.

[78] Jia Pan, Christian Lauterbach, and Dinesh Manocha. g-Planner: Real-time motion planning and global navigation using GPUs. In *AAAI Conference on Artificial Intelligence*, 2010.

[79] I. Papadimitriou and M. Tomizuka. Fast lane changing computations using polynomials. *American Control Conference, 2003. Proceedings of the 2003*, 1:48–53 vol.1, 4-6 June 2003.

[80] Benjamin J. Patz, Yiannis Papelis, Remo Pillat, Gary Stein, and Don Harper. A practical approach to robotic design for the darpa urban challenge. *J. Field Robotics*, 25(8):528–566, 2008.

[81] D. Pham, S. Asano, M. Bolliger, M.N. Day, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation cell processor. In *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, pages 184–592 Vol. 1, Feb. 2005.

[82] A. Piazzi, C.G. Lo Bianco, M. Bertozzi, A. Fascioli, and A. Broggi. Quintic g2-splines for the iterative steering of vision-based autonomous vehi-

cles. *Intelligent Transportation Systems, IEEE Transactions on*, 3(1):27 –36, March 2002.

[83] M. Pivtoraiko and A. Kelly. Efficient Constrained Path Planning via Search in State Lattices. In *'i-SAIRAS 2005' - The 8th International Symposium on Artificial Intelligence, Robotics and Automation in Space*, volume 603 of *ESA Special Publication*, August 2005.

[84] Mikhail Pivtoraiko, Ross Alan Knepper, and Alonzo Kelly. Differentially constrained mobile robot motion planning in state lattices. *Journal of Field Robotics*, 26(1):308–333, March 2009.

[85] Nathan Ratliff, Matt Zucker, J. Andrew Bagnell, and Siddhartha Srinivasa. CHOMP: Gradient optimization techniques for efficient motion planning. In *Proceedings of the International Conference on Robotics and Automation(ICRA)*, 2009.

[86] Fred W. Rauskolb, Kai Berger, Christian Lipski, Marcus A. Magnor, Karsten Cornelsen, Jan Effertz, Thomas Form, Fabian Graefe, Sebastian Ohl, Walter Schumacher, Jörn-Marten Wille, Peter Hecker, Tobias Nothdurft, Michael Doering, Kai Homeier, Johannes Morgenroth, Lars C. Wolf, Christian Basarke, Christian Berger, Tim Gülke, Felix Klose, and Bernhard Rumpe. Caroline: An autonomously driving vehicle for urban environments. *J. Field Robotics*, 25(9):674–724, 2008.

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

[88] Guodong Rong and Tiow-Seng Tan. Jump flooding in gpu with applications to voronoi diagram and distance transform. In *I3D '06: Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, pages 109–116, New York, NY, USA, 2006. ACM.

[89] Julio Rosenblatt. Damn: A distributed architecture for mobile navigation. *Journal of Experimental and Theoretical Artificial Intelligence*, 9(1):339 – 360, 1997.

[90] M. Rufli and R. Siegwart. On the design of deformable input-/state-lattice graphs. In *Proceedings of ICRA*, Anchorage, Alaska, 2010.

[91] Martin Rufli, Dave Ferguson, and Roland Siegwart. Smooth path planning in constrained environments. In *ICRA 2009*, 2009.

[92] A. Scheuer and T. Fraichard. Collision-free and continuous-curvature path planning for car-like robots. In *Robotics and Automation, 1997. Proceedings., 1997 IEEE International Conference on*, volume 1, pages 867–873 vol.1, Apr 1997.

[93] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[94] Dong Hun Shin and Sanjiv Singh. Path generation for robot vehicles using composite clothoid segments. Technical Report CMU-RI-TR-90-31, Robotics Institute, Pittsburgh, PA, December 1990.

[95] David Silver. *Learning Preference Models for Autonomous Mobile Robots in Complex Domains*. PhD thesis, Carnegie Mellon University, December 2010.

[96] Jarrod M. Snider. Automatic steering methods for autonomous automobile path tracking. Technical Report CMU-RI-TR-09-08, Robotics Institute, Pittsburgh, PA, February 2009.

[97] Rahul Sukthankar. *Situation Awareness for Tactical Driving*. PhD thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, January 1997.

[98] P. Tompkins, A. Stentz, and D. Wettergreen. Global path planning for mars rover exploration. In *Aerospace Conference, 2004. Proceedings. 2004 IEEE*, volume 2, pages 801 – 815 Vol.2, Mar 2004.

[99] Paul Tompkins. *Mission-Directed Path Planning for Planetary Rover Exploration*. PhD thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, May 2005.

[100] Allen B. Tucker, editor. *Computer Science Handbook*, chapter Parallel Algorithms (10). CRC Press, second edition, 2004.

[101] Chris Urmson, Joshua Anhalt, Drew Bagnell, Christopher R. Baker, Robert Bittner, M. N. Clark, John M. Dolan, Dave Duggins, Tugrul Galatali, Christopher Geyer, Michele Gittleman, Sam Harbaugh, Martial Hebert, Thomas M. Howard, Sascha Kolski, Alonzo Kelly, Maxim Likhachev, Matthew McNaughton, Nick Miller, Kevin Peterson, Brian Pilnick, Raj Rajkumar, Paul E. Rybski, Bryan Salesky, Young-Woo Seo, Sanjiv Singh, Jarrod Snider, Anthony Stentz, William Whittaker, Ziv Wolkowicki, Jason Ziglar, Hong Bae, Thomas Brown, Daniel Demitrish, Bakhtiar Litkouhi, Jim Nickolaou, Varsha Sadekar, Wende Zhang, Joshua Struble, Michael Taylor, Michael Darms, and Dave Ferguson. Autonomous driving in urban environments: Boss and the urban challenge. *J. Field Robotics*, 25(8):425–466, 2008.

[102] Christopher Urmson. planning from the plan. personal communication.

[103] Christopher Urmson. *Navigation Regimes for Off-Road Autonomy*. PhD thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, May 2005.

[104] Felix von Hundelshausen, Michael Himmelsbach, Falk Hecker, Andre Mueller, and Hans-Joachim Wuensche. Driving with tentacles: Integral structures for sensing and motion. *J. Field Robotics*, 25(9):640–673, 2008.

[105] Moritz Werling, Julius Ziegler, Sören Kammel, and Sebastian Thrun. Optimal trajectories for dynamic street scenarios in a frenét frame. In *Proceedings of ICRA*, pages 987–993, 2010.

[106] J. Ziegler and C. Stiller. Spatiotemporal state lattices for fast trajectory planning in dynamic on-road driving scenarios. In *Proceedings of IROS*, 2009.