PATH PLANNING AND PATH FOLLOWING FOR AN AUTONOMOUS CAR

By

Matthew Bradley

APPROVED:

Graduate Committee:

_____

Supervisor: Dr. Cen Li (Computer Science)

_____

Dr. Chrisila Pettey (Computer Science)

_____

Dr. Jungsoon Yoo (Computer Science)

_____

Dr. Chrisila Pettey, Chairperson Computer Science Department

_____

Dr. Michael Allen, Dean of the College of Graduate Studies

PATH PLANNING AND PATH FOLLOWING FOR AN AUTONOMOUS CAR

By

Matthew Bradley

A thesis submitted in partial fulfillment

of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

Middle Tennessee State University

December 2012

ACKNOWLEDGEMENTS

I would like to thank Dr. Cen Li for her advice, guidance, and encouragement during the entire writing process.

I would also like to thank the creators of the open-source software used in my simulation: XNA, Farseer Physics, C5 Collections, and XNA Game Console.

Special thanks go to Sebastian Thrun for sparking my interest in this topic, whose online education ventures gave me a head-start into the current research of autonomous vehicles.

ABSTRACT

Modern technology can be harnessed to control a passenger vehicle and have it drive more safely and more accurately than a human driver. This thesis presents modern algorithms for an autonomous car that can plan and follow safe, drivable paths. The Hybrid A* algorithm is used to find continuous paths through an obstacle-laden environment. The Stanley method is used to control a vehicle's steering to track these paths with low error. A simulation was developed that implements the path planning and path following techniques discussed in this thesis to drive a car autonomously through virtual environments mimicking real-world scenarios. Experiments were conducted to test the performance of the algorithms in specific types of environments. The experimental results demonstrate the credibility of the presented algorithms as appropriate methods for planning paths efficiently and executing them accurately in obstacle-dense environments.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

INTRODUCTION

An increasing interest in research and experimentation on autonomous vehicles, especially self-driving automobiles, has been seen in the past several years. One key component of autonomous vehicle development is the planning and execution of a navigable path to some goal destination. However, the dynamics governing the motion of self-driving cars make them quite different from previously researched autonomous vehicles. The motion of passenger vehicles is characterized as non-holonomic. A non-holonomic agent is a system that has more degrees of freedom than can be controlled [3]. The continuous state of an automobile can be described with three dimensions: a two-dimensional position and a heading [17]. The two-dimensional position[1] defines where the vehicle is in the environment, and the heading determines which direction the vehicle is pointing. However, vehicles have only two controllable degrees of freedom: the vehicle's steering and the vehicle's forward or backward movement. A simple way of describing this non-holonomic nature is to say that an automobile cannot rotate in place: it cannot change its heading without changing its position. Classical grid-based methods such as Djikstra's algorithm [5] and A* [10] insufficiently model the turning constraints of passenger vehicles. In order to properly function on today's cars, modern path planning techniques must be able to discover safe, collision-free paths that correctly account for the restrictions inherent to the vehicle's dynamics.

[1] Even though world space is three-dimensional, two dimensions are sufficient to describe a vehicle's position because it only travels in the ground plane.

Another challenge regarding path planning for passenger vehicles is the fact that these vehicles, and the obstacles they attempt to avoid, exist in a continuous environment. Standard grid-based path planning techniques cannot provide adequate resolution for planning paths in a non-discrete world without greatly inflating computation time. Neither can a vehicle navigate the discrete path that a grid-based path planner would discover [15]. Any path planning algorithm employed to find drivable paths for autonomous cars must provide a continuous path that can be computed in a reasonable amount of time.

Once a path has been planned, path following techniques are used to successfully drive the vehicle along this path from the start to the goal. These techniques are evaluated using several criteria: how accurately the vehicle follows the path, how safely the path is followed, and how quickly the goal is achieved. A path follower must perform well in these areas in real time as the vehicle travels toward the goal.

The goal of this thesis is to demonstrate current techniques used to plan paths and follow trajectories for autonomous cars. An efficient path planner is introduced that discovers safe, smooth, drivable paths in a continuous environment. The path follower presented is robust enough to follow this path with minimal error. A simulation, written in C# using the XNA game engine [22], was created to test these methodologies in virtual environments mimicking real-world scenarios.

Previous Work

One of the first contemporary forays into driverless car experimentation was the 2004 DARPA Grand Challenge [1]. This 150 mile long off-road autonomous car race through the Mojave Desert ended in complete failure: none of the teams made it farther

than 5% of the full length of the course, leaving the $1 million prize unclaimed. This fiasco jump-started research into off-road autonomous driving which led to five vehicles finishing the 132 mile long course of the 2005 DARPA Grand Challenge [2]. Stanford University's entry, Stanley [18], finished first and claimed the $2 million prize. Stanley's method of path following, introduced in the next chapter, is used as a basis for this simulation's path follower.

The success of the 2005 DARPA Grand Challenge led to the 2007 DARPA Urban Challenge [19], a road-based "obstacle course" of real-world traffic scenarios. This race introduced the academic world to autonomous driving challenges such as merging, 4-way stops, sharing the road, and parking. The winner of the $2 million prize was Boss [20], a collaboration between Carnegie Mellon University and General Motors. The second place finisher was Junior [15], the entry from Stanford's racing team. Junior's free-space urban path planner was used as a foundation for the simulation's path planning component.

After the accomplishments of the DARPA Challenges, the lead researcher of the Stanford team, Sebastian Thrun, joined Google in its driverless car project [14]. Google has used data from Google Maps and Google Street View to map roads drivable by autonomous vehicles in Nevada and California. Google employs a fleet of test vehicles outfitted with GPS, inertial measurement units, laser rangefinders, cameras, and other sensing equipment, and its cars have already driven hundreds of thousands of miles along roads and in traffic, all autonomously.

This thesis is organized as the following. Chapter 2 presents background information on the current research in path planning and path following including the

theory and motivation for the algorithms utilized in the simulation. Chapter 3 explains the implementation of these algorithms into each component of the simulation. Chapter 4 introduces the design of several experiments used to test the different elements of the system. Chapter 5 enumerates the results and discusses the consequences of these experiments. Chapter 6 discusses the conclusions of this thesis and suggests future work.

CHAPTER II

BACKGROUND

This chapter presents a high-level overview of the theory and methods developed to solve the problems of path planning and path following for a vehicle. This work forms the foundation for the implementation of the simulation described in the next chapter.

Path Planning

The path planning methods explored in this paper are based on the methods used by Stanford's Junior [15] during the 2007 DARPA Urban Challenge [19].

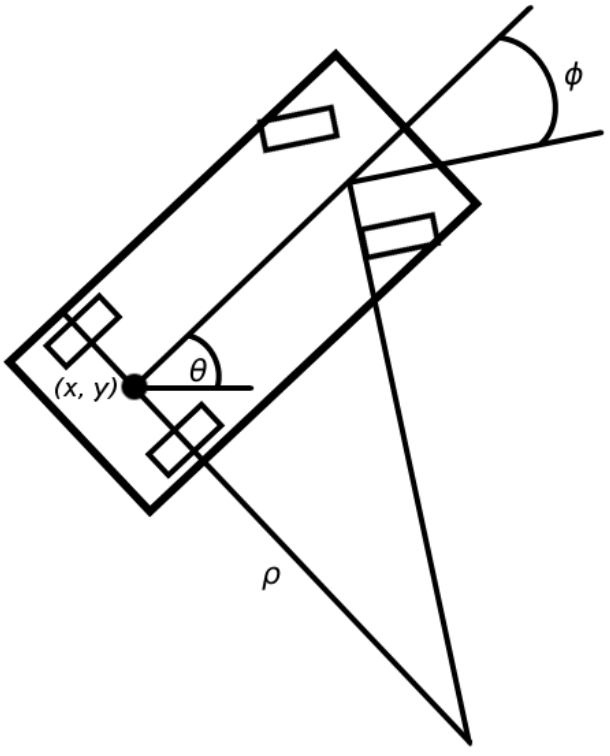


Figure 1. Three-dimensional state of a vehicle: (x, y) is the world coordinates of the center of the rear axle, and θ is the heading relative to the x-axis. The wheel deflection (ϕ) and minimum turning radius (ρ) are important in the path following routine.

Path planning is used to discover a feasible path from a starting pose to some goal pose. In autonomous vehicle path planning, a pose is commonly represented as the three-dimensional state vector (x, y, θ), where x and y describe the world location of the center of the rear axle of the vehicle and θ is the vehicle's heading relative to the x-axis [3]. This 3-D state is shown in Figure 1. The path planner will accept as input a start pose $\boldsymbol{s_0} = (x, y, \theta)_0$ and a goal pose $\boldsymbol{s_g} = (x, y, \theta)_g$ and return as output a sequence of vehicle poses $\boldsymbol{s_0}, \boldsymbol{s_1}, \ldots, \boldsymbol{s_g}$ that represents a safe, near-optimal, drivable path from $\boldsymbol{s_0}$ to $\boldsymbol{s_g}$ [6]. This trajectory is then smoothed using gradient descent, a numerical optimization technique.

The path-planner described in this thesis uses an extension of A* search [10], an algorithm commonly utilized in robotic path-finding. The search algorithm imposes a grid onto the vehicle's local environment and associates a continuous vehicle pose with each grid cell as it searches. This Hybrid A* search [15] allows a discrete environment (the grid) to be searched using continuous nodes (the vehicle poses).

Occupancy Grid

At the start of the path-planning procedure, the environment is indexed into an occupancy grid. This occupancy grid determines which cells in the environment are occupied by obstacles and which belong to the free space. There are several ways to generate an occupancy grid. It could be populated using sensors installed on a real-world vehicle, such as radar or laser rangefinders. These sensors would perceive obstacles in the environment and add them to the occupancy grid based on the current location of the vehicle and the relative location of the perceived obstacle. As the vehicle moves through the environment, the occupancy grid will be updated as new obstacles are detected.

For the simulation developed for this thesis, the occupancy grid is generated by rasterizing polygonal obstacles into a grid. This process is explained in more detail in the next chapter.

Distance Map and Voronoi Diagram

The simple occupancy grid is insufficient to find a path that safely maneuvers around obstacles. A weakness of the grid is that the path planner will prefer paths that hug obstacles when turning to minimize the path length. One way to define a tradeoff between path length and proximity to obstacles is to use a potential field [6]. A potential field may be used to augment the occupancy grid such that each cell has an associated traversal cost. Cells that are closer to obstacles have higher traversal costs; cells that are occupied by obstacles are impassable (infinite cost). The traversal costs of the potential field are used to repel the path away from obstacles while searching.

While standard potential fields define the traversal cost of a cell as a function of the distance to the nearest obstacle, the path-planner described in this paper uses a variant called a Voronoi field that defines the traversal cost as a function of the distance to the nearest obstacle and the distance to the nearest Voronoi edge [7].

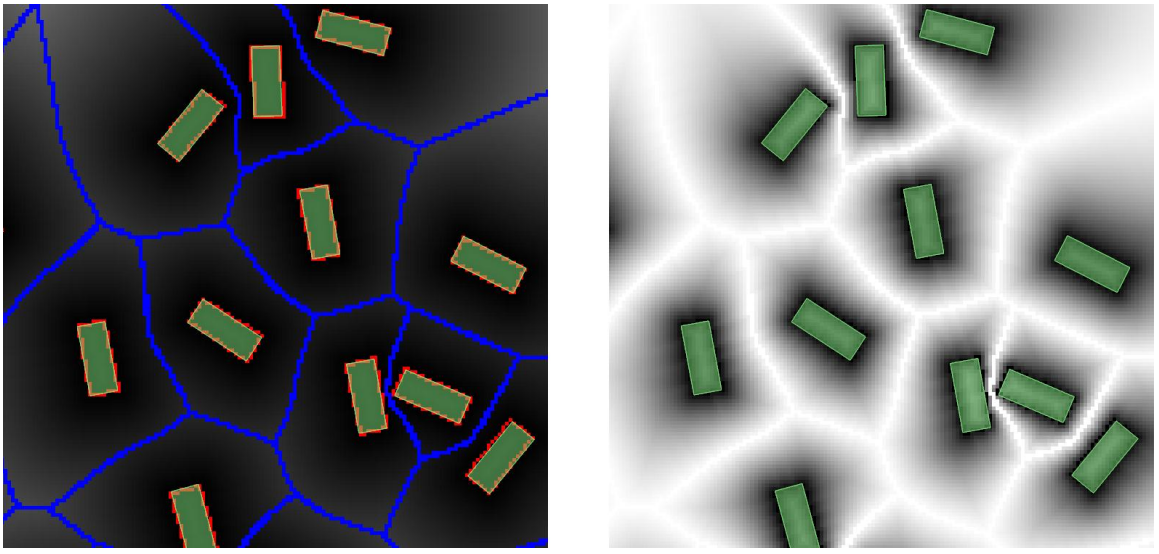
A Voronoi Diagram is a decomposition of a space based on the distance to objects in the space. For a given object *o*, the subset of the space that is closer to *o* than any other object forms the Voronoi cell of *o*. The subset of the space that is equidistant to more than one object makes up the Voronoi edges. The Voronoi Diagram concept is adapted to grids using a generalized Voronoi diagram (GVD). The GVD of the environment is the set of all cells in the grid that contain a point that is equidistant from more than one obstacle cell [13]. This is equivalent to the set of Voronoi edges.

In order to create the Voronoi field, a Euclidean distance map must first be computed on the environment grid. This distance map stores, for each grid cell, the Euclidean distance to the nearest obstacle cell. When a cell is equidistant from one or more obstacle cells, it is added to the GVD [13]. The GVD algorithm stores for each grid cell: the location of the nearest obstacle cell, the distance to the nearest obstacle, and the distance to the nearest Voronoi edge. Figure 2(a) shows an occupancy grid with its associated distance field and generalized Voronoi diagram. The next chapter explains in more detail the algorithms used to generate the distance map and GVD.

The distance map and GVD are used to define the Voronoi field as follows [7]:

$$\rho_V(x,y) = \left(\frac{\alpha}{\alpha + d_O}\right)\left(\frac{d_V}{d_O + d_V}\right)\left(\frac{(d_O - d_O^{max})^2}{(d_O^{max})^2}\right),$$

where $\rho_V(x,y)$ is the traversal cost at grid cell (x,y), d_O is the distance from the cell to the nearest obstacle, and d_V is the distance from the cell to the nearest Voronoi edge. The constants α and d_O^{max} define the falloff rate and maximum effective range of the field, respectively. If $d_O \geq d_O^{max}$, then $\rho_V(x,y) = 0$. This potential field, $\rho_V(x,y) \in [0,1]$, reaches its highest within obstacles and its lowest on the edges of the GVD. Since the Voronoi field is a function of how close a cell is to an edge of the GVD, narrow (but still safe) openings are still navigable: the field value of a cell is scaled in proportion to the total available clearance between the two closest obstacles [6]. This advantage allows the path-planner to repel the search away from obstacles in both narrow and wide openings. Figure 2(b) shows the Voronoi field computed on a sample environment. Notice how the value of a cell is scaled in proportion to the clearance between its two closest obstacles.



(a) (b)

Figure 2. In the distance map (a), the pixel brightness shows the distance to the nearest obstacle cell (shown in red), where darker pixels are closer to obstacles. The blue cells are equidistant to more than one obstacle and make up the GVD. The potential field (b) shows the traversal cost of each cell, where lighter cells have a lower cost. Notice how the slope of the field is determined by the clearance between the two closest obstacles.

Hybrid A* Search

Because autonomous vehicles exist in a continuous state space, standard algorithms used in grid-based path-planning, like Djikstra's algorithm [5] or A* [10], must be modified to capture continuous state data in discrete search nodes [7]. Hybrid A* [6] modifies the node-expansion operation of A* to store continuous vehicle poses in discrete grid cells. The path-planner performs a continuous A* search in the four-dimensional search space (x, y, θ, r), where the fourth dimension ($r \in \{0, 1\}$) represents the current direction of motion (forward or reverse).

The A* search is performed on a discretized four-dimensional grid, where each grid cell is associated with a continuous 4D vehicle pose. The occupancy grid that

spatially indexes the obstacles is used to prune unsafe children. The usual A* evaluation function f is used to determine the order of the nodes visited in the search:

$$f(x) = g(x) + h(x)$$

where $g(x)$ is the path cost from the starting node to node x and $h(x)$ is an admissible heuristic estimate of the path cost from node x to the goal node.

The starting vehicle pose is added to the A* open list. Like in the traditional A* approach, the node with the lowest f value is popped from the open list and expanded. A node is expanded by applying three steering controls (max left turn, max right turn, and no turn) with two motion controls (forward and reverse) for some fixed distance based on the grid resolution to create six continuous child nodes [6]. Each child node is checked for collisions, and unsafe children are pruned. For the remaining continuous child nodes, the grid cell that it falls into is found. The child node is pruned if the node already associated with that grid cell has a lower cost. Otherwise, the child node is associated with that grid cell and added to the open list using the f value as the node's priority. The search continues until the goal pose is reached – which returns the discovered path – or until the open list is empty, signifying that no path could be found.

Unlike traditional A*, which is guaranteed to generate an optimal path, Hybrid A* does not generate optimal paths because of the discretization of the environment and controls [6]. Additionally, Hybrid A* is not complete: it may fail to find a solution even if one exists [6]. However, Hybrid A* always finds a feasible, near-optimal solution (where one exists) in realistic environments [15]. Figure 3 shows a Hybrid A* search through a sample environment.

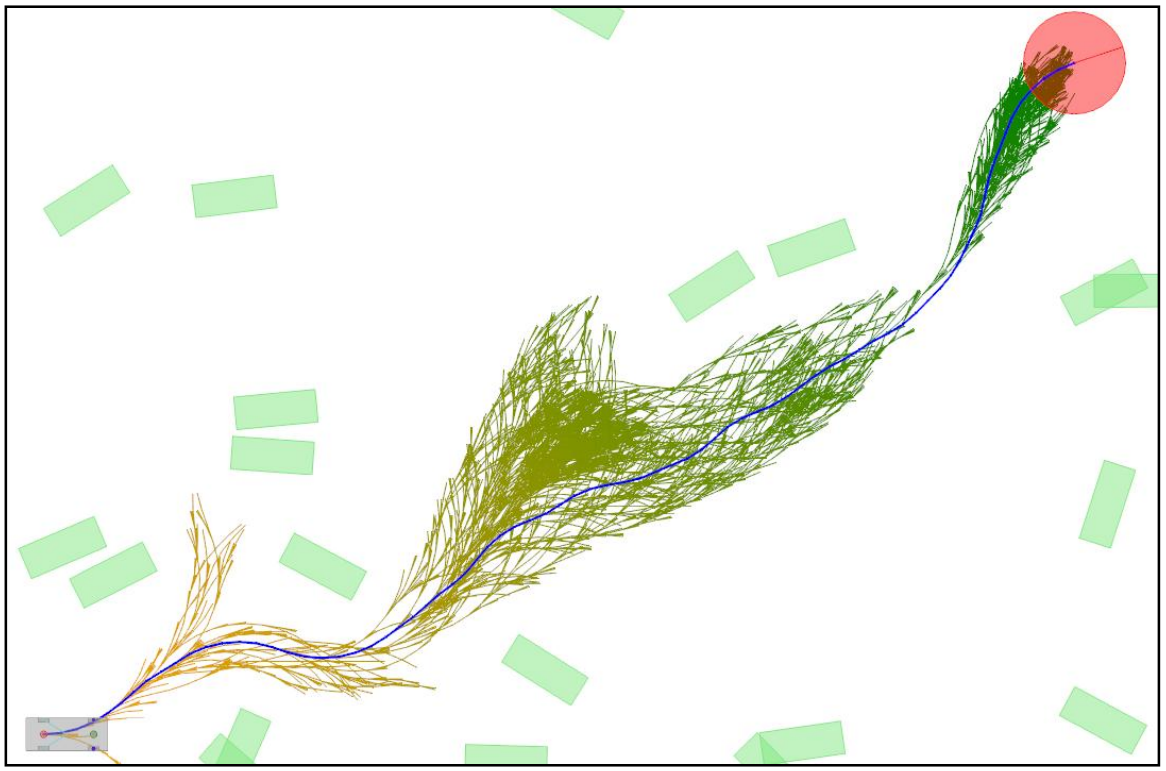


Figure 3. Hybrid A* search tree starting at the car in the bottom left and ending at the goal pose in the top right. The blue line is the lowest-cost path discovered by the search.

Heuristics

In most simple robotic path-planning applications, a Euclidean distance heuristic is sufficient. However, this heuristic fails for Hybrid A* because the path planner searches in more than two dimensions [6]. If the search approaches the goal from the wrong direction, it quickly degrades to breadth-first search. This is because the Euclidean distance from a pose to the goal is the same for all orientations. If the search is close to the destination but has not yet acquired the goal pose because it is facing the wrong direction, the search has no way to prioritize child nodes. The search degrades to breadth-first search, expanding nodes surrounding the goal pose as it hunts for the correct

orientation. The path-planner relaxes two constraints of the problem to produce two heuristics used to focus the Hybrid A* search and avoid this search degradation problem.

Relaxing the Obstacle Constraint

The first heuristic is called the *non-holonomic-without-obstacles* heuristic [7]. This heuristic relaxes the path-planning problem by ignoring all obstacles in the environment. The non-holonomic constraint is still considered: the heuristic must obey the minimum turning radius of the vehicle. This heuristic is computed by finding the optimal path from the origin to every cell in some discretized neighborhood. Because this heuristic ignores obstacles in the environment, it can be completely precomputed offline. This heuristic is obviously admissible because any obstacles in the actual environment would only make the precomputed optimal path longer. The effect of this heuristic is to prune branches of the search that approach the goal from the wrong direction [7].

Relaxing the Turn Radius Constraint

The second heuristic is a counterpart to the first and is called the *holonomic-with-obstacles* heuristic [7]. This heuristic considers obstacles in the environment but ignores the kinematic constraints of the vehicle: it does not have to obey the minimum turning radius. The heuristic is computed at the beginning of every path-planning routine by performing Djikstra's algorithm on the occupancy grid to find the length of the shortest 2D path from every cell to the goal using 8-neighbor expansions.

This heuristic is very useful in maze-like environments: since the shortest path is found from every cell in the environment to the goal, it has the property of pushing the search down open corridors and away from dead-ends. Open routes will offer a lower

heuristic value than routes leading to dead-ends, so the A* search does not waste time travelling down paths that do not lead to the goal.

Because this heuristic uses 8-neighbor grid expansions – and because an actual vehicle is not limited to only straight and diagonal paths between cells – the path lengths may overestimate the shortest path length to the goal. To account for this, every value of the heuristic is discounted by a constant factor which guarantees that the heuristic never overestimates the optimal path length to the goal, thus making it admissible. The determination of this constant factor is discussed in the next chapter.

Since both heuristics are admissible, the maximum of the two is used when determining the path-cost of a node.

Path Following

Path following involves taking a sequence of vehicle states discovered by a planner and generating a set of control actions that can be used by the vehicle to follow that path. A path tracker must provide both steering controls and velocity controls.

Steering Control

There are several simple but robust methods that can be used to track a path. Two of those – pure pursuit and the Stanley method are discussed in this section.

Pure Pursuit

The pure pursuit method [17] is one of the simplest path following methods. It involves finding a point on the path at some look-ahead distance from the current path vertex and turning the front wheel so that a circular arc connects the position of the center of the rear axle with the goal point.

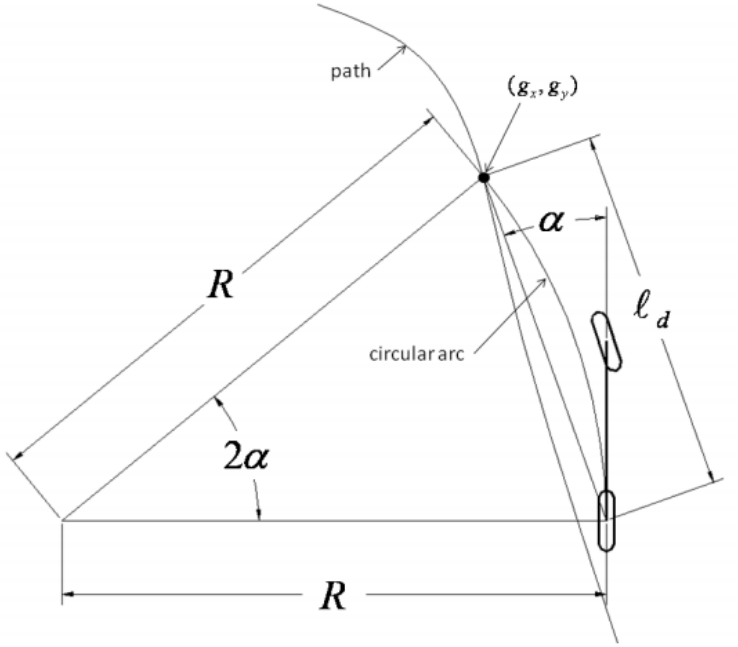


Figure 4. The pure pursuit method [17]

Figure 4 shows a diagram of the pure pursuit method: ℓ_d is the look-ahead distance, (g_x, g_y) is the goal point, R is the radius of curvature of the vehicle, and α is the angle between the vehicle's heading vector and goal vector. The desired front wheel deflect δ is found using the following equation from [17]:

$$\delta = \tan^{-1}\left(\frac{2L\sin\alpha}{\ell_d}\right),$$

where L is the distance between the vehicle's front and rear axle.

The main concern when using the pure pursuit method is choosing the look-ahead distance. If the look-ahead is too low, then the steering will be too reactionary, and the vehicle will turn too late around corners. If the look-ahead is too high, then the vehicle will cut corners and not be able to accurately track the desired path. Because of this problem, the pure pursuit method is not robust enough for precise path following [17].

Stanley Method

Another path following method was employed by Stanford's Stanley [18] during the 2005 DARPA Grand Challenge [2]. This method is an extension of pure pursuit and uses a combination of the cross track error and the heading error to steer the vehicle along the path.

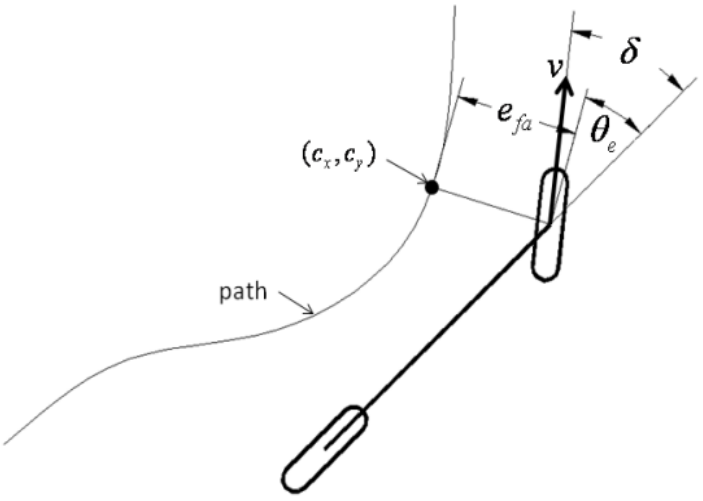


Figure 5. The Stanley method [17]

In Figure 5, the cross track error e_{fa} is the distance from the front axle to the closest point on the desired path. The heading error θ_e is the difference between the vehicle's current heading and the heading of the path at point $(c_x c_y)$. The desired front wheel deflection δ can be found using the following equation [17]:

$$\delta = \theta_e + \tan^{-1}\left(\frac{ke_{fa}}{v}\right),$$

where k is a gain parameter and v is the current velocity. When the second term is non-zero (the vehicle is off the path), the car steers towards some look-ahead point defined by

the current velocity and the gain parameter. When the second term is zero (the vehicle's front axle is on the path), the first term steers the car towards the instantaneous path heading. This reduces the overshoot as the vehicle approaches the path.

Velocity Control

There are several reasons why a controller would impose a maximum velocity on specific parts of a tracked path [18]. For instance, the goal pose at the end of the path should have a maximum velocity of zero, indicating that the vehicle should stop once it reaches its destination. As shown in Figure 6, other factors influence the maximum allowed velocity at different parts of the path, such as a global speed limit, proximity to obstacles, and path curvature. The constraints used by the simulation's velocity recommender are described in the next chapter.

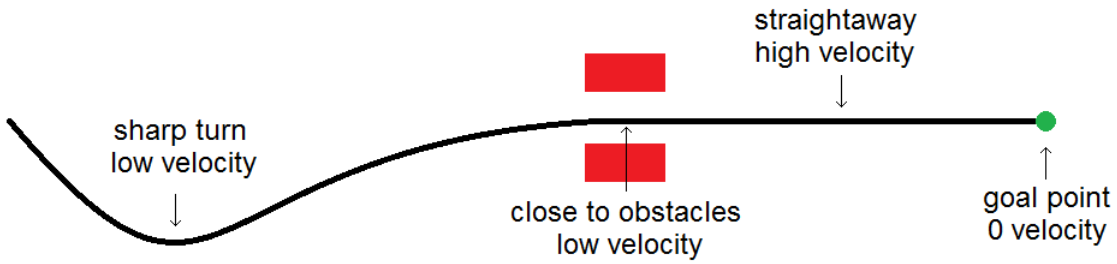


Figure 6. Different sections of a tracked path can have different recommended velocities.

Conclusion

The algorithms and theory presented in this chapter form important foundations for the path planner and path follower utilized in the simulation. The next chapter explains how each component of the simulation was implemented using the methods described here.

CHAPTER III

IMPLEMENTATION

A simulation using the algorithms and theory described in the previous chapter was developed to experiment with and test drive a holonomically-constrained vehicle from one pose to another. Each scenario given to the simulator is loaded from a mission file which defines various properties of the scenario, such as start and destination poses, path planning configuration, and the environment data. The system then progresses through all the steps necessary to plan and execute a path through the environment: the environment obstacles are decomposed into a grid, Hybrid A* is used to plan a path on that grid, and a controller generates a series of gas, brake, and steering controls in response to real-time errors to move the vehicle along that path to the goal pose.

The entire simulation was written in C#, and the project is managed using Visual Studio. C# has the disadvantage of being slower than C++; however it was chosen over a faster language because of this author's familiarity with XNA [22], Microsoft's game engine, on which the simulation is built. XNA includes high-level game loop and drawing libraries and provides the system with a good starting point for a complex, visual simulation of vehicle movement. Along with XNA, the following third-party libraries are included in the dependencies:

- *Farseer Physics* [8]: a physics engine which drives the collision detection between the vehicle and the environment. It is also used to accurately simulate the basic dynamics of a vehicle, such as acceleration, braking, and steering.

- *C5 Collections* [12]: a C# generic collections library that is included mainly for its implementation of a priority queue, which is needed by the path planner.
- *XNAGameConsole* [21]: a library that provides a command line console inside the simulation which is used to display system information and change simulation parameters.

Figure 7 provides a high-level overview of the process the simulation takes from loading a mission to planning a drivable path to successfully reaching the goal pose. The implementation of each of these components is described in detail in the following sections.

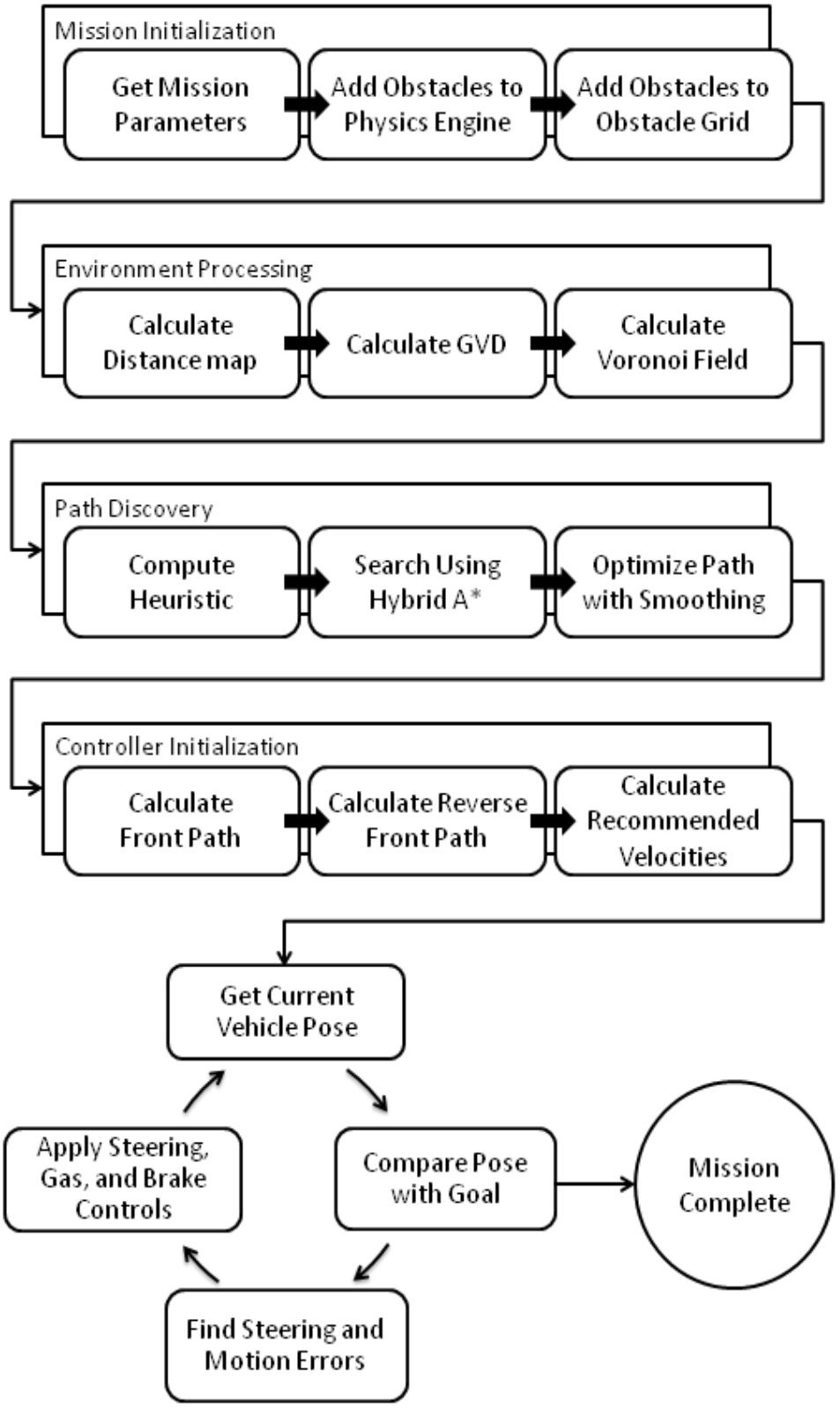


Figure 7. High level flow chart of the simulation

Mission Loader

The information used to configure a trip to be taken by the vehicle is contained inside a mission file. Each mission is defined by a starting pose, a destination pose, and an environment. The environment contains information for all obstacles in the scenario, as well as obstacle grid parameters such as size and resolution. The mission file is written in JavaScript Object Notation, or JSON, a human-readable, associative-array document standard. This file is parsed by the mission loader, and all parameters are loaded into the simulation. The format of the mission file is described in Appendix A.

Path Planner

Once a mission is loaded, the path planner can begin finding a path from the start pose to the destination pose in the environment defined by the mission. The goal of the path planner is to generate a safe, feasible path from the start to the destination. A stated in the previous chapter, a *pose* is a three dimensional state vector (x, y, θ) that describes a position and orientation. Before a path can be found, the environment must be decomposed into an obstacle grid suitable for planning.

Occupancy Grid

Each obstacle added to the simulation by the mission's environment information is a static, polygonal shape with a continuous position. Polygons are appropriate for estimating the shape of real-world objects; however, they are not well-suited for the planning algorithm used to discover a collision-free path. Hybrid A* operates on an occupancy grid, where each cell has some degree of occupancy or path cost used to find efficient paths. In order to use polygonal obstacles in the path planner's occupancy grid,

they must first be indexed into a grid, where each cell is either empty or is occupied by a polygonal obstacle.

Each polygonal obstacle in the environment is decomposed into the cells it occupies using Bresenham's line algorithm [4]. This algorithm is commonly employed by graphics hardware to rasterize polygons into pixels on a screen, but it works equally well for any polygon-to-grid conversion. Each line of the polygon is rasterized into a set of cells that estimate the shape of that line. All cells for every line in the polygonal obstacle are combined together to form the cell decomposition of that obstacle. The set of cells is added to the occupancy grid with their state set as occupied by that specific obstacle.

The rasterization of an environment with polygonal obstacles into an occupancy grid is shown in Figure 8. The occupancy grid and extensions of it provide several uses to the path finding routine, such as finding the traversal cost of a cell and optimizing the detection of collisions.

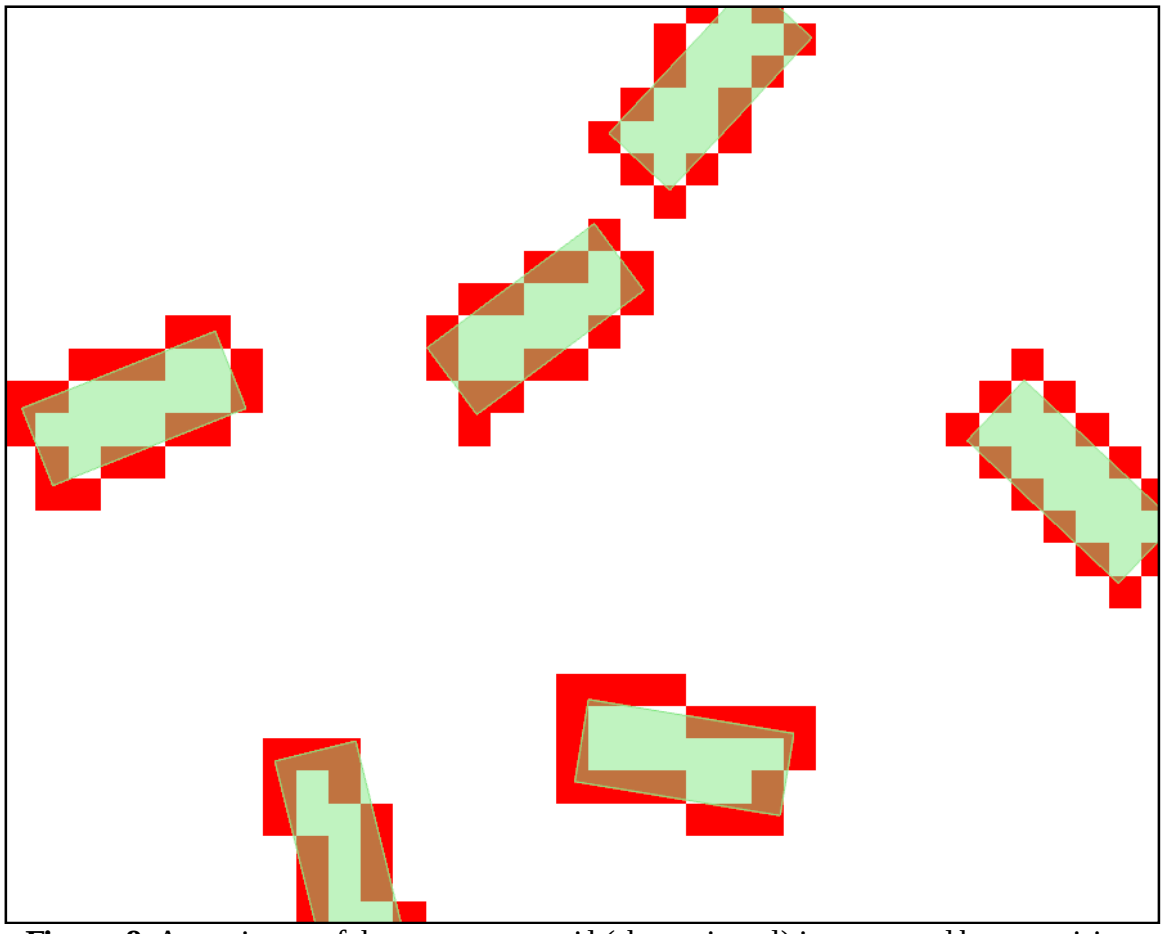


Figure 8. An estimate of the occupancy grid (shown in red) is generated by rasterizing the polygonal obstacles in the environment using Bresenham's line algorithm.

Voronoi Field

As explained in the previous chapter, a potential field is used to push the path search away from obstacles. An extension of the potential field called a Voronoi field is used in this simulation. In order to generate the Voronoi field, two things must first be calculated: a distance map and a generalized Voronoi diagram.

A distance map stores, for each grid cell, the Euclidean distance from that cell to the nearest obstacle cell. Cells that are equidistant from more than one obstacle are part of the generalized Voronoi diagram [13].

The distance map and GVD can be simultaneously constructed using the brushfire method [11]. This algorithm is very similar to Djikstra's algorithm for solving the single-source shortest path problem. However, instead of starting at one node and updating the shortest path to all other nodes, the brushfire algorithm starts at all obstacle cells and recursively updates the distances of all surrounding cells.

The method used in the simulation to generate the distance map and GVD is the improved dynamic brushfire algorithm outlined by Lau, Sprunk, and Burgard in [13]. Their algorithm computes the distance map while dynamically adding cells to the GVD when a cell is computed to have the same distance to more than one obstacle.

Reeds-Shepp Paths

Reeds-Shepp paths [16] are geometrically-calculated optimal paths for a vehicle that can go forward and backward constrained by a minimum turning radius. The introduction of the Reeds-Shepp paths marked an important breakthrough in the field of autonomous car path finding: the optimal path can be analytically computed in constant time for any start and goal poses, even if the poses are very far away. A Reeds-Shepp path consists of combinations of curved and straight path sections, some going forward and some in reverse. The curved sections of the path all have the same constant radius: the minimum turning radius of the vehicle. Reeds and Shepp showed that all optimal paths for a holonomically-constrained vehicle that can move both forward and backward can be described by one of the following patterns [16]:

$$C|C|C, \quad CC|C, \quad CSC, \quad CC_u|C_uC, \quad C|C_uC_u|C,$$

$$C|C\pi_{/2}SC, \quad C|C\pi_{/2}SC\pi_{/2}|C, \quad C|CC, \quad CSC\pi_{/2}|C$$

S represents a straight section of the path; C represents a curved section of the path which can be either a left or right turn; C_u in a path means that there are two curved sections in the path with the same arc length; $C_{\pi/2}$ is a curved section along an arc that subtends an angle of 90 degrees; and | represents a change in gear (switching from forward drive to reverse or from reverse to forward drive). These nine patterns represent 48 separate types of paths that can be computed using 68 formulas, detailed in [16]. These formulas compute the lengths of each primitive action (left turn, right turn, or straight) of a path pattern. To find the optimal path from a start pose to a goal pose, each of these formulas must be computed to find the pattern with the shortest length. (This process can be made more efficient by three-dimensional cell decomposition of the optimal patterns; however, this was not implemented in the simulation and is beyond the scope of this thesis.) Figure 9 shows two examples of Reeds-Shepp paths.

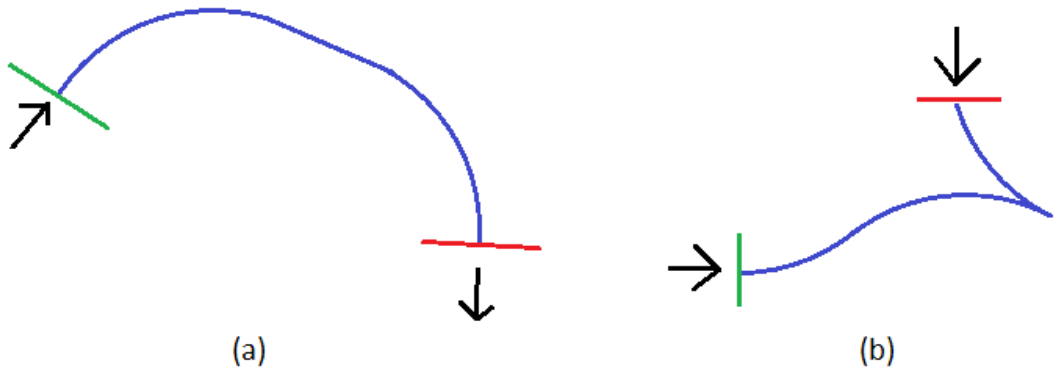


Figure 9. Two Reeds-Shepp paths: the green and red lines show the start and goal position. The arrows show the start and goal orientations. The path patterns used to describe these paths are CSC (a) and $CC|C$ (b). Reeds-Shepp paths are useful for quickly finding the optimal path in an open environment. They can even be used when the vehicle needs to approach the goal pose in reverse, such as backing up into a parking space.

Since Reeds-Shepp paths are computed analytically, they can only be used in environments with no obstacles [6]. However, they provide excellent utility to the path planning routine. While searching for a drivable path through the environment, Reeds-Shepp paths can be used to jump over large open spaces to the goal without having to search through the space in between. They are also used to compute a heuristic used by the A* search algorithm. Both of these uses are discussed in more detail in the following subsections.

Hybrid A*

In this work, a variant of A* search known as Hybrid A* [6] is used to find a drivable path through the environment to the goal pose. The path planner searches through four-dimensional space using the state vector (x, y, θ, r), where $(x \in \mathbb{R}, y \in \mathbb{R})$ is the world position of the center of the vehicle's rear axle, $(\theta \in [-\pi, \pi])$ is the vehicle's orientation with respect to the x-axis, and $(r \in \{0, 1\})$ represents the vehicle's current direction of motion (either forward or reverse).

All nodes discovered during Hybrid A* search are added to a priority queue where a node's priority is calculated by the distance-plus-heuristic cost function:

$$f(x) = g(x) + h(x)$$

The first term is the cost of the path from the starting node to the current node x, where a node j that is a child of i has the following path cost [7]:

$$g(j) = g(i) + dist_{ij}\left(1 + \upsilon \times voro_j + \rho \times rev_j\right) + \sigma \times switch_{ij}$$

This function is dependent on the following: (i) $dist_{ij}$ is the Euclidean distance between nodes i and j; (ii) $rev_j = 1$ if node j's direction of motion is in reverse or 0

otherwise; (iii) $voro_j$ is the value of the Voronoi field at node j; (iv) $switch_{ij} = 1$ if the directions of motions of nodes i and j are different or 0 otherwise; and (v) ρ, υ, and σ are constant penalty factors. The $voro$ term pushes the search away from obstacles and toward the centers of narrow driving corridors. The rev term is a multiplicative cost that penalizes paths driven in reverse, since reverse driving is inherently slower and more dangerous than forward driving. And the $switch$ term penalizes frequent changes in the direction of motion to account for the time it takes to come to a complete stop to change gear [6]. The constants ρ, υ, and σ control how much each term influences the path cost.

The second term of $f(x)$ is an admissible heuristic function that estimates the cost of the path from node x to the goal node. The following section discusses the two heuristics used in the simulation.

Heuristics

Two heuristics are used while searching to focus Hybrid A* toward a feasible path: the *non-holonomic-without-obstacles* heuristic and the *holonomic-with-obstacles* heuristic. This section explains how each heuristic is calculated. Since both heuristics are admissible, the maximum of the two heuristics is used.

Relaxing the Obstacle Constraint

The non-holonomic-without-obstacles heuristic is generated by relaxing the obstacle constraint: all obstacles are ignored, but the non-holonomic limitations are still accounted for. This heuristic is computed offline by finding the optimal path from a goal cell to all other cell's within some neighborhood of the goal. There are two reasons why only a neighborhood surrounding the goal is precomputed. Firstly, this heuristic is mostly useful only as the search approaches the goal: the correct search direction is usually

toward the goal when the search is far away from the goal. When the search is near the goal, the search must then take into account the orientation of the goal pose. Secondly, a smaller precomputed heuristic grid takes less time to load into memory and query.

Since this heuristic operates in an empty environment, not all possible goals need to be handled. In fact, only one goal pose is considered: $(x = 0, y = 0, \theta = 0)$. A three-dimensional grid is formed around this goal, and the optimal path from the center of each grid cell to this goal is calculated using Reeds-Shepp curves. Whenever a lookup is performed on this heuristic, the entire heuristic grid is translated and rotated to match the actual goal pose of the mission.

Relaxing the Turn Radius Constraint

By ignoring the vehicle's turn radius limitations, a set of estimated solutions to the path planning problem can be quickly discovered on the occupancy grid using dynamic programming. This forms the basis of the holonomic-with-obstacles heuristic. The vehicle is treated as a holonomic agent: it can move in any direction without having to change position. The obstacles in the environment, indexed by the occupancy grid, are used to find possible 8-neighbor paths from the start position to the goal position. While the non-holonomic-without-obstacles heuristic gave a three-dimensional estimation, this heuristic is only two-dimensional, ignoring the vehicle's orientation. The following pseudocode shown in Figure 10 describes the method (a form of Djikstra's algorithm) that calculates the holonomic-with-obstacles heuristic.

This heuristic is environment-specific, so it must be computed at the beginning of every mission. To increase efficiency, the list of neighboring grid cells returned by the `get8Neighbors` function is memoized for each cell to speedup future neighbor lookups

of the same cell. After this function executes, h will contain the heuristic value for every cell in the environment grid. However, as it currently stands, this heuristic is not admissible. This is because movement from one cell to another is limited to only eight directions, whereas a vehicle can move in any direction. Figure 11 shows an example of a path that has an overestimated heuristic value.

```
function holonomicWithObstacles(goal):
    for each cell in grid:
        h[cell] := ∞

    closed := empty set
    open := {goal}
    h[goal] := 0

    while open is not empty:
        current := cell in open with lowest h-value
        remove current from open
        add current to expanded

        for each neighbor in get8Neighbors(current):
            if neighbor not in closed and
               neighbor not occupied by obstacle:
                if neighbor is adjacent to current:
                    dist := 1
                else: // neighbor is diagonal to current
                    dist := √2

                dist := dist + h[current]
                if dist < h[neighbor]:
                    h[neighbor] := dist
                    add neighbor to open
```

Figure 10. Pseudocode for the holonomic-with-obstacles heuristic

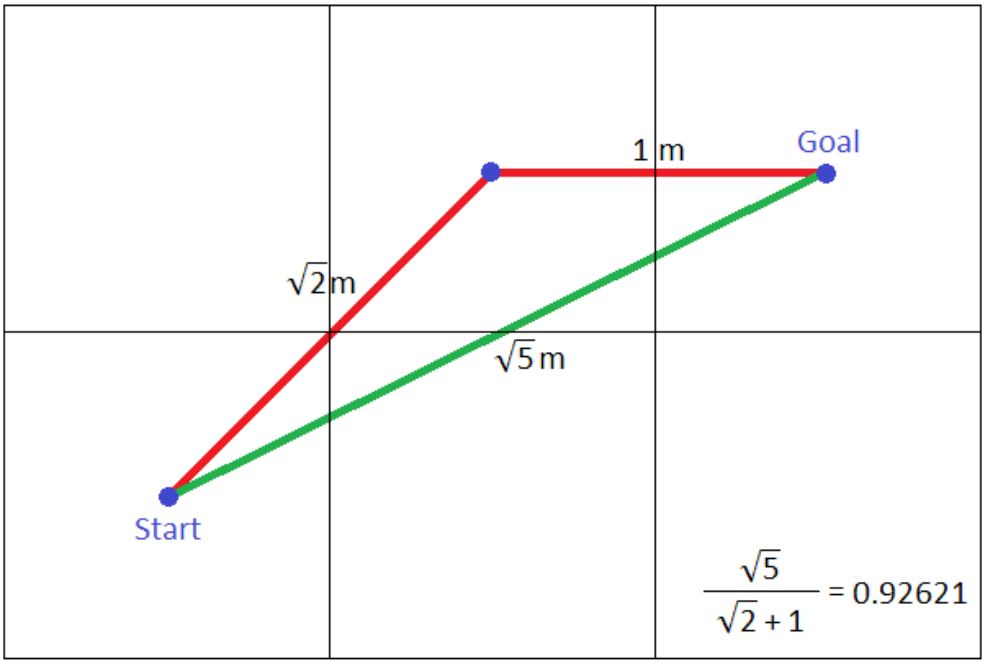


Figure 11. The red line represents the heuristic's shortest path to the goal, and the green path is the actual shortest path to the goal. The heuristic overestimates the path cost because movement between cells is restricted to only eight directions. The shortest path is 0.92621 times as long as the value given by the holonomic-with-obstacles heuristic.

The figure actually shows the situation with the *largest possible overestimation* that the heuristic can produce. Any path shorter than the red path would involve only a single movement from the start cell to the adjacent goal cell, in which case the heuristic path and the optimal path would be the same. Paths longer than the red path (such as a path that extends just one more cell to the right) would give higher overestimation ratios. The ratio of the actual shortest path cost to the heuristic value will never be lower than 0.92621. Therefore, to make this heuristic admissible, every heuristic value looked up during the A* search is discounted by this factor. Now this heuristic will never overestimate the actual path cost and will still provide the advantage of pushing the search away from dead-ends.

Hybrid A Algorithm*

Hybrid A* is a variant of the standard A* algorithm that operates on continuously valued nodes. Standard A*, which uses discrete search nodes, is not appropriate for non-holonomic vehicle path planning, since a car-like vehicle can achieve any continuous orientation or position in the world space. To use A* search in a continuous world, each discrete grid cell has a continuous vehicle pose associated with it. Steering and motion controls are performed on these vehicle poses to expand a node, generating child nodes. If a safe child node falls into a grid cell that is already occupied by another continuous vehicle node, the node that has the lower f-value (signifying that its path will probably lead to the goal with less cost) is kept, and the other is discarded. This situation is illustrated in Figure 12.

When a node is expanded, six child nodes are generated (Figure 13): three moving forward (left turn, straight, and right turn) and three moving backward (left turn, straight, and right turn). The steering control applied to left and right turns is a constant turning radius equal to slightly less than the maximum turning radius of the vehicle. This is used to partially compensate for the time it takes to turn the steering wheel from one extreme to the other. The motion control is applied for some predetermined length (empirically, the length of a Hybrid A* grid cell works well). For straight nodes, the vehicle is simply translated in the direction of motion; turning nodes are expanded by moving the vehicle along an arc of a circle. The algorithm for Hybrid A* is shown in Figure 14.

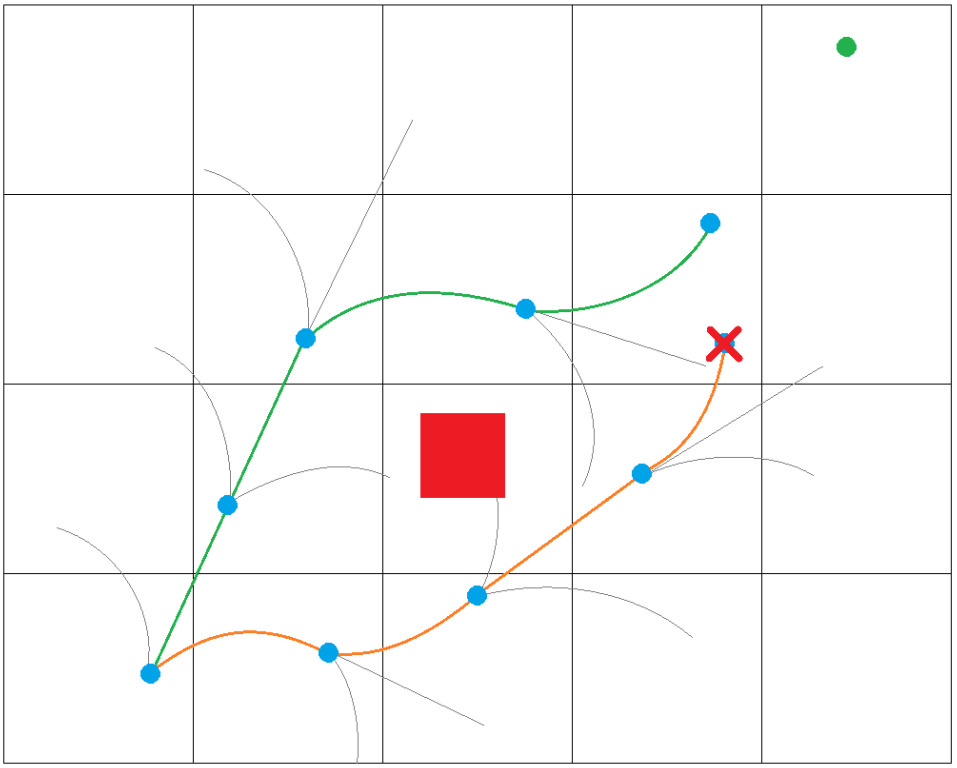


Figure 12. A search discovering a path around the red obstacle to the green goal node in the top-right grid cell. Both the green path and the orange path expand nodes to reach the same grid cell. But since the node at the end of the green path is closer to the goal (representing a lower f-value), it will be kept as that grid cell's continuous pose, and the node terminating the orange path will be discarded. The orange path will never be considered again; however, its branching nodes may still play a part in the search.

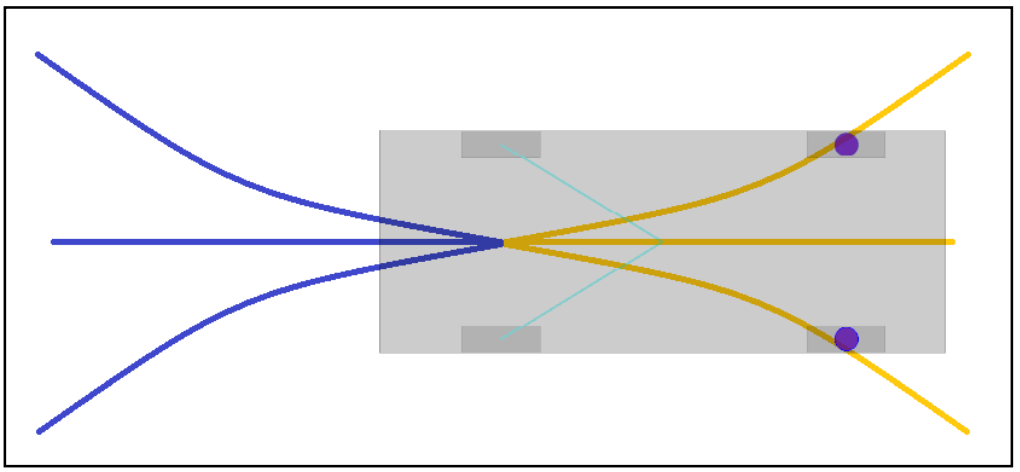


Figure 13. During Hybrid A* search, expanding a node with three steering controls (left, straight, and right) and two motion controls (forward and reverse) creates six child nodes.

```
function hybridAstar(start, goal):
    startCell := cell that start pose occupies
    assoc[startCell] := start

    open := empty priority queue
    add startCell to open with priority 0

    while open is not empty:
        currentCell := cell in open with lowest priority
        remove currentCell from open
        current := assoc[currentCell]
        if current == goal:
            return reconstructPath(current)

        for each child in getChildren(current):
            g := current.g + path cost from current to child
            f := g + h(child)
            childCell := cell that child occupies
            assocNode := assoc[childCell]

            if assoc[childCell] == null:
                child.g := g
                child.f := f
                child.from := current
                assoc[childCell] := child
                add childCell to open with priority f
            else if f < assocNode.f:
                child.g := g
                child.f := f
                child.from := current
                assoc[childCell] := child
                if (childCell is in open):
                    update priority of childCell node to f
                else:
                    add childCell to open with priority f

    return null since no path could be found

function getChildren(node):
    children := empty list
    for each steer in [straight, left, right]:
        for each gear in [forward, backward]:
            child = apply steer and gear to node
            if child is safe:
                add child to children

    if node should expand Reeds-Shepp curve:
        child := Reeds-Shepp solution from node to goal
        add child to children

    return children
```

Figure 14. Pseudocode for the Hybrid A* algorithm

As discussed in [6], Hybrid A* search "uses a discretized space of control actions," meaning that the search "will never reach the exact continuous-coordinate goal state." Only six children are expanded from each search node, and the goal pose could never be exactly achieved with such a low resolution search. To account for this inaccuracy of the search, a seventh child node is generated for some nodes by solving the optimal Reeds-Shepp path from the current pose to the goal pose. If the calculated Reeds-Shepp path results in no collisions with obstacles, it is added as a node to the A* search. The calculation of a Reeds-Shepp curve is done in constant time; however, it is an expensive computation because dozens of equations must be solved to find the shortest path from the node to the goal. Because of this, not every node is expanded with a Reeds-Shepp child. The probability of adding a Reeds-Shepp child node is a function of the heuristic distance to the goal [6]. This works well because Reeds-Shepp children are computed more frequently as the search approaches the goal, where the curve is less likely to collide with obstacles. Because the Reeds-Shepp path ends at the goal pose, it is always the next node to be expanded, which finishes the search early without having to expand nodes all of the way to the goal. The addition of a Reeds-Shepp child node enhances search speed and provides a path that terminates precisely at the goal pose. Figure 15 shows how a Reeds-Shepp node improves Hybrid A* search.

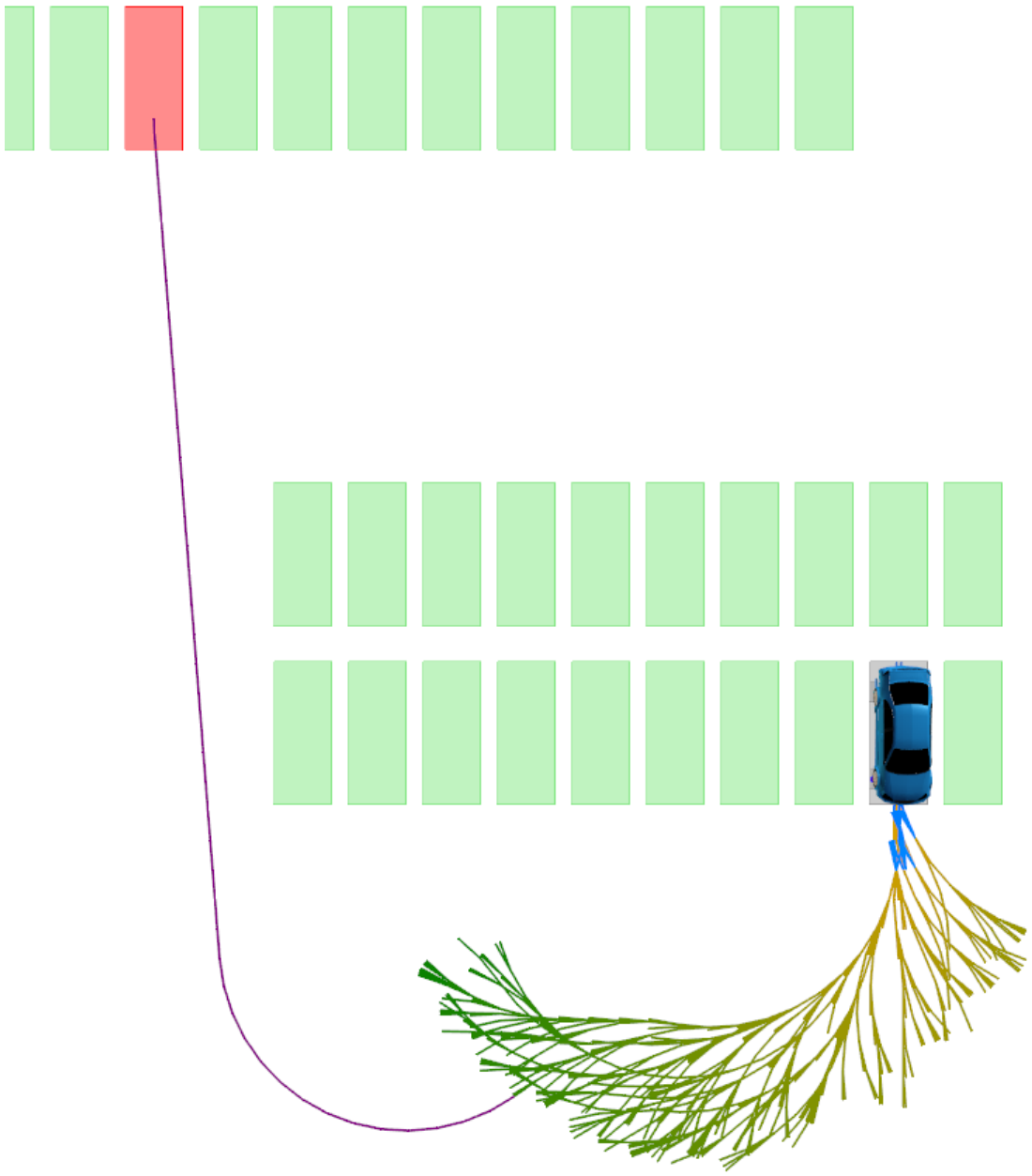


Figure 15. Hybrid A* search is used to find a path from one parking space to another. The purple line is the Reeds-Shepp path from one of the expanded nodes to the goal pose. Notice how the search can skip over much of the environment with one node; the Reeds-Shepp child gives a significant performance boost to Hybrid A*.

The Hybrid A* algorithm combines the efficiency of discrete A* search with the accuracy of continuous vehicle poses. However, as stated in the previous chapter, Hybrid A* is neither optimal nor complete due to the discretization of the environment and the vehicle controls. Perfect completeness could never be attained since Hybrid A* operates in a continuous world that is represented by a discrete grid. However, Hybrid A* is probabilistically complete, meaning as the resolution of the grid becomes finer, the probability of finding a solution increases. Unfortunately, very fine resolutions are impractical due to computational limitations. A resolution of 0.5 meters to 1 meter is sufficient to find paths in most realistic scenarios.

To address the issue of optimality, some post-processing must be done on the discovered path.

Smoothing

The path produced by Hybrid A* is not guaranteed to be optimal; however, it almost always lies close to the global optimum [6]. The near-optimal path is improved using numerical optimization to make it smoother and push it even closer to the global optimum. To smooth the A* path, gradient descent is used to minimize the following objective function [6]:

$$f(i) = w_{\mathcal{O}} \sum_{i=1}^{N} (|\boldsymbol{x}_i - \boldsymbol{o}_i| - d_{max})^2 + w_{\mathcal{S}} \sum_{i=1}^{N-1} (\Delta \boldsymbol{x}_{i+1} - \Delta \boldsymbol{x}_i)^2 + w_{\rho} \sum_{i=1}^{N} \rho_V(x_i, y_i) + w_{\kappa} \sum_{i=1}^{N-1} (\kappa_i - \kappa_{max})^2$$

Given a sequence of path vertices $\boldsymbol{x}_i = (x_i, y_i), i \in [1, N]$: $\boldsymbol{o}_i$ is the obstacle nearest to the vertex; $\Delta \boldsymbol{x}_i = \boldsymbol{x}_i - \boldsymbol{x}_{i-1}$ is the displacement vector at the vertex; $\rho_V(x_i, y_i)$

is the value of the Voronoi field at the vertex; κ_i is the instantaneous curvature at the vertex; κ_{max} is the maximum allowed curvature limited by the turning radius of the vehicle; d_{max} is the maximum distance of the collision field of an obstacle; and w_O, w_S, w_ρ, and w_κ are constant weights. In order to push the path closer to the global optimum subject to the factors of the objective function, $\sum f(i)$ for all $i \in [1, N]$ must be minimized.

The first term penalizes collisions with obstacles and seeks to push vertices away that are too close (when $|\boldsymbol{x}_i - \boldsymbol{o}_i| < d_{max}$). The second term is a measure of the smoothness of the path and seeks to have the path travel as straight as possible. The third term pushes the vertices toward low areas in the Voronoi field, as far away from nearby obstacles. The last term limits the maximum allowed curvature, avoiding turns that are too sharp for the vehicle to make. The term weights w_O, w_S, w_ρ, and w_κ determine how each term influences the optimization. The following values for these weights were found empirically during testing: $w_O = 0.002$, $w_S = 4$, $w_\rho = 0.2$, and $w_\kappa = 4$.

In order to use the objective function in gradient descent, the derivative of each term in the function must be found. The derivatives of some of the terms are very involved, and their solutions are beyond the scope of this thesis; however, the differentiation of each term can be found in [7] and [6]. The optimization technique used in this simulation is iterative gradient descent instead of the more complex conjugate gradient descent used in [6]. This was to avoid the need for a third-party numerical optimization library. Iterative gradient descent is not as efficient as conjugate gradient; however, it still provides good smoothes in a low number of iterations.

While smoothing, the start and goal poses are kept constant; they are not affected by the smoothing. Also, any poses where the vehicle switches from forward to reverse driving are also kept constant, which keeps the cusp points from being changed by the curvature term of the objective function. To smooth the path discovered by Hybrid A*, the algorithm defined in Figure 16 is used:

```
function smooth(vertices):
    for count := 0 to numIterations:
        for v in vertices:
            correction := -w_o * d_obstacle(v)
            correction := correction - w_s * d_smooth(v)
            correction := correction - w_v * d_voronoi(v)
            correction := correction - w_k * d_curvature(v)
            normalizer := w_o + w_s + w_v + w_k
            correction := correction / normalizer
            v := v + alpha * correction
    return vertices
```

Figure 16. Pseudocode for the smoothing function. The `d_` functions return the gradients for each of the terms in the object function. The `w_` constants are weights which decide how much each term influences the correction. The `alpha` constant (`alpha` ≤ 1) controls how quickly the smoother approaches an optimal path; it must not be set too high to avoid oscillations around the optimum between successive iterations of the outer loop.

As shown in Figure 17, smoothing removes unnecessary curves and kinks in the discovered path, artifacts of inaccurate discretization of the environment. By removing excessive swerving in the path, the vehicle can more closely follow the intended trajectory and travel at a higher speed. Also, the original discovered path assumes the steering wheel can be instantaneously actuated to the correct position, which is impossible since the steering wheel can only be turned at some maximum angular velocity. The optimized path requires much slower turning of the steering wheel, smoothing out the vehicle's jerky steering.

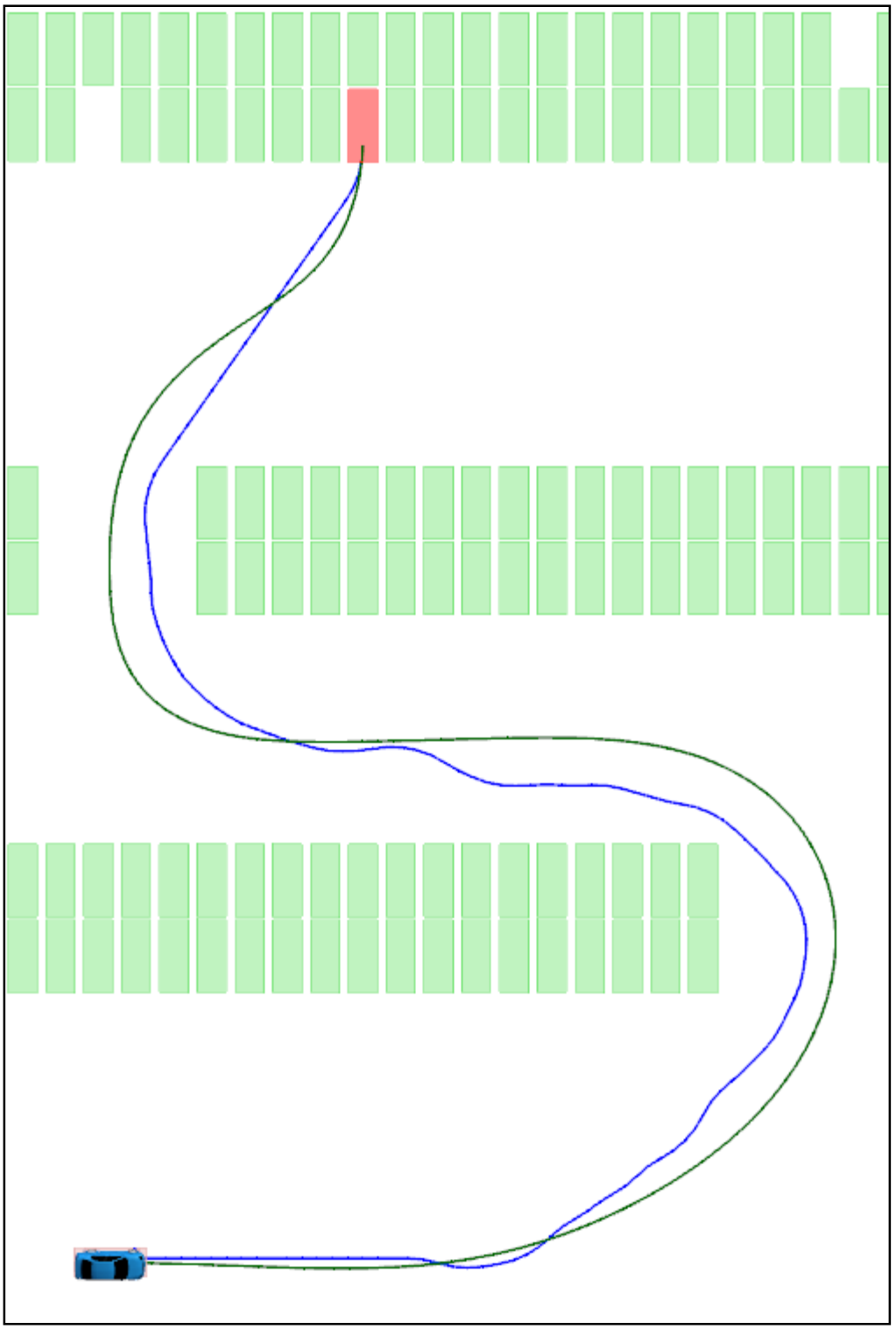


Figure 17. The unsmoothed path (blue) and the smoothed path (green). Path smoothing allows for faster speeds, more accurate path tracking, and a more comfortable ride.

Vehicle Controller

Once a feasible path has been discovered and smoothed by the path planner, it is delivered to the vehicle controller. The vehicle controller provides steering, gas, and brake controls to track the planned path as accurately and as quickly as possible. New vehicle controls are generated for each frame of the simulation (up to 60 frames per second). The steering controller used is the Stanley method, introduced in the previous chapter, coupled with a proportional-derivative (PD) controller. The gas and brake are controlled by a proportional-integral (PI) controller that tracks a recommended velocity for each vertex in the path. These will be discussed in more detail in the following subsections. But first, the physics engine and the role is plays in the vehicle controller will be introduced.

Physics Engine

Farseer Physics [8] is a 2D XNA physics engine used for collision detection and applying vehicle controls to the simulated vehicle. In the top-down, two-dimensional world handled by the physics engine, the vehicle is a simple rectangular body with four rectangular wheels. The two back wheels are attached to the body using fixed joints and have no rotational movement. The two front wheels are attached using revolute joints, which allow them to rotate to give the vehicle steering. An axis-aligned tensor damping controller is added to the four wheels; this controller provides friction perpendicularly to each wheel to simulate grip. No friction is added in the direction the wheel is facing to simulate rolling. Steering controls from the vehicle controller rotate the front wheels left and right; friction from turned wheels causes the rear acceleration force to push the front of the car sideways causing the vehicle to turn. The gas control applies a force to the

vehicle body in the direction it is facing causing it to accelerate. The brake control gives a force in the opposite direction until the vehicle comes to a stop.

Steering Controller

Steering is the most important part of path following. To steer the vehicle along the path, a proportional-derivative (PD) controller is used. The proportional term is used to correct the steering to the desired angle, and the derivative term is used to smooth out the controls over time to avoid sharp steering. The Stanley method is used to determine the desired steering angle.

Tracking the Front Path

The Stanley method was initially developed for Stanford's entry into the 2005 DARPA Grand Challenge [2], an autonomous vehicle race on an off-road course. However, it was quickly evident that the standard Stanley method would be insufficient for accurate path tracking in a parking lot type environment. Therefore, the following change was made: instead of tracking the center of the front axle to the discovered path, the front axle is tracked to a transformation of the path. The original path generated by Hybrid A* contains vertices to be tracked by the center of the rear axle. So, each vertex in the path is translated in the direction of motion by the distance between the rear and front axles to produce a path to be tracked by the center of the front axle. This allows for closer path tracking since the steering is influenced by the front wheels. If the center of the front axle closely tracks the front axle path using the Stanley method, then the rear axle will closely track the path returned by Hybrid A*, guaranteeing safe driving from the start pose to the goal pose.

The vertices of the path found by the Hybrid A* search are four-dimensional (position in 2D space, orientation, and direction of motion). However, the smoothing operation causes the orientation dimension of the vertices to be lost; smoothing translates the positions of the poses, so the orientations associated with each pose no longer apply. An orientation is needed for each vertex in order to generate a front path to be tracked by the center of the front axle. The following equation is used to estimate the vehicle's orientation at vertex i.

$$\theta(i) = \tan^{-1}\frac{y_{i+1} - y_{i-1}}{x_{i+1} - x_{i-1}}$$

The estimated orientation is simply the average of the displacement vectors between the vertex's previous and subsequent path points. This orientation is calculated at each vertex in the original path to translate the point in the estimated direction of motion. The first and last vertex in the path use the orientations of the start and goal pose, since those poses are not moved by the smoothing process. Cusp points (vertices where the vehicle changes between forward and reverse driving) are also kept immobile during smoothing, so their pose orientations are used as well when generating front path vertices. Figure 18 shows the planned paths for the centers of both the rear and front axle in an example scenario.

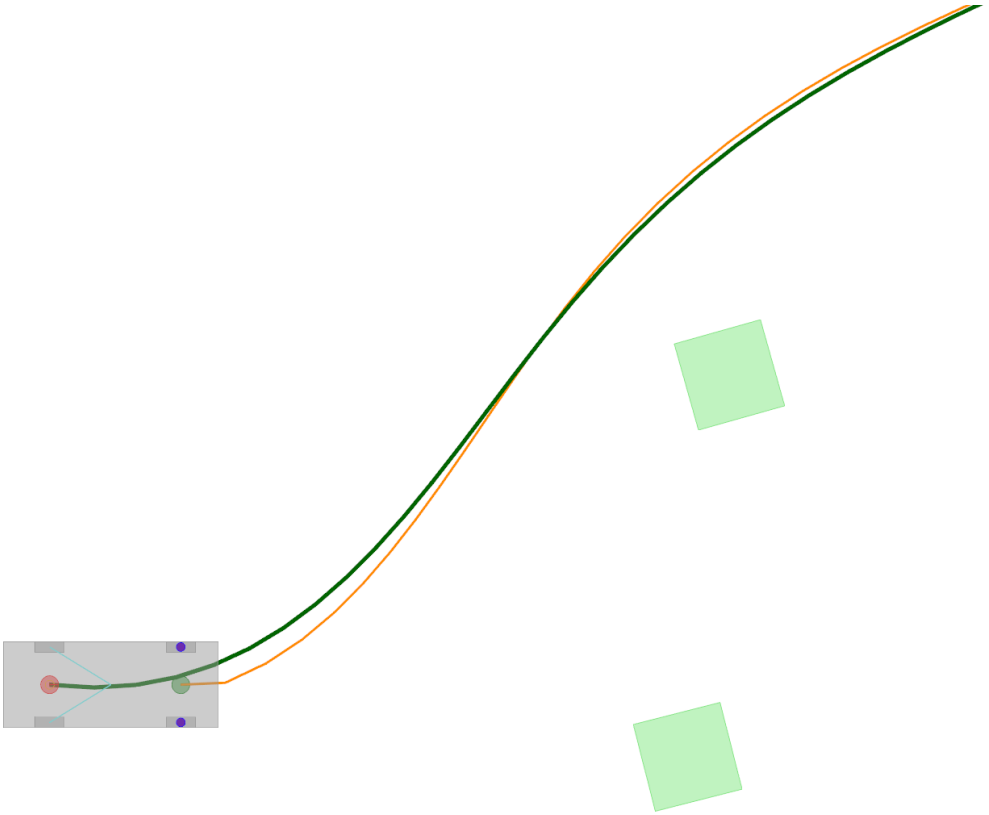


Figure 18. The rear path discovered and smoothed by the path planner is shown in green. The vehicle controller calculates the front path (in orange) by translating the rear path points in the estimated direction of motion. The front path is used to calculate the error between the desired and actual wheel angle, which is used to determine the correct steering control to correct the error. When following a path, the front of the vehicle will swing wide in turns to keep the center of the rear axle on the rear path.

Path Following Algorithm

Once the front path is generated, the algorithm presented in Figure 19 is used to control the steering of the vehicle on each update frame of the simulation.

```
function getSteeringControl(pose, wheelAngle, elapsedTime):
    i := target vertex (the next path vertex the car will meet)
    howFarAlong := fraction of the path the vehicle has driven
                   from the previous vertex to the next vertex
    frontAxlePos := position of front axle of pose
    closestFrontPoint := point on the line between the previous
                         and next vertices of the front path that
                         is closest to frontAxlePos
    distance := dist(frontAxlePos – closestFrontPoint)

    prevHeading := FrontPath[i] – FrontPath[i – 2]
    nextHeading := FrontPath[i + 1] – FrontPath[i – 1]
    headingDiff := nextHeading - prevHeading
    desiredHeading := prevHeading + headingDiff * howFarAlong
    thetaDiff := pose.orienation – desiredHeading
    desiredAngle := thetaDiff + atan(k * distance)

    error := desiredAngle – wheelAngle
    dError := (error – prevError) / elapsedTime
    prevError := error

    return w_error * error + w_dError * dError
```

Figure 19. Pseudocode for the steering controller

The `getSteering` function accepts three arguments: the current three-dimensional pose of the vehicle, the vehicle's current wheel angle, and the amount of time elapsed since the previous update frame. The desired wheel angle is calculated by the following equation, a variant of the Stanley equation introduced in the previous chapter:

$$\delta = \theta_e + \tan^{-1}(ke_{fa})$$

This equation differs from the Stanley method in that speed is not accounted for. In the simulation implementation, the vehicle never reaches speeds over 10 miles per hour, while Stanley reached higher speeds on the off-road course of the DARPA Grand Challenge. Very low speeds tended to cause the vehicle to oscillate around the desired path when using the original equation, so the speed term was omitted in favor of more stable steering.

θ_e is calculated in much the same way that orientations are calculated to generate the front path. It is computed by using linear interpolation between the estimated orientations at the previous path point and the next path point, using the distance traveled so far between those two points to determine the weight of each.

The second term in the equation influences the steering in proportion to the cross track error: the farther the vehicle is away from the desired path, the more the controller steers toward it. When the vehicle is on the path (a cross track error of zero), then the desired heading is only influenced by the estimated orientation θ_e. The constant k is used to control how quickly the controller steers back toward the path.

Once the desired orientation is found, the difference between it and the angle of the wheels is used in a PD controller to generate a steering control. The proportional term is used to rotate the wheels toward the desired direction of motion in proportion to the current error. The derivative term is used to smooth out jerky steering by easing in and out of error corrections. The `w_error` and `w_dError` constants control how much each term affects the final steering control.

The steering value generated by the steering controller is used to drive a rotational motor attached to the front wheels in the physics engine. The steering control is clamped

to between [-1, 1], where -1 is maximum turn speed to the left and 1 is maximum turn speed to the right. The maximum turn speed of the wheel motor is set to 1.2 radians per second in the simulation, or about one revolution of the steering wheel every fifth of a second (at a steering ratio of 14:1). When the physics simulation updates a frame, this turn speed is used to rotate the front wheels, which provides a new direction of motion.

Path Following in Reverse

Following a path in reverse is slightly different from tracking a path while driving forward, as anyone who has driven a vehicle knows. The path following will have a large amount of error if the vehicle controller tries to track the center of the front axle to the front path while traveling in reverse. This is because the front path was generated by translating the rear path vertices *forward* in the direction of orientation.

To solve this problem, a simple technique was developed: the representation of the vehicle is mirrored along the axis of the line passing through the rear axle. A vehicle with its wheels turned at a given angle will track its rear axle along a circle of constant radius regardless of its driving direction. Therefore, two vehicles with the same rear axle position but opposite orientations (Figure 20) will travel in opposite directions along the same circle as long as their front axles have equal but opposite wheel angles. A "fake" front axle center is used to determine the cross track error between the mirrored vehicle and the desired path. This "fake" front axle is produced by translating the position of the rear axle in the opposite direction of orientation. In order to determine the path this "fake" front axle must follow, the vertices of the rear path are translated in the *reverse* direction of orientation to produce a counterpart to the front path called the reverse front path.

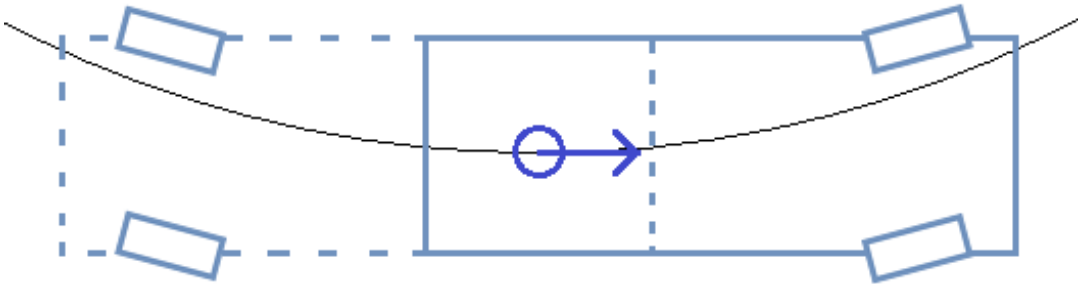


Figure 20. The vehicle controller drives in reverse by pretending it is driving forward but in the opposite direction. The calculated wheel angle is mirrored to produce the correct steering control that will track the center of the rear axle along the desired path.

The algorithm described above for forward driving is also applied to reverse driving; however, the front path is replaced by the reverse front path and the front axle position is replaced by the "fake" front axle position. The desired wheel angle calculated in the algorithm is mirrored (a desired 15 degree left turn would turn into a 15 degree right turn) to take into account that the vehicle is actually traveling in reverse. Figure 21 shows an example of the vehicle backing out of a parking space.

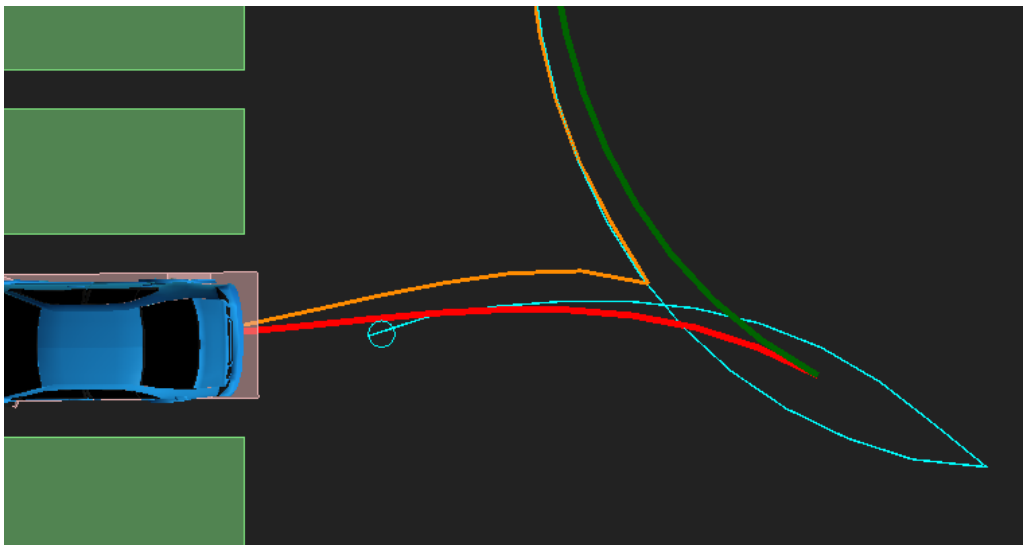


Figure 21. The vehicle backs out of a parking space. The red and green path (reverse and forward driving) is the desired route of the center of the rear axle. The cyan path is the reverse front path that is tracked by the "fake" front axle (cyan circle). Once the vehicle reaches the cusp, it will use the front path (in orange) to steer while driving forward.

Velocity Controller

As discussed in the previous chapter, each vertex in the desired path has an associated recommended velocity. This velocity can be generated from several factors such as proximity to obstacles or path curvature. Each recommended velocity is used to determine the desired speed of the vehicle as it travels along the path. The vehicle should slow down in lower velocity sections of the path and speed up in faster velocity sections.

For the velocity controller implemented in the simulation, three factors determine the recommended velocity at each point on the path: the global speed limit, the instantaneous curvature of the path at that vertex, and a deceleration constraint based on the velocity of the succeeding states in the path.

Global Speed Limit

The global speed limit is imposed on all path points regardless of the features specific to each point. When in forward drive, the speed limit of the vehicle is 10 miles per hour; the speed limit is 2.5 miles per hour when driving in reverse.

Path Curvature

Given that $\Delta \mathbf{x}_i = \mathbf{x}_i - \mathbf{x}_{i-1}$ is the displacement vector at point i of the path, the instantaneous curvature at i can be estimated with the following equation [6]:

$$k = \frac{\left|\tan^{-1}\frac{\Delta y_{i+1}}{\Delta x_{i+1}} - \tan^{-1}\frac{\Delta y_i}{\Delta x_i}\right|}{\|\Delta \mathbf{x}_i\|}$$

The curvature is estimated as the change in tangential angle at point i divided by the length of the displacement vector at point i. The reciprocal of curvature ($r = \frac{1}{k}$) is the radius of curvature. This radius of curvature is used as a measure of how sharp a curve is. When a vehicle enters a sharp turn, it needs to slow down to successfully maneuver the

turn and stay on course. The radius of curvature can be used to impose some arbitrary maximum lateral acceleration defined by $a = \frac{v^2}{r}$. This equation can be solved for v to find the recommended velocity with respect to path curvature for each path point:

$$v = \sqrt{a_{max} \frac{\|\Delta \mathbf{x}_i\|}{\left|\tan^{-1}\frac{\Delta y_{i+1}}{\Delta x_{i+1}} - \tan^{-1}\frac{\Delta y_i}{\Delta x_i}\right|}}$$

The constant a_{max} is the maximum lateral acceleration the vehicle can achieve when going around a curve. Being constrained by this lateral acceleration causes the vehicle to have to slow down during sharp curves in the path. In the simulation, the maximum lateral acceleration is set to 0.5 $^{m}/_{s^2}$.

Deceleration Constraint

It is not enough to limit the velocity at the sharp sections of a curve; the vehicle must also slow down before it enters the turn in order to track to the correct speed at the correct location. In order to have the vehicle slow down as it enters slower sections of the path, a deceleration constraint is also imposed on the path, where the maximum velocity of a path point is limited by the succeeding point with some deceleration constant added on. This constraint ensures that the vehicle gradually slows down as it reaches slower sections of the path without having to slam on the brakes. The following kinematic equation shows how the deceleration constraint velocity is calculated for vertex i:

$$v_{decel}(i) = \sqrt{{v_{i+1}}^2 + 2a_{decel}\|\boldsymbol{x}_{i+1} - \boldsymbol{x}_i\|}$$

The recommended deceleration velocity is calculated by determining the speed at vertex i needed to achieve the speed at vertex $i+1$ by decelerating at a_{decel} $^{m}/_{s^2}$. In order to calculate this, the last path point velocity must be calculated first, and then the

velocity recommender must iterate backwards until it reaches the first path point. The last path point is always set to zero (since the vehicle should always stop at the goal pose). The second-to-last path point is then considered, and the maximum velocity is found that will allow the vehicle to come to a stop by slowing down by the constant deceleration. Then the velocity at the third-to-last path point is determined by considering the velocity at the second-to-last path point, and so forth.

These three factors each recommend a maximum velocity; the actual velocity limit attached to each path point is the minimum of these three. In addition, some path points have a maximum velocity of zero, such as the goal position and points where the vehicle needs to stop in order to change the direction of motion. Figure 23 shows the recommended velocities of a path through a parking lot. Figure 22 is the pseudocode of the velocity controller showing how it uses the error between the desired speed and actual speed to generate motion controls.

```
function getMotionControl(speed, elapsedTime)
    prev := previous path point
    next := next path point
    howFar := fraction of the path the vehicle has driven
              from the previous vertex to the next vertex
    desiredSpeed = v[next] * howFar + v[prev] * (1 - howFar)
    vError := speed - desiredSpeed
    vPastError := vError * elapsedTime
    vTotalError := w_vError * vError + w_vPastError * vPastError
    if vTotalError > 0:
        brake := vTotalError
        gas := 0
    else:
        brake := 0
        gas := -vTotalError
    return (gas, brake)
```

Figure 22. Pseudocode for the velocity controller

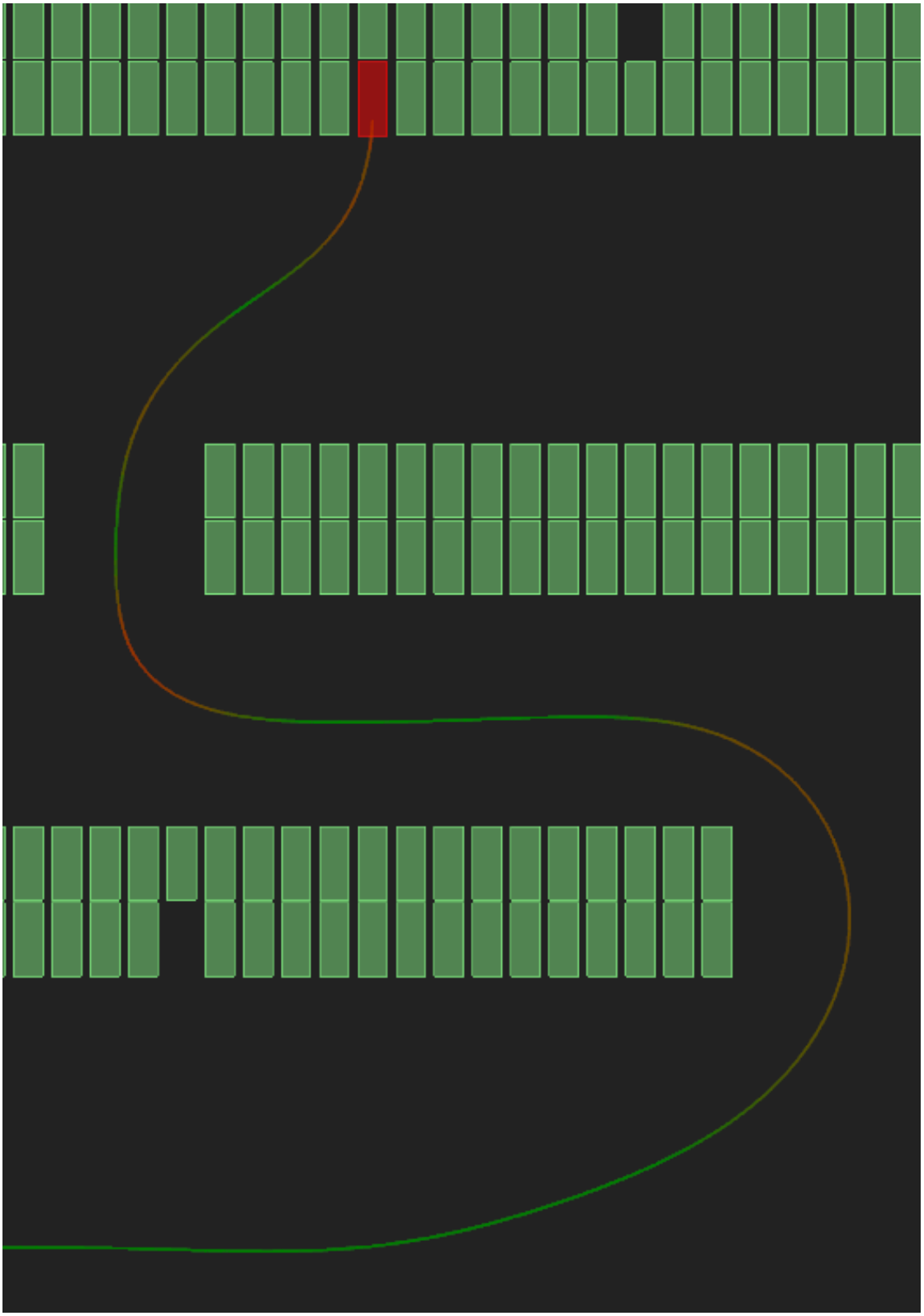


Figure 23. This path has target velocities shown with different colors. Green sections have a higher velocity, and red sections have a slower velocity.

Finite State Machine

At a high level, the vehicle controller is implemented as a finite state machine. This FSM provides definitions for the possible vehicle states as it travels from the start to the goal with clear conditions for state transitions. The vehicle controller finite state machine has five possible states, as shown in Figure 24.

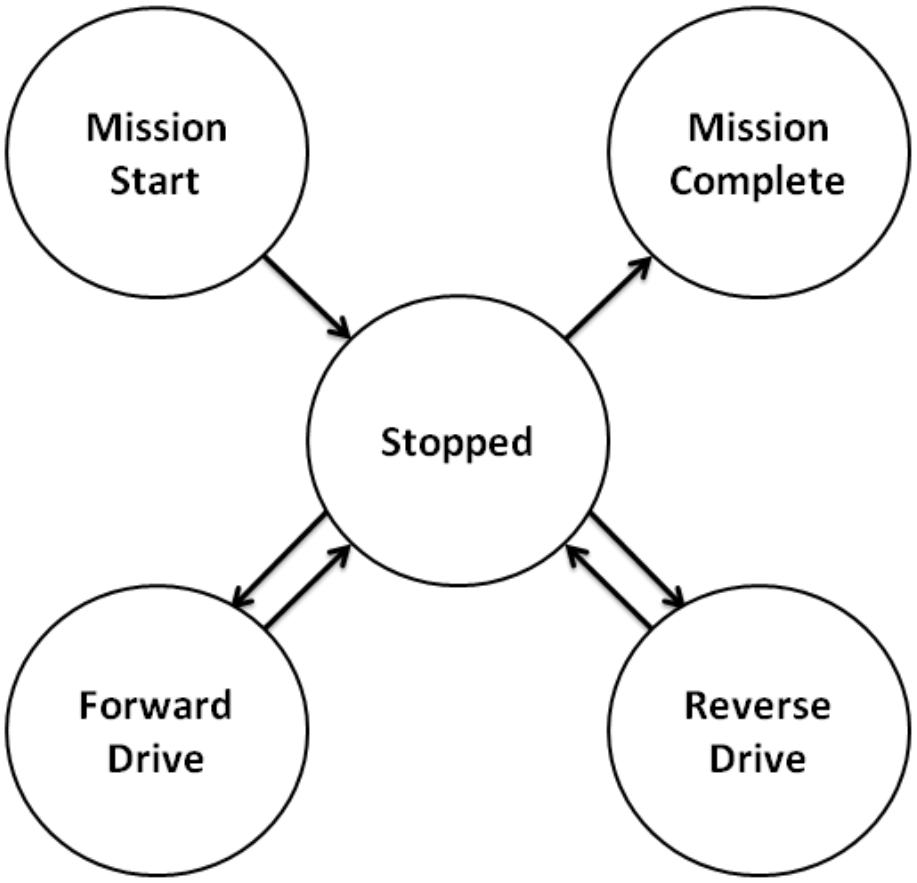


Figure 24. The finite state machine implemented by the vehicle controller. Once the mission is loaded, the controller transitions from *Mission Start* to *Stopped*. It then loops through *Stopped*, *Forward Drive*, and *Reverse Drive* as it travels along the path until it reaches the goal pose, where it enters the *Mission Complete* state to finish the mission.

The state definitions and transition conditions are as follows:

- *Mission Start*: this is the initial state when the mission is loaded. In this state, the controller calculates the front path, the reverse front path, and the recommended velocity of each path point.
    - This state transitions to *Stopped* after the controller initializes.
- *Stopped*: this state is reached when the vehicle is at the start pose, the goal pose, or any cusp point of the path (where the vehicle switches from forward gear to reverse gear). If the vehicle is in this state and has not yet reached the goal pose, it will turn the steering wheel while stationary to orient its wheels towards the next path point. This helps the vehicle maintain accurate path tracking as it is leaving the start pose or a cusp point.
    - When the wheels are correctly oriented, this state transitions to *Forward Drive* or *Reverse Drive* depending on the expected gear of the next section of the path.
    - This state transitions to *Mission Complete* when the vehicle is stopped at the goal pose and the wheels are reoriented to pointing straight forward.
- *Forward Drive*: this is the vehicle's state when it is tracking the path in forward gear. The algorithm previously described is used to track the center of the front axle to the calculated front path.

    - This state transitions to *Stopped* when the vehicle comes to a stop at a cusp point or the goal pose.

- *Reverse Drive*: this is the vehicle's state when it is tracking the path in reverse gear. To track the path in reverse, the algorithm is modified to have the "fake" front axle track to the reverse front path.

    - This state transitions to *Stopped* when the vehicle comes to a stop at a cusp point or the goal pose.

- *Mission Complete*: this state represents the end of the mission. It is reached when the vehicle achieves the goal pose.

In order to test the performance of the simulation, several experiments were designed to test different aspects of the implementation. The experiments that were conducted are introduced in the next chapter. The results of each experiment are presented and discussed in Chapter 5.

CHAPTER IV

EXPERIMENTS

Several experiments were performed on the implemented autonomous vehicle simulation to test its path finding and path following efficiency, accuracy, and robustness. Three environments, shown in Figure 25, were used to test the general performance of the simulation. These environments were used to represent the activities of executing a path through obstacles without collisions, planning a path through a maze, and accurately parking in a parking lot.

Each environment measured 150 meters by 150 meters with a grid resolution of 0.75 meters to give a 200x200 obstacle grid. A resolution of this size is small enough to accurately estimate the shape of obstacles and large enough to provide a generalized Voronoi diagram (GVD) that can be generated efficiently.

Table 1 shows the execution times for each step in the path planning process. GVD generation is the time it takes to load all obstacles into the obstacle grid, generate the generalized Voronoi diagram, and calculate the Voronoi field. The heuristic calculation is the time needed to compute the holonomic-with-obstacles heuristic[1]. The search time is the time taken to discover a path through the environment. The number of nodes expanded for each A* search is also shown. The smoothing time measures how long it took to smooth the path found by Hybrid A*.

[1] The other heuristic, non-holonomic-without-obstacles, is precomputed offline for all environments.

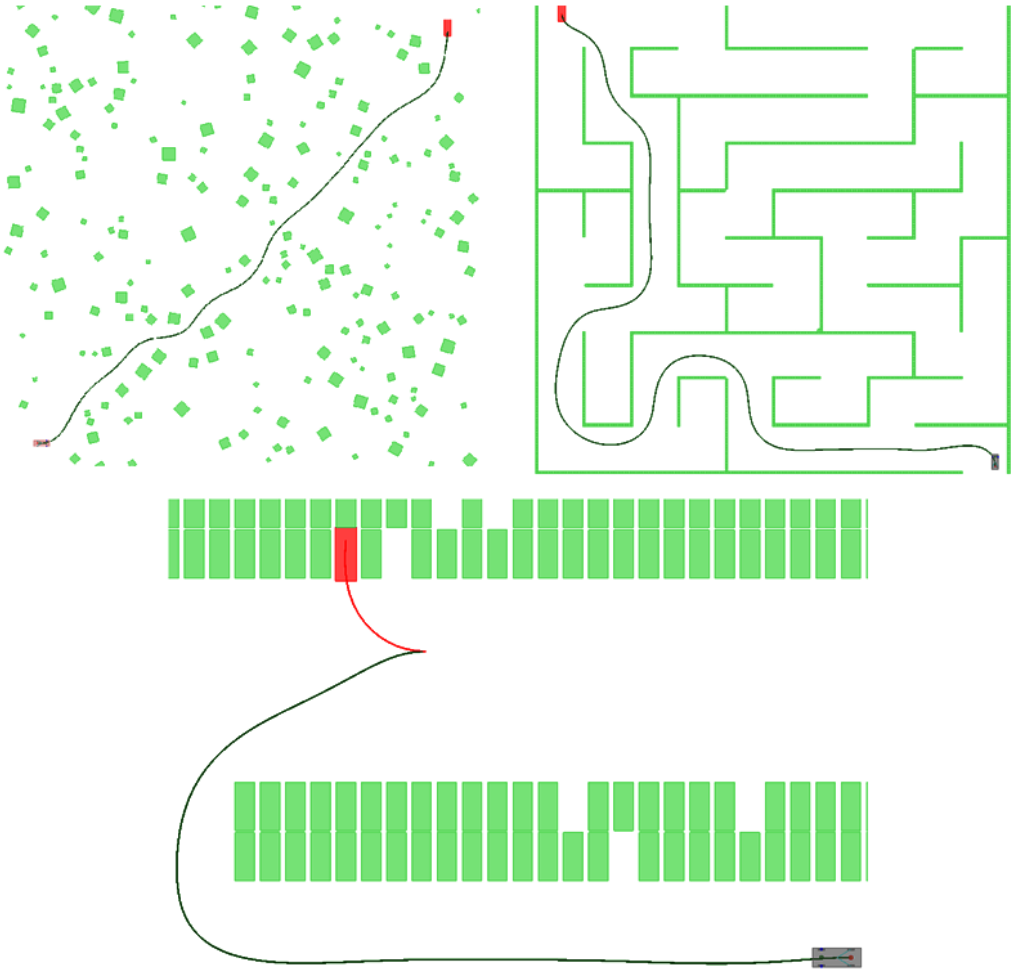


Figure 25. A random environment (top-left), a maze environment (top-right), and a parking environment (bottom). These missions test the simulation's general performance.

Table 1. Path planning times for the three environments

	Maze	**Random**	**Parking**
GVD Generation	682ms	751ms	733ms
Heuristic Calculation	72ms	87ms	92ms
Hybrid A* Search	4630ms (40024 nodes)	430ms (5868 nodes)	914ms (6145 nodes)
Smoothing	626ms	366ms	225ms
Total Path Planning Time	**6010ms**	**1634ms**	**1964ms**

Each environment in this thesis was executed on an Intel Core i5-2500K quad-core processor running at 4.4 GHz. These times represent the approximate order of magnitude for the time it takes to plan a path through a moderately-sized environment. This shows that the presented algorithms can be executed quickly enough to provide only a small delay between when the goal is selected and when the car begins driving toward it.

However, each environment stresses different components of the path planning/path following system. A random environment tests the vehicle's obstacle avoidance, specifically how well the path is smoothed and how accurate the vehicle controller tracks the path. As exhibited in Table 1, the path planner struggles with maze-like environments. This is not surprising, since it is inherently difficult to find paths through a maze. A parking environment gauges the overall performance of the path following components, with a specific interest in testing reverse path tracking, goal accuracy, and the controller's finite state machine.

Each of these three environment types was used as the basis for an experiment that particularly assesses the weakest components in each environment. These experiments are presented in the following sections. The results of each experiment are discussed in the next chapter.

Testing Obstacle Avoidance

A random environment was used to test how well the vehicle avoids obstacles when driving to the goal. The random environment contains 200 square obstacles, each with a random position, size, and rotation. An example random environment is shown in Figure 26.

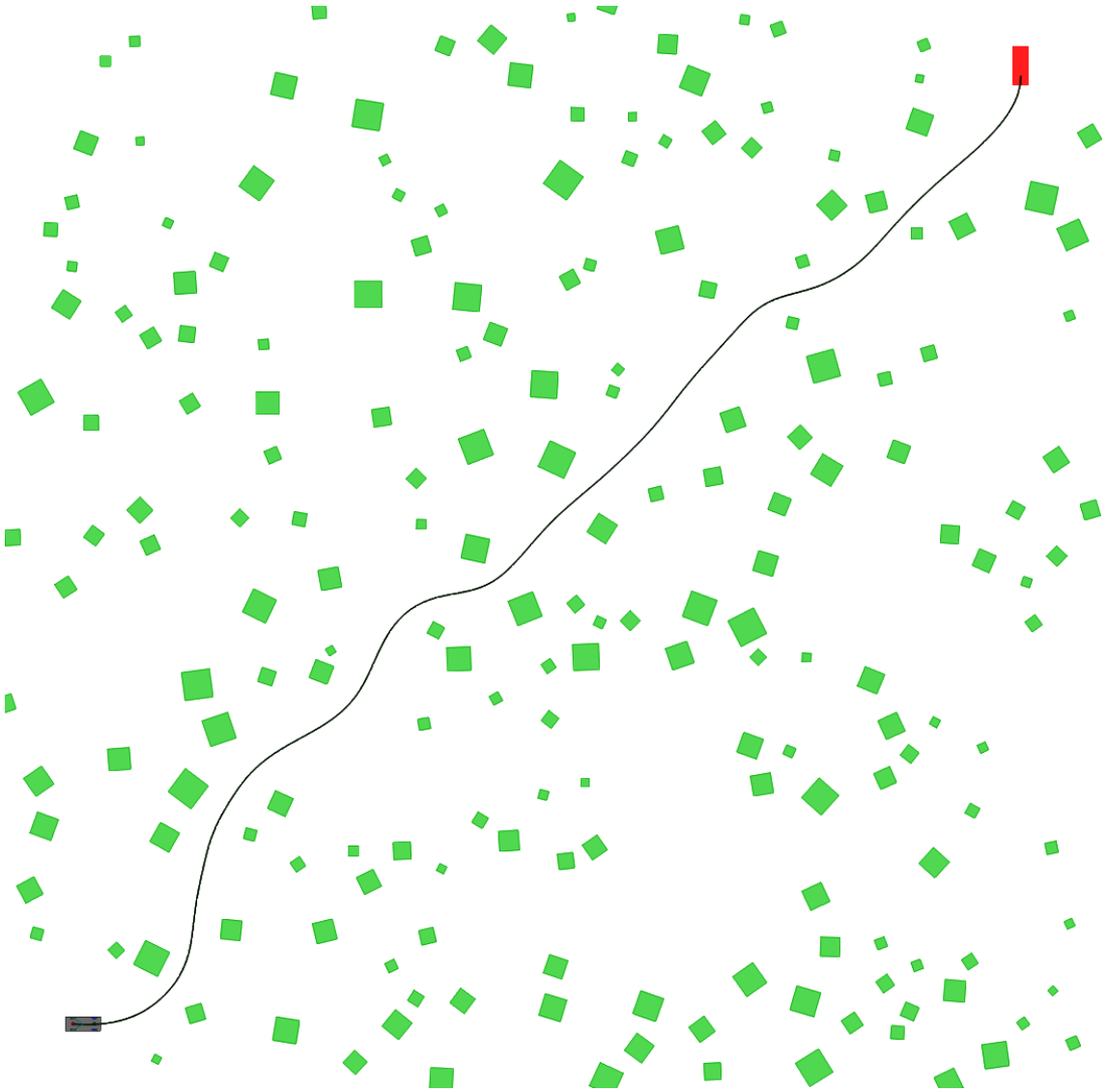


Figure 26. A random environment. This type of environment places many obstacles between the start and goal poses; they test the vehicle's obstacle avoidance performance.

In order to successfully navigate an obstacle-dense environment, the vehicle must plan a safe, drivable path, and the controller must track that path as closely as possible. The comfort of the vehicle's passengers is also important; the vehicle's path must be smooth and free of unnecessary swerving. This is the purpose of smoothing the path discovered by Hybrid A*.

The performance of the path planner's obstacle avoidance was tested by executing the path following routine on smoothed and unsmoothed paths. The vehicle tracked 100 smoothed and 100 unsmoothed paths, where each path was found on a different randomly generated environment. The start and goal poses of each environment were the same; only the positions, sizes, and orientations of the 200 obstacles were random. For each mission, the following statistics were tracked: travel time, average speed, average cross track error, average lateral acceleration, and average change in lateral acceleration. The results of this experiment are discussed in the next chapter.

Testing Path Planning

As mentioned previously, it is particularly difficult to efficiently find paths through a maze-like environment. However, this is not due to inadequacies with the path planning algorithm; it's a simple fact that path planners need more computation time to find paths in complex environments. With this in mind, certain features can improve the performance of a path planner. In the case of Hybrid A*, the heuristics greatly affect the computational efficiency of the path planner. This experiment shows how each heuristic – the non-holonomic-without-obstacles heuristic and the holonomic-with-obstacles heuristic – influences the search algorithm.

The non-holonomic-without-obstacles heuristic considers the length of the optimal path to the goal in an obstacle-free environment. This heuristic pushes the search away from paths that approach the goal from the wrong direction. Consider a maze with two branching paths close to the goal, where each branch leads to the goal, but one branch results in the vehicle facing the wrong direction (Figure 27). Using this heuristic would cause the search to spend more time expanding nodes down the correct branch.

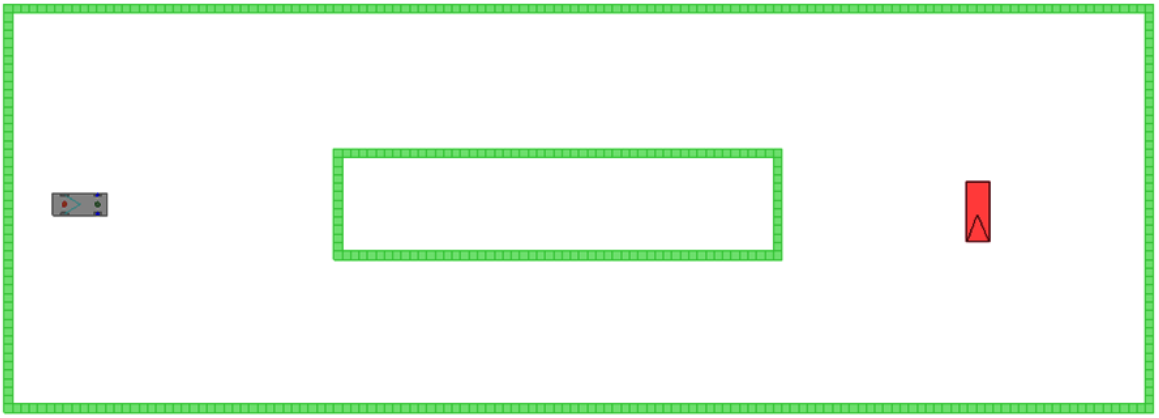


Figure 27. The non-holonomic-without-obstacles heuristic should drive the search down the correct path in this sample environment. The search should prefer the bottom path so that the vehicle has the correct orientation when it reaches the goal position.

The holonomic-with-obstacles heuristic ignores the turning constraints of the vehicle and computes the path lengths to the goal using Djikstra's algorithm. This heuristic is useful in mazes because it causes Hybrid A* to not waste time searching down paths that lead to dead ends (like the environment shown in Figure 28).

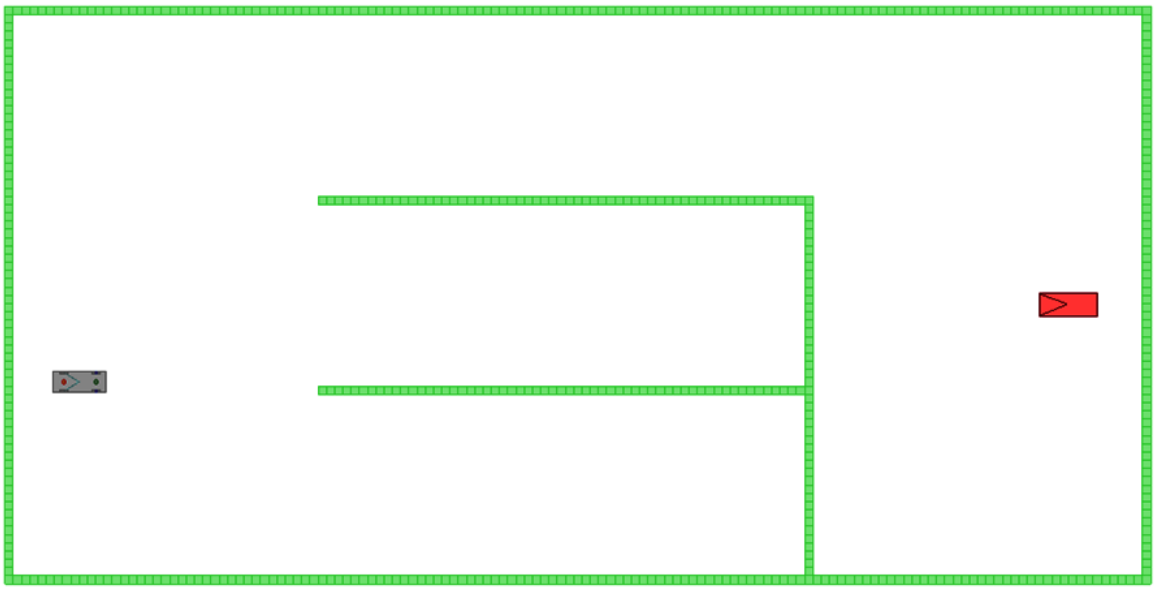


Figure 28. This example utilizes the holonomic-with-obstacles heuristic causing Hybrid A* to only explore viable branches. Paths that lead to dead ends should be ignored.

Testing Goal Accuracy

In order to test the simulation's reverse path tracking and goal accuracy, the following parking environment was developed. The environment consists of two rows of 10 parking spaces on opposite sides, with a row of parked cars in between, as shown in Figure 29. The experiment iterates over this environment 100 times: the car starts in each of the 10 parking spaces on the left and must park in each of the 10 spaces on the right. To assess the performance of reverse path tracking, the orientations of the start and goal poses are set so that the vehicle must back out of the start parking space and back into the goal parking space. The spaces in the left and right rows that are not selected as the start space or goal space are filled with car-shaped obstacles to simulate a full parking lot. The middle row is full with obstacles but has two gaps. When the vehicle reaches the goal pose, the positional and rotational error between the goal pose and the last vehicle pose is recorded. The results for this experiment are discussed in the next chapter.

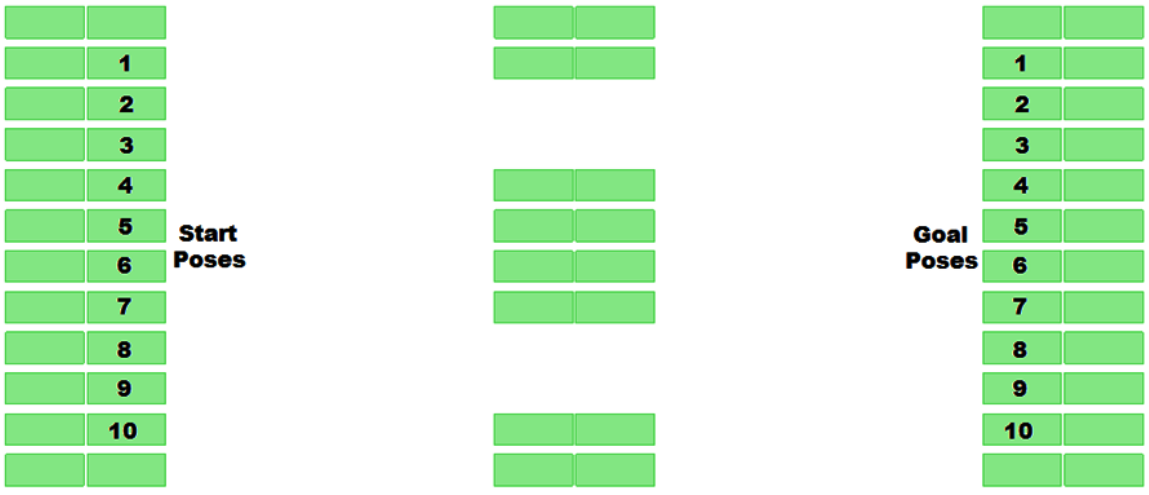


Figure 29. The environment used for the parking experiment. The vehicle starts in one of the 10 spaces on the left and drives to one of the 10 spaces on the right, making 100 runs.

CHAPTER V

RESULTS AND DISCUSSION

This chapter presents and discusses the results from the experiments introduced in the previous chapter. These results demonstrate the performance of the autonomous vehicle simulation with regards to three main environment types: a random environment, a maze-like environment, and a parking environment.

Random Environment

The primary purpose of the random environment experiment was to evaluate the vehicle's smoothing and path tracking operations. In order to successfully navigate an obstacle-dense environment with minimal discomfort to the passengers, the desired path must be properly smoothed and tracked as closely as possible. Table 2 shows the results for the path following implementation on 100 unsmoothed and 100 smoothed paths.

Table 2. Path tracking statistics for smoothed and unsmoothed paths

	Smoothed	**Unsmoothed**
Travel Time	76.6 s	101.4 s
Avg. Speed	6.16 mph	4.65 mph
Avg. Cross Track Error	2.8 cm	1.8 cm
Avg. Lateral Acceleration	0.315 m/s^2	0.316 m/s^2
Avg. Change in Lateral Accel.	0.247 m/s^3	0.613 m/s^3
Collisions	0 out of 100	2 out of 100

This table shows how smoothing a path can affect the performance of the vehicle controller. All values (except for the number of collisions) are averaged across all 100 randomly generated environments to provide a representative set of statistics for

smoothed and unsmoothed paths. There are several important differences between following a smoothed path and an unsmoothed path.

Firstly, when following a smooth path, the vehicle can ease in and out of turns, allowing for a faster speed around the curve. The smooth paths gave a significant increase in average vehicle speed over the unsmoothed paths. Even though the distance from the start to the goal was the same in every randomly generated environment, the vehicle can achieve the goal in less time when the path is smoothed.

Secondly, smoothed paths give a higher average cross track error than unsmoothed paths. This stems mostly from the fact that the vehicle drives faster on smooth paths. At higher speeds, the vehicle controller has less time to respond to changes in the curvature of the path. On unsmoothed paths, the vehicle must drive slowly to correctly track the unnecessary kinks and curves in the path. This slower movement allows for a better cross track error at the cost of a longer trip and excessive swerving. The average cross track error on the smoothed paths was still relatively low, less than 3 cm.

Lastly, smoothing the desired path provides a much smoother ride for the vehicle's passengers. The lateral acceleration referred to in Table 2 is how much acceleration to the left or right a passenger would feel when going around a curve. It is calculated on each frame of the simulation by the following formula, where v is the vehicle's velocity, and r is the radius of curvature determined by the vehicle's wheel angle:

$$a = \frac{v^2}{r}$$

The passengers will feel an increased lateral acceleration if the vehicle travels around a curve at a high speed or if it goes around an especially sharp turn. Although both smooth and unsmoothed paths give the same average lateral acceleration, smooth paths gave a significantly lower *change* in lateral acceleration. The speed with which the vehicle goes around smooth curves balances out the sharpness of the unsmoothed curves to give roughly the same lateral acceleration. However, unsmooth paths require the vehicle to constantly change the steering angle, which in turns causes the passengers to feel frequent changes in the direction of the lateral acceleration. This shows the obvious fact that smoother paths allow for a much more comfortable ride in autonomous vehicles.

Maze-like Environment

Maze-like environments particularly test the utility of the path planner's heuristics. Each heuristic used by Hybrid A* tackles a specific problem presented by maze-like environments: the non-holonomic-without-obstacles heuristic prefers paths that approach the goal with the correct heading, and the holonomic-with-obstacles heuristic pushes the search away from paths that lead to dead ends.

Non-holonomic-without-Obstacles Heuristic

This heuristic is expected to influence Hybrid A* search by preferring paths that approach the goal from the correct direction. The environment introduced in Figure 27 was executed using both the without-obstacles heuristic and a standard Euclidean distance heuristic. The results are shown in Figure 30. This heuristic is very useful in parking lot environments: there may be many paths to a goal position, but only one path ends with the vehicle in the correct orientation.

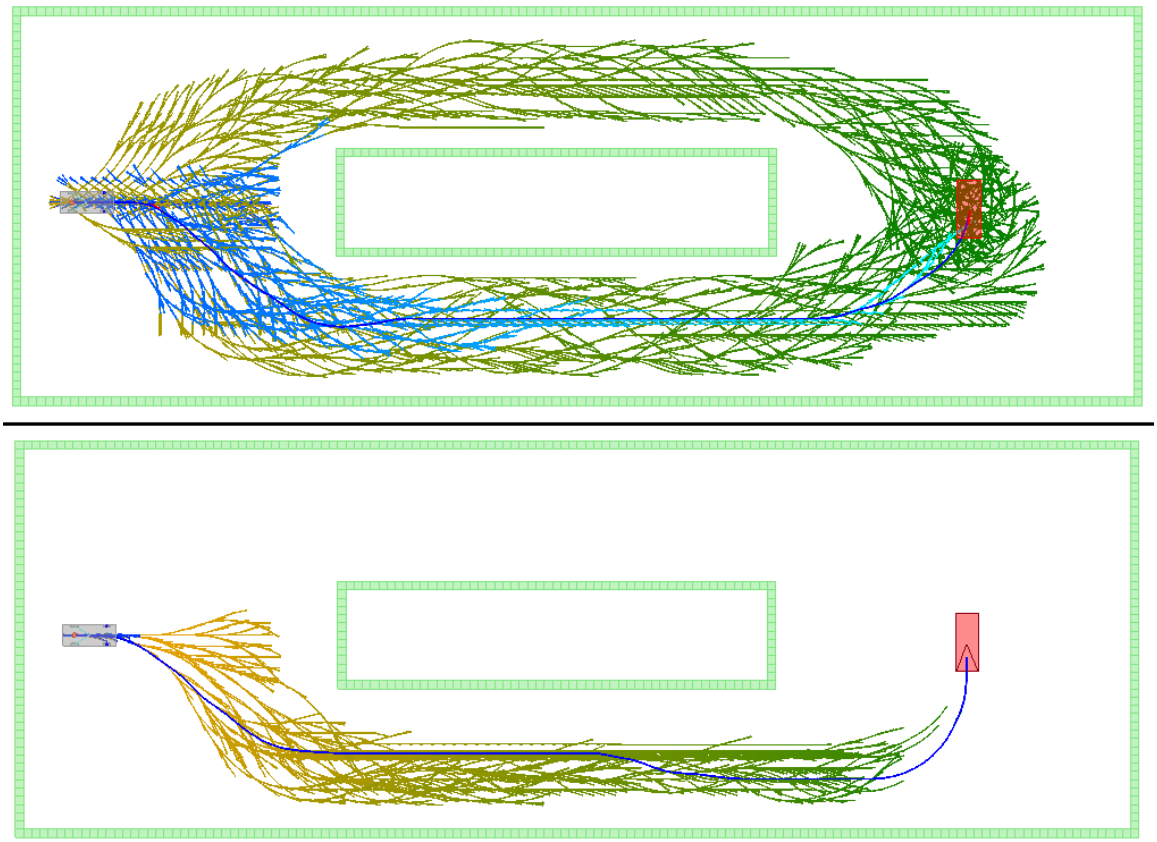


Figure 30. The Euclidean distance heuristic (top) expands 11,673 nodes. The non-holonomic-without-obstacles heuristic (bottom) expands only 3,816; it greatly increases search efficiency in maze-like environments with this type of obstacle by pruning wasteful paths that approach the goal from the wrong direction.

Holonomic-with-Obstacles Heuristic

Many paths in a maze may lead to dead ends. This heuristic avoids such paths by first estimating the path length to the goal from every position in the environment using Djikstra's algorithm on the obstacle grid. A comparison between this heuristic and a standard Euclidean distance heuristic was performed on the environment introduced in Figure 28. The results of this execution are shown in Figure 31. This heuristic provides significant speedups to Hybrid A* search by avoiding paths that lead to dead ends.

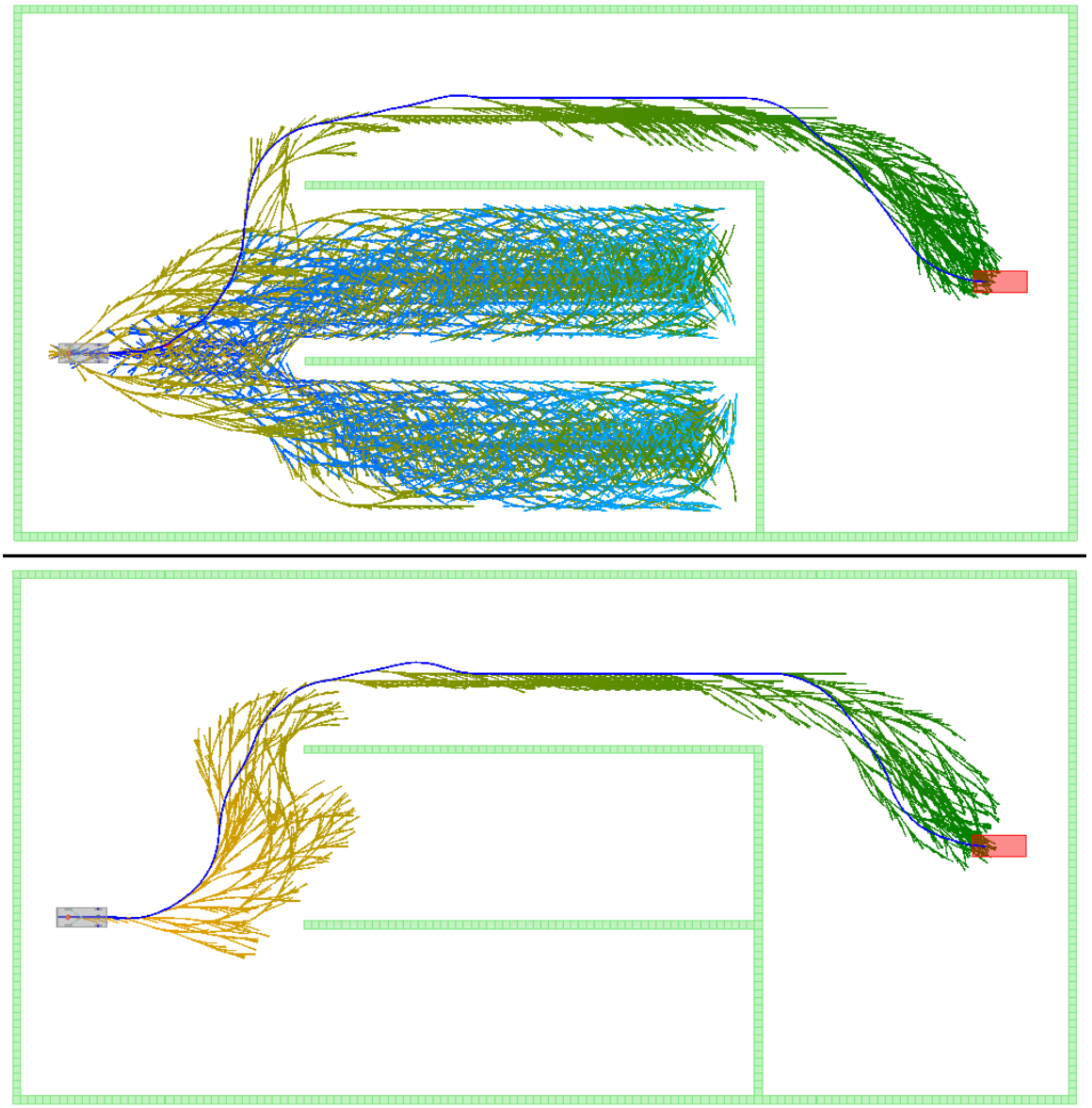


Figure 31. The Euclidean distance heuristic (top) expands 31,810 nodes. The holonomic-with-obstacles heuristic (bottom) does much better: it only expands 4,765 nodes. Because the execution using the Euclidean distance heuristic has no prior knowledge of the structure of the maze, it must exhaustively search through each path until it finds one that does not lead to a dead end. The knowledge gained by pre-solving the environment using Djiksta's algorithm greatly increases search efficiency in these types of environments.

Combing the Heuristics

Because each of the two heuristics is admissible, meaning they never overestimate the path length to the goal, the combination of the two can be used to provide a better performing heuristic function. Whenever the heuristic function is evaluated, the *h*-value for both heuristics is found, and the larger one is used. Now both heuristics can offer their utility to Hybrid A*: the holonomic-with-obstacles heuristic is mostly used when the search is far away from the goal to avoid paths that lead to dead ends, and, as the search nears the goal, the non-holonomic-without-obstacles heuristic pushes the search away from paths that approach from the wrong direction.

Figure 32 shows an environment that combines the types of obstacles presented in the previous two environments. There are five possible paths: three paths lead to dead ends, and two paths are open. However, only one of the two open paths allows the vehicle to arrive at the goal position with the correct orientation. Three heuristics – Euclidean distance, non-holonomic-without-obstacles, and holonomic-with-obstacles – as well as the combined heuristic were used to find paths in this environment. Table 3 shows how well each heuristic performed in Hybrid A* search.

Table 3. Efficiency results for each heuristic

	Nodes Expanded	**Search Time**
Euclidean Distance	200,021	23.5 s
Non-holonomic-without-Obstacles	84,541	9.2 s
Holonomic-with-Obstacles	19,852	2.7 s
Combined	14,181	1.4 s

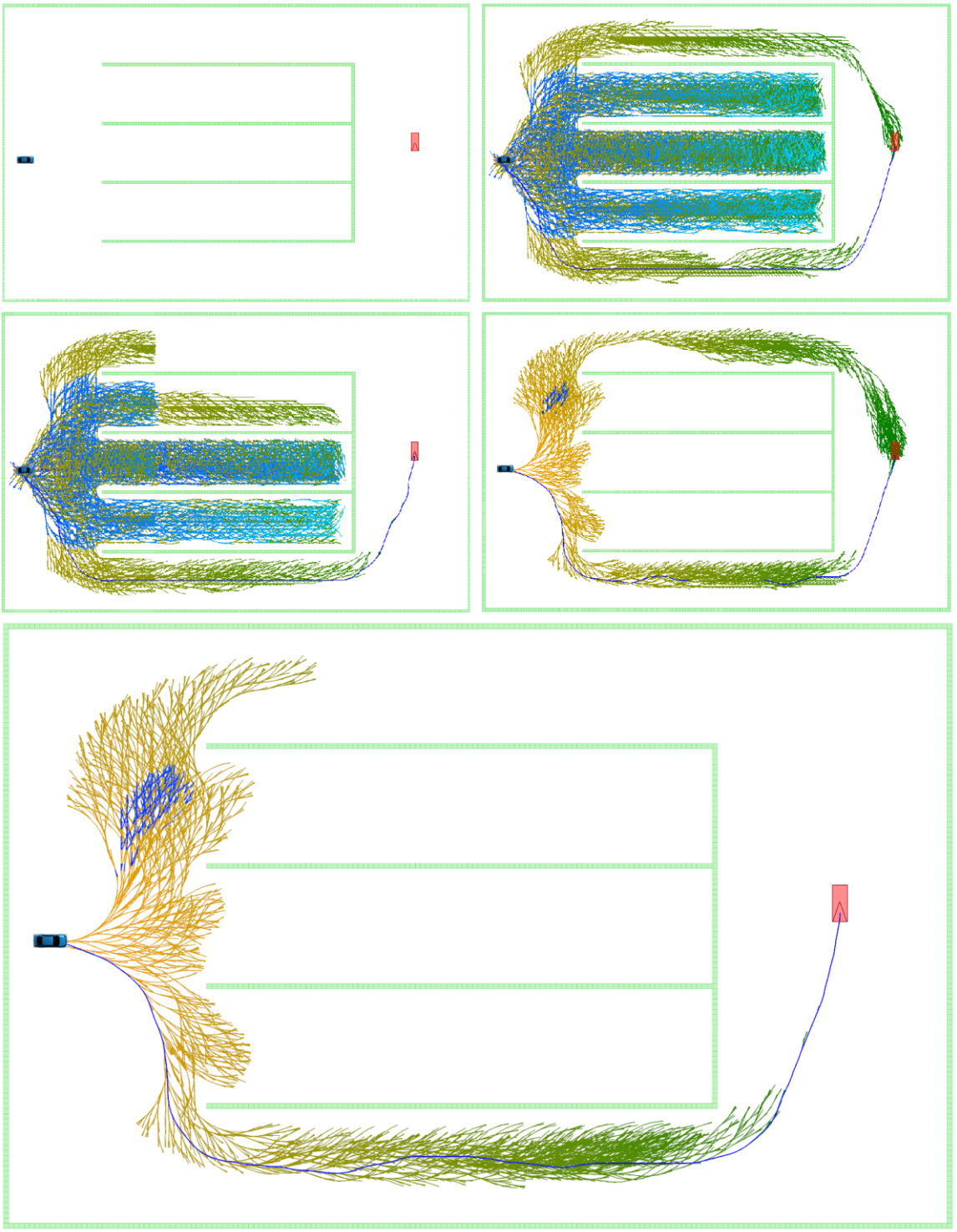


Figure 32. The test environment (top-left) contains three dead ends and two open paths. Only one path arrives with the correct orientation. Three heuristics were used: Euclidean distance (top-right, 200,021 nodes expanded), non-holonomic-without-obstacles (mid-left, 84,541 nodes expanded), and holonomic-with-obstacles (mid-right, 19,852 nodes expanded). The combined heuristic (bottom) is best, expanding only 14,181 nodes.

As shown by the table, the holonomic-with-obstacles heuristic provides more speedup in efficiency than the non-holonomic-without-obstacles heuristic in this particular environment. This is because the holonomic heuristic can prune much of the environment from the Hybrid A* search tree; those paths have already been determined to lead to dead ends by pre-solving the environment using grid-based Djikstra's algorithm. However, both heuristics combined together provide better efficiency than either heuristic alone. The holonomic-with-obstacles and the non-holonomic-without-obstacles heuristics consistently give more than an order of magnitude difference in the number of nodes expanded by Hybrid A* than the standard Euclidean distance metric.

Parking Experiment

In order to perform well in a parking environment, the vehicle controller must be able to track the path accurately, especially in reverse. Since this experiment involves entering a parking space in reverse, the reverse path tracking performance directly affects the goal pose error. Figure 33 shows two paths that were followed during this experiment.

The goal pose error was averaged over all 100 executions. The vehicle's final state was 1.6 cm away from the goal position on average, less than the average cross track error in the random environment experiment. The vehicle had an average orientation error of 0.9 degrees. Across all 100 executions, the maximum goal position error was 4.6 cm, and the maximum goal orientation error was 1.6 degrees. These results are very encouraging considering the difficulty some humans have with accurately parking a vehicle, especially in reverse.

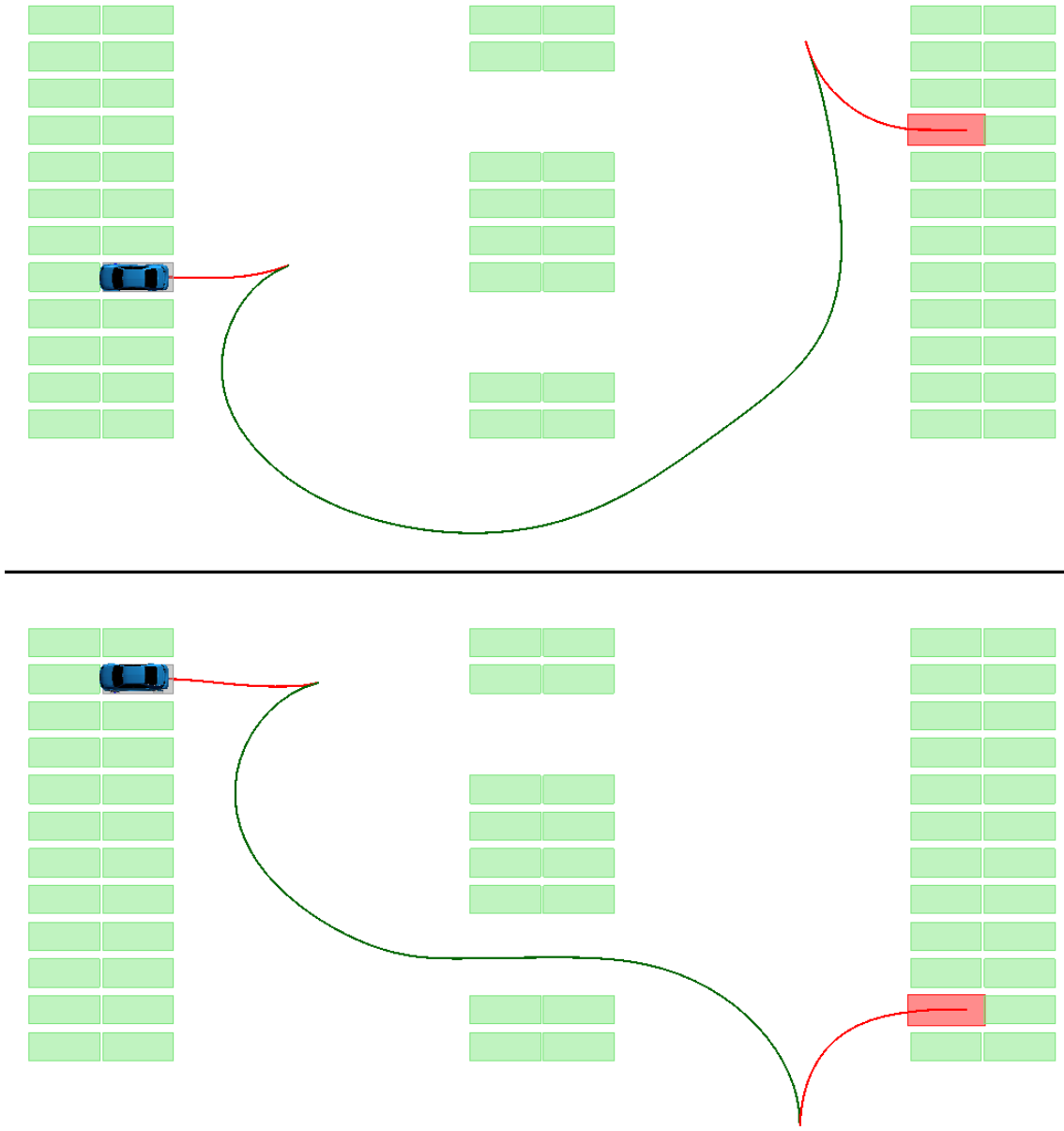


Figure 33. Example paths in the parking experiment: start space 7 to goal space 3 and start space 1 to goal space 10. The red lines are sections of the path driven in reverse.

CHAPTER VI

CONCLUSIONS

This thesis introduced and analyzed algorithms used by an autonomous car for efficient path planning in a continuous world and accurate path following in an obstacle-laden environment. Many of these algorithms are used in current autonomous vehicle research. These algorithms were implemented in a simulation and tested in virtual environments mirroring real-world situations. The simulation and the algorithms it implements performed their purposes efficiently and accurately. The path planner can produce a safe, smooth path through an environment containing hundreds of obstacles on the order of a few seconds. The path follower can accurately track this path and arrive at the goal position with an error of just a few centimeters. The simulation developed for this thesis can serve as a foundation for future work in autonomous cars and will be released to the public as open-source software.

Future Work

This thesis uses algorithms to successfully navigate a vehicle through a small, static environment. However, there are many other types of autonomous driving a driverless car would have to face, such as highway driving and driving in heavy traffic. The simulation could be extended to produce a fully-featured autonomous vehicle simulator. In addition, there are several improvements that could be made to the simulation as it currently stands.

This simulation does not address sharing an environment with other vehicles or dynamic objects. Since the success of real-world autonomous driving in the past few years, there has also been a myriad of research into coordination between autonomous

agents. Autonomous coordination could be employed in the simulation to allow multiple vehicles to drive along separate paths in the same environment. Each vehicle would have to achieve its goal pose without colliding with the other vehicles in the situation. The vehicles would not only avoid each other but coordinate their path planners so that each vehicle will reach its goal as quickly as possible without blocking others from their goals.

This thesis presents algorithms useful for only one type of path planning: free-space planning, where there is no defined structure to the environment. This type of path planning would be used mostly for navigating a parking lot or negotiating around an obstacle in the road. However, there are separate planners used to find paths in structured environments like traffic lanes and intersections. These planners usually feature dynamic programming and state machines to determine the correct course and obey traffic rules. They are usually coupled with some sort of obstacle detection and avoidance in order to drive on a road safely with other cars. A road path planner could be implemented to navigate the vehicle along streets and through intersections until it reaches a parking lot, where the planner utilized by the simulation would take over to successfully park the car in a parking space.

The vehicle model used by the simulation to plan and follow paths is a relatively simple one. A full dynamic representation of a vehicle could be added to the simulation to more precisely model the steering and motion of a car. This model could be used to more accurately follow a planned path by determining the exact steering, gas, and brake controls at each step of the simulation needed to reduce the cross track error. This proactive approach would be an improvement over the current reactive approach of responding to errors in path tracking after they happen.

Looking Forward

The future of autonomous cars is an uncertain one. There are already autonomous cars on the road now; however, there are still major technical, ethical, and legal challenges that need to be tackled before the vision of the ubiquitous driverless car becomes a reality. The replacement of the human with the computer in handling one of the most dangerous activities mankind faces would be a great boon to society. Traffic accidents cause more than 30,000 deaths a year in the United States [9]. A computer-controlled car can react faster and more accurately than a human; it would not be susceptible to emotion, incompetence, or inexperience.

There are some necessary legal obstacles that need to be considered before autonomous vehicles can take the road. However, at the time of this writing, two states, Nevada and California, have already passed legislation permitting computer-controlled vehicles on the road, and many other states are following suit. Automotive companies like Audi, Volvo, General Motors, and BMW have already developed autonomous driving technology. The framework is in place, sans some technical challenges such as autonomous coordination. They are only waiting for the legislative world to catch up.

This thesis is only a small bit of research in the giant world of autonomous cars. This author hopes that the explanation of algorithms and implemented simulation add to the treasury of autonomous driving, if only to give the readers of this thesis some credibility to the idea that driverless cars could be a part of the near future. The forward march of technology is hardly a predictable one, yet this author trusts you will see him *not* driving his brand new auto-car on the highway one day.

REFERENCES

[1] "2004 Grand Challenge." *Defense Advanced Research Projects Agency*. DARPA, 13 Mar. 2004. Web. 19 Oct. 2012. <http://archive.darpa.mil/grandchallenge04/>.

[2] "2005 Grand Challenge." *Defense Advanced Research Projects Agency*. DARPA, 31 Dec. 2007. Web. 19 Oct. 2012. <http://archive.darpa.mil/grandchallenge05/>.

[3] Barton, Matthew J. *Controller development and implementation for path planning and following in an autonomous urban vehicle*. Diss. The University of Sydney, 2001.

[4] Bresenham, Jack E. "Algorithm for computer control of a digital plotter." *IBM Systems journal* 4.1 (1965): 25-30.

[5] Dijkstra, Edsger W. "A note on two problems in connexion with graphs." *Numerische mathematik* 1.1 (1959): 269-271.

[6] Dolgov, Dmitri, et al. "Path planning for autonomous vehicles in unknown semi-structured environments." *The International Journal of Robotics Research* 29.5 (2010): 485-501.

[7] Dolgov, Dmitri, et al. "Practical Search Techniques in Path Planning for Autonomous Driving." *Proceedings of the First International Symposium on Search Techniques in Artificial Intelligence and Robotics (STAIR-08)* (2008): n. pag.

[8] "Farseer Physics Engine." CodePlex, 9 Apr. 2011. Web. 19 Oct. 2012. <http://farseerphysics.codeplex.com/>.

[9] "Fatality Analysis Reporting System (FARS) Encyclopedia." *National Highway Traffic Safety Administration*. NHTSA, 2010. Web. 19 Oct. 2012. <http://www-fars.nhtsa.dot.gov/>.

[10] Hart, Peter E., Nils J. Nilsson, and Bertram Raphael. "Correction to a formal basis for the heuristic determination of minimum cost paths." *ACM SIGART Bulletin* 37 (1972): 28-29.

[11] Kalra, Nidhi, Dave Ferguson, and Anthony Stentz. "Incremental reconstruction of generalized Voronoi diagrams on grids." *Robotics and Autonomous Systems* 57.2 (2009): 123-128.

[12] Kokholm, Niels, and Peter Sestoft. "The C5 Generic Collection Library for C# and CLI." N.p., 27 May 2011. Web. 19 Oct. 2012. <http://www.itu.dk/research/c5/>.

[13] Lau, Boris, Christoph Sprunk, and Wolfram Burgard. "Improved updating of Euclidean distance maps and Voronoi diagrams." *IEEE Intl. Conf. on Intelligent Robots and Systems (IROS), Taipei, Taiwan*. 2010.

[14] Markoff, John. "SMARTER THAN YOU THINK; Look Officer, No Hands: Google Car Drives Itself." *The New York Times*. The New York Times, 10 Oct. 2010. Web. 19 Oct. 2012. <http://www.nytimes.com/2010/10/10/science/10google.html>.

[15] Montemerlo, Michael, et al. "Junior: The stanford entry in the urban challenge." *Journal of Field Robotics* 25.9 (2008): 569-597.

[16] Reeds, J. A., and L. A. Shepp. "Optimal paths for a car that goes both forwards and backwards." *Pacific Journal of Mathematics* 145.2 (1990): 367-393.

[17] Snider, Jarrod M. "Automatic steering methods for autonomous automobile path tracking." *Robotics Institute, Carnegie Mellon University, technical CMU-RI-TR-09-08* (2009).

[18] Thrun, Sebastian, et al. "Stanley: The robot that won the DARPA Grand Challenge." *The 2005 DARPA Grand Challenge* (2007): 1-43.

[19] "Urban Challenge." *Defense Advanced Research Projects Agency*. DARPA, 3 Nov. 2007. Web. 19 Oct. 2012. <http://archive.darpa.mil/grandchallenge/>.

[20] Urmson, Chris, et al. "Autonomous driving in urban environments: Boss and the urban challenge." *Journal of Field Robotics* 25.8 (2008): 425-466.

[21] "XNA Game Console." CodePlex, 24 Aug. 2009. Web. 19 Oct. 2012. <http://console.codeplex.com/>.

[22] "XNA Game Studio 4.0." *MSDN*. Microsoft, 6 Oct. 2011. Web. 19 Oct. 2012. <http://msdn.microsoft.com/en-us/library/bb200104(v=xnagamestudio.40).aspx>.

APPENDIX

APPENDIX A

FORMAT OF A MISSION FILE

This appendix defines the format of the JSON mission file. The *root* object of the mission file can contain the following properties:

```
{
    "start": (pose),                  the start pose
    "goal": (pose),                   the goal pose
    "environment": (environment)      the mission's environment
}
```

Each `(type)` symbol needs to be replaced by a specifically formatted array or object. The various "types" are defined as follows.

The `(value)` type represents one numerical value. The value can be defined as a single integer or floating point number, or it can be defined as a range. The range has two numerical values which define the upper and lower bounds of a randomly chosen number. If a range array is parsed, the value is randomly chosen between the lower and upper bound at the time the mission is loaded.

```
(value) → (number)                    constant number
(value) → [(number), (number)]        random number in this range
```

The `(pose)` type represents a three-dimensional vehicle pose. This is defined as an array with three elements: $[x, y, \vartheta]$, where ϑ is the orientation in radians.

```
(pose) → [(value), (value), (value)]
```

The `(environment)` type contains various environment parameters such as size and resolution, as well as the environment's obstacles. In lieu of obstacles, the environment can instead provide a bitmap image to be parsed. Each black pixel in the bitmap describes the position of a square obstacle in the environment. If the mission file does not define the size, origin, and resolution of the environment, the following default values are used:

`height: 150,`	the height in meters
`width: 150,`	the width in meters
`origin: [0, 0],`	the bottom-left corner of the environment
`resolution: 0.75`	the size of each cell in meters

At this size and resolution, the environment will be 200 cells by 200 cells. The `(environment)` type is defined as follows:

```
(environment) → [(obstacle), (obstacle), ...]
(environment) → {
      "height": (number),
      "width": (number),
      "origin": [(number), (number)],
      "resolution": (number),
      "bitmap": (string),
      "obstacles": [(obstacle), (obstacle), ...]
}
```

The `(obstacle)` type represents a rectangular obstacle. The obstacle is described by the following properties: height, width, *x*-position, *y*-position, and rotation. The (*x*, *y*) position defines the position of the center of the obstacle. The `(obstacle)` type can be

defined by a two, three, four, or five element array, where each element in the array is a `(value)` type. If the size is not defined by the obstacle, then the obstacle will be a 2 meter by 2 meter square. If the rotation is not defined, then the obstacle will have no rotation. The `(obstacle)` type can have any of the following definitions:

```
(obstacle) → [<x>, <y>]
(obstacle) → [<x>, <y>, <rotation>]
(obstacle) → [<x>, <y>, <size>, <rotation>]
(obstacle) → [<x>, <y>, <width>, <height>, <rotation>]
```

The following is an example mission file. It defines the start pose, a random goal pose, and four 5 meter by 5 meter obstacles with random positions and orientations.

```
{
    "start": [10, 10, 0]
    "goal": [[50, 100], [50, 100], [0, 3.14159]]
    "environment": [
        [[0, 150], [0, 150], 5, 5, [0, 3.14159]],
        [[0, 150], [0, 150], 5, 5, [0, 3.14159]],
        [[0, 150], [0, 150], 5, 5, [0, 3.14159]],
        [[0, 150], [0, 150], 5, 5, [0, 3.14159]]
    ]
}
```