Safe Motion Planning for Autonomous Driving

by

Micah Wylde Class of 2012

A thesis submitted to the faculty of Wesleyan University in partial fulfillment of the requirements for the Degree of Bachelor of Arts with Departmental Honors in Computer Science

Acknowledgements

I would like to thank my advisor, Professor Eric Aaron, for introducing me to the world of AI research and giving me exceptional freedom and opportunity to explore as a member of his lab. He also deserves thanks for introducing me to my topic through an assignment in his Artificial Intelligence class. This thesis would not have been possible without his guidance and support.

I would also like to thank my friends, who have supported me throughout these past four years. My housemates, Jen, Miles, Evan, and Ann, in particular deserve thanks for handling with aplomb the craziness that theses can invoke.

I also must thank my parents, who have always pushed me to achieve everything I can, and who are more than anybody else responsible for me being here today.

Finally, I would like to thank Jeffrey Ruberg, who collaborated with me on a class project on autonomous driving which inspired some of my thoughts in this thesis and drove my interest in the subject.

Abstract

Self-driving cars have the potential to revolutionize transportation by making it cheaper, safer, and more efficient. In this thesis we describe a novel motion planning system, which translates high-level navigation goals into low-level actions for controlling a vehicle. Specifically, the motion planning system is responsible for choosing at each time step an appropriate velocity and steering angle, which can then be implemented by the driving hardware or simulator. Our planner is able to compute a safe and efficient trajectory in a dynamic environment while staying within its lane and avoiding obstacles.

The planner works as follows: given a road map represented as a graph, an incremental heuristic search method is used to generate an optimal path. A section of this path nearest to the agent is smoothed using spline techniques to generate an arclength parameterized reference path for the agent to follow. A two-level hierarchical state space is produced that encompasses the various possible choices of steering angle. Using a vehicle motion model, the future positions of the car are determined for each action. Those actions that lead to invalid states (those that bring the agent in contact with an obstacle or out of its lane) are ruled out, and the action cost for each state is computed according to its path-following behavior. Finally, the action that leads to the lowest cost state is chosen and executed.

The planner was implemented in simulation and tested in various driving scenarios with results comparable or better to those produced by other motion planning systems in terms of speed, safety, path-following behavior and comfort.

Contents

Chapte	er 1. Introduction	1
1.1.	Overview of the paper	3
Chapte	er 2. Related work	4
2.1.	Perception	4
2.2.	Global planning	5
2.3.	Motion planning	6
Chapte	er 3. Planning	8
3.1.	Motion model	8
3.2.	State expansion	10
3.3.	Ideal path generation	12
Chapter 4. Action selection		18
4.1.	Staying in lane	18
4.2.	Pedestrian avoidance	20
4.3.	Path cost	22
4.4.	Action selection	23
Chapte	er 5. Methods and demonstrations	27
5.1.	Simulation environment	27
5.2.	Methods	28
5.3.	Metrics	29
5.4.	Demonstrations and results	32

5.5.	Analysis	40
Chapter	6. Discussion and future work	43
6.1.	Future work	44
6.2.	Conclusion	45
Bibliogra	aphy	47
Appendi	x A. Ancillary algorithms	49
Appendi	x B. Tables and values	50

CHAPTER 1

Introduction

Self-driving cars hold much promise for the future of transportation. Robotic vehicles can be safer, more convenient, and fuel-efficient than their human-driven counterparts. AI researchers have been interested in the potential for vehicular navigation since the 1960s. In order to spur development in this area, the Defense Advanced Research Projects Agency (DARPA) organized the Grand Challenge in 2004, which tasked contestants to build cars to autonomously navigate a 142 mile-long course in the Mojave desert [24]. Though none of the vehicles were able to finish the course, another competition held the next year was more successful. Four teams finished in the allotted time, traversing a treacherous desert course with no human guidance.

Building on this achievement, in 2007 DARPA organized the Urban Challenge, which took place in a simulated urban environment [4]. To win, cars had to navigate a maze of streets while accounting for other traffic and following California traffic laws at all times. Six teams finished the 61 mile course with the winner, "Boss" from Carnegie Mellon University, taking a little more than four hours [27]. That so many failed even in a contrived urban scenario (lacking unpredictable obstacles like pedestrians and allowing unrealistically slow speeds) shows the immense difficulty in creating a workable vehicle.

A successful autonomous vehicle should satisfy two main requirements: it must at all times be safe and legal according to the rules of the road and it should reach its destination efficiently in terms of time and fuel. This involves choosing good routes from map data, integrating that information with observations from onboard sensors, and directing the mechanical controls of the car (primarily steering and acceleration/braking) to follow the chosen route while avoiding obstacles. The first is within the realm of graph algorithms and can be solved using an incremental search heuristic method which can dynamically adjust paths according to changing conditions like traffic or obstacles. The second is approached from the field of computer vision and involves tracking and classifying obstacles and other roadway information like signs and traffic lights. This thesis focuses on the third challenge, which can be called motion planning.

In particular, we discuss the following problem: given a path through the road map and a set of classified obstacle trajectory observations, what is the best choice of steering angle and acceleration at each time step? We assume that our vehicle travels according to the Ackermann model of vehicular motion [21], which describes the arc on which the car will travel given a particular setting of its control variables, steering angle ϕ and acceleration a. For each action (ϕ, a) we can derive the arc that the car will follow if the action is chosen from the current time t_0 to a time t_s . Then, we can compare each action to an ideal, smooth path and choose the closest while ruling out those actions which might produce a collision.

Motion planning is performed over a three-level hierarchical state space where states are 5-tuples $(t, \mathbf{p}, \theta, \phi, v)$, which are respectively time, vehicle position, heading angle, steering angle and velocity. From each state, the agent can choose between actions (ϕ, a) , where the allowed range of ϕ is restricted to values close to the current ϕ , to prevent abrupt changes in lateral velocity. ϕ and a are discretized to make evaluation of the state space tractable.

Using a vehicular motion model, we can simulate the trajectory that would be

followed by the agent should it choose a particular path through the state space. The trajectory is checked as to whether it may collide with obstacles and whether it remains in lane. From those actions remaining the one that best follows the reference path is chosen.

1.1. Overview of the paper

We continue in Chapter 2 by giving an overview of the research in autonomous agent navigation and related areas. Chapters 3 and 4 together describe the mechanics of the motion planning system. The former outlines the planning mechanisms for building the state space and generating the reference path, while the latter describes how the state space is pruned and evaluated to produce an action. Chapter 5 describes our simulation environment, the demonstrations done to evaluate the effectiveness of our planner and analyzes the results. Finally, Chapter 6 gives a concluding discussion of our work and areas for future research.

CHAPTER 2

Related work

A complete autonomous vehicular system is comprised of many pieces. These are often broken up into three subsystems, which might be called perception, mission planning and motion planning. The perception system is responsible for analyzing the data from various sensors (such as LIDAR, RADAR, video, and GPS) to infer information about the world like road boundaries, trajectory and classifications of obstacles and other roadway information. The global planner is responsible for high-level deliberative planning: given a goal position it should produce an efficient path through the road graph from the car's current position to its goal position. Finally, the motion planner takes as input the global path and perception information and chooses settings for each of the car's control variables (steering, acceleration, braking, gear shifts, etc.) such that it follows the path while safely avoiding obstacles and following traffic laws. In the rest of this chapter we will describe some of the prior work in these three areas.

2.1. Perception

Perception for autonomous vehicles encompasses various problems in computer vision and related fields. The desired result is an accurate three-dimensional model of the world surrounding the vehicle as well as localization of the vehicle within that model. We focus on two specific areas relevant to this thesis: determining the boundaries of the road and lane and tracking and classifying obstacles.

2.1.1. Lane detection. Staying within a lane and on the road is crucial for legal driving. While map data may give the general path a road follows, determining the precise boundaries requires local perception. This is usually accomplished using either LIDAR or video sensors. The system presented in [27] uses sidemounted LIDAR sensors to determine the road shape by detecting sharp changes in geometry. Under the assumption that roads are generally flat, these should indicate road edges. To detect lanes, the laser data is searched for evidence of painted lane markers. Lane boundaries are assumed to be brighter than the surrounding road, and register as higher intensity by the LIDAR. The result of these processes is a set of points constituting the acceptable boundaries for the vehicle which is used by the motion planning system.

2.1.2. Obstacle classification and tracking. To avoid collisions it is imperative that the motion planning system have an accurate list of the obstacles present. It is also helpful to track obstacles between observations and classify them (e.g., as pedestrians, cars, bicycles, animals, etc.) so that their future behavior can be predicted. And while detection of obstacles in the roadway is relatively simple with LIDAR data (e.g., in [12]) tracking and classification are more challenging. In [6], a standard pedestrian classifier ([5]) is combined with a hypothesise-and-verify tracker to produce a list of likely pedestrian trajectories from stereo video data. The tracker described in [7] integrates radar and laser data to generate and a list of dynamic obstacle trajectory hypotheses, each of which are checked for conformance to a simple vehicular motion model.

2.2. Global planning

Global planning algorithms generally solve a graph search problem: given two nodes in a weighted, directed graph, find the least costly path from one to the

other. By constructing a graph from the road map with edge costs determined by distance and other factors, like traffic and speed limits, we would like to find the fastest route from the current location of the agent to its destination. The A* algorithm [13] has been widely used in AI for this purpose as it is very fast and accurate. However, it can still be too slow to run in real time on very large graphs. In response, several "incremental" alternatives have been proposed. These allow fast replanning by storing and reusing information from previous searches. The Dynamic A* (or D*) algorithm [26] and its more widely used successor, D*-lite [16], initially assume that all paths are traversable. When previously unknown obstacles are detected—a road that has been closed, a tree branch in the road, etc.—the algorithm is able to quickly route around it without recomputing the entire plan. Anytime Dynamic A* (AD*) [20] improves the real-time applicability of these algorithms by relaxing their guarantees of optimality. As an "anytime" algorithm, it can return a solution at any point in its execution by starting with a sub-optimal one and gradually improving it in as much time as is available.

2.3. Motion planning

Motion planning for autonomous vehicles is a diverse field. Methods can be broken up into two basic approaches: reactive and deliberative. Reactive systems typically recompute the plan at each time step; examples include potential-fields and dynamical based approaches in which obstacles exert repulsive forces and targets exert attractive forces. Such methods are very good at avoiding collision with dynamic obstacles, but have difficulty performing complex, longer-term actions like three-point turns and can get stuck in local minima. Deliberative methods by contrast have two stages: planning and execution. During the former a fully-specified motion plan is computed, typically an expensive process. In the

latter the motion plan is simply run. Deliberative systems can often find very good solutions in static environments but must re-plan when the environment changes.

As a result of these shortcomings much research has been done on hybrid systems which combine deliberative global planning with reactive local planning. The general approach is to generate a graph representation of the environment and perform a search on it to compute a path from current location to destination. Then, at each time step, a trajectory is chosen which most closely approximates this path while avoiding obstacles.

In [8], AD* is used to find a path through a state space constructed from the action set (x, y, θ, v) , where x, y are the position, θ is pose and v is velocity. This plan is continually updated as new local information is detected by the sensors. In each timestep, a vehicular motion model generates various possible trajectories which are compared against the precomputed path, and the best chosen and executed. This idea is extended in [19] by using a multi-resolution state space, which uses fine discretization near the agent and coarse discretization further away, limiting the size of the graph. The states are 4-tuples (x, y, θ, v) that are reachable given some feasible action as determined by their vehicular model. However this process is sufficiently expensive that the state space must be constructed offline. In contrast, [21] plans globally over a state space consisting only of position (x, y) to produce an ideal path. Local planning is done to follow this path as closely as possible while trajectories that collide with obstacles are eliminated.

CHAPTER 3

Planning

In this chapter we describe the planning that is done to prepare for action selection, described in Chapter 4. We begin in Section 3.1 by outlining a motion model that predicts future states of the car given certain input variables. Using this model we give in Section 3.2 the state space that describes the state changes resulting from taking actions. We conclude in Section 3.3 by describing how we generate a smooth reference path for the agent to follow.

3.1. Motion model

We simulate car dynamics by using an Ackermann model of vehicular motion. An Ackermann vehicle steers with its two front wheels, which do not turn at the same rate. Rather than both wheels forming the same angle with the car's axis, they trace out two circles with the same center point \mathbf{c} and different radii. Most consumer cars utilize Ackermann steering as it helps prevent tire slippage during turns.

Taking this into account, given a state $s_0 = (0, \mathbf{p}, \theta, \phi, v)$ and an action (ϕ_a, a) we can compute $s_t = (t, \mathbf{p}_t, \theta_t, \phi_t, v_t)$, the state reached by traveling for t seconds with steering angle ϕ_a and acceleration a. Figure 3.1 provides a geometric representation of the following derivation. The approach we take is to compute the position \mathbf{q} of the inner back wheel (right for a right turn, left for a left turn) and r, the distance between that wheel and \mathbf{c} and with that find \mathbf{c} . For turning right, we find the position of the back-right wheel by shifting \mathbf{p} halfway along the tangent

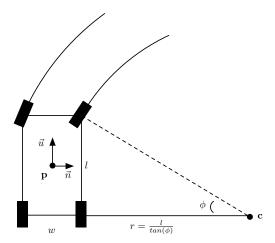


FIGURE 3.1. This diagram demonstrates Ackermann steering. The car is at point \mathbf{p} with $\theta=0$ and is executing a right turn with steering angle ϕ . The vectors \vec{u} and \vec{n} are its tangent and normal respectively. l and w are the length and width or the car.

and normal vectors:

$$\mathbf{q} = \mathbf{p} + \vec{n} \frac{w}{2} - \vec{u} \frac{l}{2}.$$

Then we can find r using simple trigonometry:

$$r = \frac{l}{tan(\phi)}$$

and \mathbf{c} by shifting \mathbf{q} by r along the normal vector:

$$\mathbf{c} = \mathbf{q} + \vec{n}r.$$

Next is δ , which is the number of radians around the circle the car travels in time t

$$\delta = v \frac{t}{r},$$

and we can find the position of the car \mathbf{p}_t by rotating the vector \vec{w} from \mathbf{p} to \mathbf{c} by δ radians:

$$\begin{aligned} \vec{w} &= \mathbf{p} - \mathbf{c} \\ d &= atan2(\vec{w_y}, \vec{w_x}) \\ \mathbf{p}_t &= \mathbf{c} + \langle |w| \cos(d - \delta), |w| \sin(d - \delta) \rangle. \end{aligned}$$

The function atan2(y, x) gives the angle between the positive x-axis and the vector from the origin to (x, y). It can be defined in terms of arctan as

$$atan2(y, x) = 2arctan\left(\frac{y}{\sqrt{x^2 + y^2} + x}\right).$$

We find θ_t simply as

$$\theta_t = \theta - \delta$$

Turning left (i.e., when $\phi < 0$) is accomplished similarly. Traveling straight ($\phi \approx 0$) must be handled separate to avoid dividing by zero. In that case we need merely project the car along its current vector of movement u by tv meters:

$$\mathbf{p}_t = \mathbf{p} + tv\vec{u}$$

and as there is no turning, $\theta_t = \theta$.

3.2. State expansion

We use a novel three-level hierarchical state space (as shown in Figure 3.2) where each state is a 5-tuple $(t, \mathbf{p}, \theta, \phi, v)$ respectively time, position, heading angle, steering angle, and velocity. It is expanded as follows: from the stating state (the current position, pose, turning angle and velocity of the agent) we generate every possible action and, for each action, compute the state resulting from taking

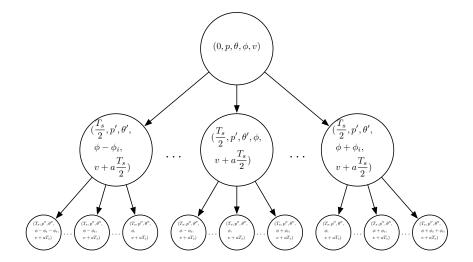


FIGURE 3.2. A representation of the state space as a three level tree. The root node is the current state, with time t=0. The second level covers half of the time horizon T_s and includes values of ϕ ranging from $\phi - \phi_i$ to $\phi + \phi_i$ by ϕ_e . The third level covers the rest of the horizon.

that action for half of the time horizon T_s . Then the process is repeated for each of the new states, producing a three-level tree. The time horizon determines how far forward we look. Too small a value and steering will be erratic, as the planner over-fits the reference curve. Too large and it becomes impossible to fit arcs to the path. However, this behavior is a function of distance rather than time, and the distance covered in a set amount of time varies with velocity. Therefore, in contrast to [21], we use a constant sampling distance D_s and define T_s in terms of it and velocity v:

$$T_s = \frac{D_s}{v}$$
.

We define possible actions in a discretized action space. Velocity control is handled outside of the state space, so we only consider changes in the turning angle ϕ . We define a discretization constant ϕ_{ϵ} which gives the smallest increments of ϕ allowed, which is necessary to keep the state space tractable. Additionally, we

define a maximum and minimum turning angle, ϕ_{max} and ϕ_{min} , with physically realistic values (see Table B.1).

Naively, we could generate the set of allowed actions as $\{\phi \mid \phi_{min} \text{ to } \phi_{max} \text{ by } \phi_{\epsilon}\}$. However this could produce potentially dangerous and uncomfortable steering. Instead we restrict the set of allowed steering angles to those that lie relatively close to our current steering angle, preventing large values of $\dot{\phi}$. We accomplish this with a parameter, ϕ_i , which gives the maximum range of movement in one time step in either direction. Thus, our action space is defined as

$$\{\phi \mid min(\phi_{min}, \phi - \phi_i) \text{ to } max(\phi_{max}, \phi + \phi_i) \text{ by } \phi_{\epsilon} \}.$$

With these parameters, our state space S is limited in size to

$$|S| \le 1 + 2\phi_i \phi_\epsilon + (2\phi_i \phi_\epsilon)^2$$

which is tractable for reasonable values of the parameters. The hierarchical state space aims to allow the agent to look further into the future with a fuller sense of the potential actions that could be taken, as forcing a single action for a long time horizon can lead to undesirable navigation choices and poor approximation of the ideal trajectory described in the following section. We limit our space to three levels to reduce the computational burden and the effects of exponential expansion.

3.3. Ideal path generation

Once the state space has been built, we must evaluate the cost of reaching each leaf state in order to choose the best action. This is accomplished by generating an "ideal" reference path that smoothly hews to the center of the lane and finding dynamically feasible trajectories that follow it as closely as possible.

The path generation algorithm takes as input a list of road points generated

by a global planning algorithm like AD* or D*-lite. From these points a cubic Hermite spline curve is computed which describes a smooth path along the road. Cubic Hermite splines consist of third degree polynomials each of which is of the Hermite form, which means that each polynomial is described by two control points and two tangent points. We use Cardinal splines [23], which are a type of cubic Hermite spline with some useful properties for this purpose: they are continuous to the first derivative (so called C(1) continuous) which allows the car to smoothly turn when following the path, the curve is guaranteed to pass through the two control points, which prevents it from straying too far from the center of the lane, and they are relatively simple and quick to compute.

Cardinal splines have a parameter τ , for tension, which governs the scaling of the derivatives. This allows more "taut" or "loose" curves around the control points. Setting $\tau=0$ produces Catmull-Rom splines, which are very smooth, while $\tau=1$ produces straight lines. Computing a segment of a Cardinal spline involves specifying the position and first derivative of the start and end of the curve. Specifically, we must find a function that has the following properties given points $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$.

$$f(0) = p_1$$

$$f(1) = p_2$$

$$f'(0) = \frac{1}{2}(1 - \tau)(p_2 - p_0)$$

$$f'(1) = \frac{1}{2}(1 - \tau)(p_3 - p_1)$$

We can find arbitrary points on f by computing the basis matrix from these

CHAPTER 3. PLANNING

equations:

$$B = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -s & 0 & s & 0 \\ 2s & s - 3 & 3 - 2s & -s \\ -s & 2 - s & s - 2 & s \end{bmatrix}$$

where $s = (1 - \tau)/2$.

Then, we have f(u) with $(0 \le u \le 1)$ as

$$U_b = [1, u, u^2, u^3]^T B$$

$$v_x = [\mathbf{p}_{1,x}, \mathbf{p}_{2,x}, \mathbf{p}_{3,x}, \mathbf{p}_{4,x}]$$

$$v_x = [\mathbf{p}_{1,y}, \mathbf{p}_{2,y}, \mathbf{p}_{3,y}, \mathbf{p}_{4,y}]$$

$$f(u) = \langle v_x \cdot U_b, v_y \cdot U_b \rangle$$

This computes a single segment of the spline. Given a list of points \mathbf{p}_i with $1 \leq i \leq n$, we can compute the spline for the entire path by computing the segment for each sub-list of size 4. That is, given a function $g(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d})$ which computes f for the points $\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}$ we compute the list of path functions as $F = \{g(\mathbf{p}_i, \mathbf{p}_{i+1}, \mathbf{p}_{i+2}, \mathbf{p}_{i+3}) \mid 1 \leq i < n-1\}$. The only problem remaining is the end points, which are not included in the spline. The solution is to generate points \mathbf{p}_0 and \mathbf{p}_n to preserve the desired properties of the end segments. We do so by simple reflection:

$$\mathbf{p}_0 = \langle 2\mathbf{p}_{1,x} - \mathbf{p}_{2,x}, 2\mathbf{p}_{1,y} - \mathbf{p}_{2,y} \rangle$$
$$\mathbf{p}_n = \langle 2\mathbf{p}_{n-1,x} - \mathbf{p}_{n-2,x}, 2\mathbf{p}_{n-1,y} - \mathbf{p}_{n_2,y} \rangle.$$

With this we can set the conditions on i in the definition of F to $0 \le i < n$ to generate the full path.

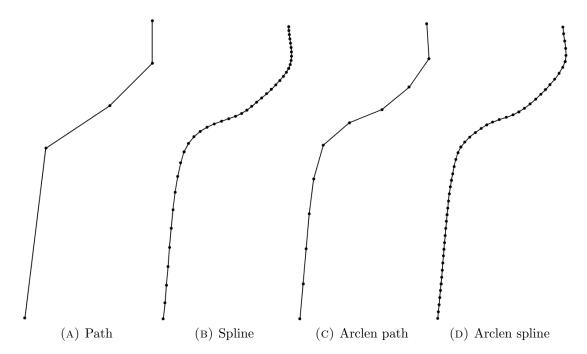


FIGURE 3.3. In (A) we have the original path as provided by the global planner. Using this path as the input to the spline algorithm produces (B). Note that the distances between points in the spline are not constant. To solve this, we reparameterize the spline by arclength, by first producing a new path with equally spaced control points (C) and then computing the spline using those points (D).

The functions F_i have an unfortunate property for our purposes: each is parameterized in the domain [0,1], which means that segments of different lengths are given the same space in the domain of F, as can be seen in Figure 3.3. This is problematic as we will be comparing this path to the trajectories traced out by the car as it travels from state to state. In order to make this comparison meaningful, we would like both functions to be parameterized by the same thing. The most obvious candidate is to parameterize by arc length, as we can compare arc lengths for any kind of curve.

Arc length on a curve c is defined by the integral

$$L(c) = \int_0^1 \sqrt{x'^2(t) + y'^2(t)} dt$$

where x(t) and y(t) give the x and y coordinates of the points at position t which ranges from 0 to 1. As there is no analytical approach that can solve this integral for arbitrary spline curves, we must use numerical techniques. However, standard numerical integration can lead to unacceptably large errors, so we make use of an adaptive integrator presented in [9]. This algorithm works by performing non-adaptive integration in two equal subintervals and subdividing recursively on each until the error reaches an acceptable level. For the non-adaptive integration we use the Gaussian Quadrature algorithm as described by [22].

The challenge is to generate a function a(s) which gives the point on the full curve that lies at arclength s. We do so using the technique similar to the one described in [28]. Let S_i be length of the path given by segment F_i , L^* be the arc length of the entire curve:

$$L^* = \sum_{i=0}^n S_i.$$

The idea is to find m points \mathbf{q} along the spline such that $L(\mathbf{q}_{i+1}) - L(\mathbf{q}_i) = \overline{l}$ for all $0 \le i < m$ where $\overline{l} = L^*/m$. That is to say, the points \mathbf{q}_i are equally spaced along the spline (see Figure 3.3c). Then, we can create a set of new spline functions G_i using \mathbf{q} as the point set instead of \mathbf{p} . Finally, we define a(s) as follows:

(3.1)
$$\sigma(s) = \begin{bmatrix} \frac{s}{\bar{l}} \end{bmatrix}$$

(3.2)
$$t_p(s) = \frac{(s - \sigma(s)\bar{l})}{\bar{l}}$$

$$(3.3) a(s) = G_{\sigma(s)}(t_p)$$

where $\sigma(s)$ gives the segment that the point at arclength s will be found in and $t_p(s)$ gives the input to the function $G_{\sigma(s)}$ that produces that point (note that the domain of each G_i is $\{0 \mid 0 \le t \le 1\}$).

We focus first on the problem of finding the points \mathbf{q}_i . For each index i

 $(0 \le i < m)$ we first find the ideal position of the point in terms of arclength as $t = i\bar{l}$. We then find the segment j such that

$$\sum_{i=0}^{j-1} L(S_i) < t < \sum_{i=0}^{j}$$

holds true, which is the segment of the original spline in which our point will be found. We also find \bar{s} , the distance along segment j that t will be found:

$$\bar{s} = t - \sum_{i=0}^{j-1} L(S_i)$$

Given \bar{s} , we need to find a corresponding number \bar{t} in the domain of $F_j(t)$, i.e.

(3.4)
$$\bar{s} = \int_0^{\bar{t}} \sqrt{x'^2(t) + y'^2(t)} dt$$

where $(x(t), y(t)) = F_j(t)$. We do so via the recursive subdivision algorithm shown in Figure A.1. At each step we compute the midpoint of the range and \tilde{s} , the arclength up to the midpoint using Equation (3.4). We then find the absolute difference $|\tilde{s} - \bar{s}|$. If it is within some ϵ , we return the midpoint as our \bar{t} . Otherwise, we recurse on the half that contains our point (the left half if $\tilde{s} > \bar{s}$ and the right half otherwise). Finally, we can compute $\mathbf{q}_i = F_j(\bar{t})$.

Once the entire set of points \mathbf{q}_i for $0 \le i < m$ has been computed, we generate a new spline using the \mathbf{q}_i as control points and compute a(s) as described in Equation (3.3).

CHAPTER 4

Action selection

In the previous chapter we described how to generate a state space for planning and how to compute a smooth, arclength parameterized, path a(s). This chapter covers how to build a motion planning system that follows that path as closely as possible while staying in its lane and avoiding static and dynamic obstacles. Determining out-of-lane points and obstacle prediction and avoidance are described in Sections 4.1 and 4.2 respectively. The path following algorithm, which keeps the agent close to the reference path, is described in section 4.3. Finally, all of these factors are combined in section 4.4 for the action selection algorithm.

4.1. Staying in lane

Legal and safe driving generally requires that cars stay in their lane and on the road. While there are exceptions (e.g., passing cars when allowed, changing lanes) in general we wish to rule out paths that do not obey this restriction. We do so by computing a polygon representing a portion of the lane which contains the agent and checking whether the points in a potential path fall within that polygon.

We first must define the boundaries of the lane as a list of points placed along the edges of the lane. For the purposes of this thesis, we take it as given that such data is available. This is justified by the presence of systems such as those referenced in Section 2.1. In simulation we generate comparable lane boundary points by shifting the center line to the right by a constant lane width to produce

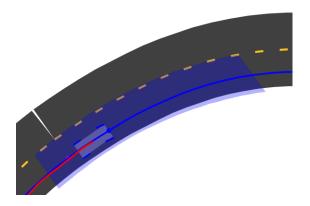


FIGURE 4.1. A depiction of the stay-in-lane system. The translucent blue region is used to validate points along each proposed trajectory using the point-in-poly algorithm in Figure A.2.

the outer edge. We define b(s) to be a function that computes these shifted points at arclength s.

We then compute points along these curves in the vicinity of the car using a value

$$s^* = \operatorname*{argmin}_s f(s) = d(a(s), \mathbf{p}_a)$$

where d is the Euclidean distance function and \mathbf{p}_a is the position of the agent. That is, s^* is the point on the spline that is closest to the agent. With that and an ϵ we find two sets of points A and B which give the lane and the road boundaries respectively:

$$A = [a(s) \mid s \in (s^* - \epsilon, s^* + D_S + \epsilon)]$$

$$B = [b(s) \mid s \in (s^* - \epsilon, s^* + D_S + \epsilon)]$$

These are discretized at a reasonable level to make the computations tractable. As roads are in general fairly smooth, even a rough discretization is effective. With these point lists, we can define a polygon P for the section of the road by the agent as the concatenation of A with the reverse of B. Figure 4.1 shows P for a stretch of road.

We want to know if an arbitrary point \mathbf{p} resides within the polygon P, the "point-in-polygon" problem from computational geometry. There are several efficient algorithms to compute this in the literature. We use one described in [11] which is valid for arbitrary polygons. The algorithm works by tracing a ray from the point and counting how many times it intersects the polygon. An even number of intersections implies that the point is outside the polygon while an odd number implies that it is inside. The full algorithm is given in Figure A.2.

We can now define the function out_of_lane that we will use in section 4.4:

$$\texttt{out_of_lane(p)} \; = \; \texttt{point_in_poly}(\mathbf{p}_x, \mathbf{p}_y, P_{xs}, P_{ys}, |P|)$$

One issue remains: how do we find s^* ? That is, given the function a(s), what arclength s corresponds most closely to the current position of the car? As a(s) is non-linear there is no general method for finding that point. However, by making the simplifying assumption that our path will be fairly smooth given that the road it follows must be navigable by a car, we can find an approximation for this point by using a simple subdivision algorithm, similar in principle to the one given in Figure A.1.

The recursive algorithm is called each time with a range (a, b) in which to search for the point. It divides the range into two halves with m = (a + b)/2 as the midpoint and computes the point at the center of each segment. The distance from each point to the agent position is computed and whichever segment is closer is recursed upon. This is repeated to some constant recursion depth, at which point the closest midpoint is returned.

4.2. Pedestrian avoidance

We model pedestrians as points with state (x, y, \vec{v}) , where (x, y) is its position and \vec{v} its velocity vector. Given a sequence of positions as detected by the sensing

system (as in Section 2.1) we choose the two most recent positions $\mathbf{p}_1 = \langle x_1, y_1 \rangle$ and $\mathbf{p}_2 = \langle x_2, y_2 \rangle$ at times t_1 and t_2 respectively (with $t_1 > t_2$). Then we compute the current state as

$$(x, y, \vec{v}) = \left(x_1, y_1, \frac{\mathbf{p}_1 - \mathbf{p}_2}{t_1 - t_2}\right)$$

Here we set the position to its most recently observed position, while setting its velocity to the vector from \mathbf{p}_2 to \mathbf{p}_1 scaled by the inverse of the time delta.

We then check whether the pedestrian is in the road (using a procedure analogous to the one in section 4.1) or heading into the road according to its θ . Those that are not are eliminated from consideration. This step is necessary to avoid giving excessive space to pedestrians walking along the road.

For each of the n remaining pedestrians p_i we compute a set of functions K

(4.1)
$$\kappa_i(t) = (p_i + \vec{v}_i t, \ 3 + t)$$

(4.2)
$$K = \{ \kappa_i \mid i \in [0, n) \}$$

each of which defines a circle for the pedestrian in terms of (center, radius) at each time t. Based on the assumption that the pedestrian will continue along its current vector of motion, at time t = 0 the pedestrian is placed at its current position, while larger values of t project it along its velocity vector. The radius starts at a reasonable value for safely avoiding pedestrians (here, 3 meters) and increases linearly with time to represent the additional uncertainty. Using this set we define a predicate (Figure 4.2) which informs whether an action will intersect the circle at any point within the stopping distance D_{st} .

While these ideas are presented in terms of pedestrians, they are applicable to any moving obstacle that exhibits similar motion: generally forward, with changes in direction accomplished by slow turning. Stop signs and traffic lights can also be modeled in this manner as virtual obstacles generated for so long as necessary.

FIGURE 4.2. This predicate determines whether a car at point \mathbf{p} at time t might with a pedestrian κ . Here D_{st} is the stopping distance, which is the minimum distance the car can be safely stopped (defined in terms of the maximum comfortable braking speed Br_{max}). Implicit are the variables pos, for the agent's current position, and l, for the car's length. D_p is the distance from the pedestrian to stop. The first clause of the conditional prevents the car from slowing too soon, while the second checks for actual collisions.

4.3. Path cost

In addition to staying in its lane and avoiding obstacles, we would like the agent to smoothly navigate along the road to its destination. In Section 3.3 we computed a function a(s) which gives points along the reference path as a function of arclength. In this section we define the path cost function $c_p(t, \mathbf{p}, \theta, \phi, v)$ which enables the path following behavior. A cost function should provide an accurate measure of how undesirable a particular action is. In the context of the path following system, this is a measure of how similar the path produced by a particular action matches the ideal path over a time horizon T_s .

There are two components to the path cost function: the cost of the current action and the error induced by taking the previous action. The error term e helps prevent small errors from compounding over time, resulting in drift from the reference path. The path cost is computed as the sum of absolute angle deltas between the agent's heading vector and the tangent vector at the corresponding point on the ideal path function. We can define the angular difference $d_a(\theta, \iota)$

between angles θ and ι as

(4.3)
$$d_a(\theta, \iota) = atan2(sin(\theta - \iota), cos(\theta - \iota)).$$

With that we define the path cost function, which takes a state as input and returns the cost of being in that state:

(4.4)
$$c_p(t, \mathbf{p}, \theta, \phi, v) = |d_a(\theta, \theta_p(s^* + tv)) + e|$$

where $\theta_p(s)$ gives the direction of the vector tangent to the reference path a(s) at point s. It also makes use of s^* , the arclength corresponding to the point on the reference path closest the agent (see Section 4.1) and T_s , the time horizon.

The last step is finding the error term e. This is computed as the signed sum of the angle differences between the values of θ taken on by the vehicle in the previous time step and those tangent to the reference path for the same period.

4.4. Action selection

In this section we describe the action selection system, which is responsible for choosing the best action (ϕ, a) at each navigation step. Our goal is to find the action which will cause the car to most closely follow the reference path while avoiding obstacles and staying in the lane. The idea: eliminate actions that could lead to collisions and most of those that cause the car to leave the lane, then minimize the path cost function c_p for those actions remaining. If no actions remain, slow down.

The first step is to generate the full state space. In Section 3.2 we described how the state space is expanded. In short, we make two choices of ϕ within the discretized set of allowed choices. We are therefore computing the cost of pairs (ϕ_1, ϕ_2) . As we describe the full cost function, each component will operate on a list of 2n states $(t, \mathbf{p}, \theta, \phi, v)$, where n is the sampling frequency. For every pair

 (ϕ_1, ϕ_2) , we compute the full state S as

$$S = \left\{ (t, \Pi(t, \phi_1, A), \Theta(t, \phi_1, a), \phi_1, V(t, A)) \mid i \in (1, n), t = \frac{i}{n} \frac{T_s}{2} \right\} \cup \left\{ (t, \Pi(t, \phi_2), \Theta(t, \phi_2), \phi_2, V\left(\frac{T_s}{2} + t, A\right)) \mid i \in (1, n), t = \frac{T_s}{2} \left(1 + \frac{i}{n}\right) \right\}$$

where $\Pi(t, \phi, v)$ is a function that gives the car's position after traveling t seconds with acceleration a and turning angle ϕ , $\Theta(t, \phi, a)$ is a comparable function that gives the car's heading angle, V(t, a) is a function that gives the velocity after accelerating at a for t seconds and T_s is the time horizon. We compute S as the union of two sets, each responsible for half the time horizon. As velocity control is handled separately, we use a constant acceleration A that is the maximum acceleration we might choose: A_{max} unless we are traveling above the maximum velocity V_{max} .

We now apply these state sets to the functions described in the previous sections. First, obstacle avoidance. Given the set of obstacle functions K from Equation (4.2) and the might_hit predicate from Figure 4.2, we can define a function safe(S) which determines whether the actions leading to state set S might cause the car to hit something:

$$safe(S) = \begin{cases} \text{true} & \text{if } \exists \kappa \in K, (t, \mathbf{p}, \theta, \phi, v) \in S \text{ s.t. } \texttt{might_hit}(\mathbf{p}, t, \kappa) \\ \text{false} & otherwise. \end{cases}$$

We simply check for collisions with every obstacle at every position in the state space.

The next component is the in_lane_q function, which determines the fraction of points for a given state set that lie outside the lane. Once computed, we reject actions with quotients above a certain level and penalize the rest. We do not

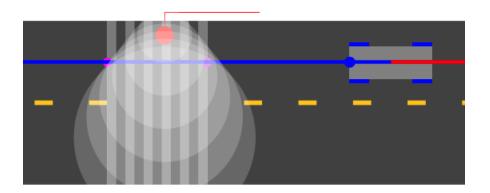


FIGURE 4.3. A pedestrian (red dot) crosses the street, blocking the agent's path. The agent predicts the pedestrian's future trajectory, shown here as translucent white circles, with each circle representing a range of possible positions at a particular time t.

reject all such actions, as it may be preferable to steer outside the lane in some cases, for example to give room to a bike riding along the road or another car. We define the function as

$$(4.5) in_lane_q(S) = \frac{|\{\mathbf{p} \mid (t, \mathbf{p}, \theta, \phi, v) \in S \text{ where } out_of_lane(\mathbf{p})\}|}{2n}.$$

We compute the set of points that lie outside the lane, compute the cardinality of that set, and divide by 2n, the total number of points.

With these and the path cost function $c_p(S)$ defined in the previous section we can write the full cost function:

$$(4.6) c(S) = \begin{cases} c_p(S)(1 + in_lane_q(S)\alpha) & \text{if } safe(S) \text{ and } in_lane_q(S) < \beta \\ \infty & \text{otherwise} \end{cases}$$

where α and β are parameters that can be tuned to control lane behavior. We then choose the state set that minimizes the cost function:

$$(4.7) S^* = \underset{S}{\operatorname{argmin}}(c(S))$$

and choose our turning angle ϕ^* as the ϕ_1 used to generate S^* . However, we must still handle the case where $c(S^*) = \infty$. This occurs when there are no valid actions, as in Figure 4.3. In that case, we leave ϕ unchanged from the previous action, but acceleration is set to the maximum braking acceleration Br_{max} . With this, we choose the next action a as

(4.8)
$$a = \begin{cases} (\phi^*, \text{ if } v < V_{max} \text{ then } A_{max} \text{ else } 0) & \text{if } c(S^*) < \infty \\ (\phi, Br_{max}) & \text{otherwise.} \end{cases}$$

CHAPTER 5

Methods and demonstrations

To demonstrate the suitability of these methods to autonomous vehicular navigation, we have implemented the algorithms described in Chapters 3 and 4 as well as a simulation environment. In section Section 5.1 we describe the simulator and other implementation details. Section 5.2 describes the experimental methods used. In Section 5.3 we list the metrics by which the effectiveness of our system is measured. Section 5.4 describes the demonstrations run, and Section 5.5 analyzes the results.

5.1. Simulation environment

The navigation system and simulation were implemented in Scala [2], a statically typed language that runs on the Java Virtual Machine (JVM). Scala was chosen for its speed, rich type system, ease of programming and simple integration with Java libraries. The system was built using the Actor model of concurrency, wherein each independent computation unit (actor) is allowed to communicate with the others solely by sending and receiving messages. This eliminates many of the traditional pitfalls of shared-memory concurrency as well as the need for controls like mutexes or semaphores. We use the actor implementation provided by the Akka [1]. concurrency library for Scala.

The actor model is particularly advantageous for our domain, as it allows simulations to be modeled closely to physical robotics systems. For example, a sensor actor can be responsible for surveying the environment and reporting information to the agent when it comes within sensing distance. As the agent can only rely on the information it receives, it is harder for it to "cheat" and make decisions based on information it could not know in a physical environment. Therefore we can model the agent as an actor that receives information about its environment from sensors, and then chooses and sends an action to the simulation environment, which computes the new state of the agent.

As result of this design the simulation is completely asynchronous and runs in real-time. By necessity, therefore, the navigation system must be fast enough to react to a changing environment. This requirement gives guidance as to which algorithms are usable in the real-time context of robotics where responding too slowly can be dangerous.

5.2. Methods

To evaluate the effectiveness of the motion planning system described here, we ran it on various scenarios, described in Section 5.4. A simulation scenario is defined in terms of several parameters: a road map, a starting state $(\mathbf{p}, \phi, \theta, v)$ for the agent, a set of dynamic or stationary obstacles and a destination. The map is defined as a weighted graph, with nodes representing positions in space and edges representing roads between those positions. The agent is provided with the road map at initialization, but other environmental information must be obtained from the simulated sensors.

The agent begins by computing a path to its destination using A* search. It then proceeds through the steps of navigation as described in Chapters 3 and 4. Once an action (ϕ, a) has been chosen it is sent to the simulator, which responds with the new state $(\mathbf{p}, \phi, \theta, v)$ of the agent. When an obstacle is present within the sensing range of the car, its position and an identifier are sent to the agent,

which allows the agent to track the trajectory of the obstacle.

The simulator is responsible for continually updating the state of the agent according to the most recent action chosen. This requires finding the current position, heading angle and velocity according to the vehicular motion mode in Section 3.1. The state is updated approximately every 100 milliseconds, a time frame chosen to balance computational expense with realism. After every state computation the state of the simulation is logged for later analysis. Each test is run for the shorter of 60 seconds or until the agent reaches its destination.

All tests were run on a machine with an Intel Core 2 Duo 2.93 Ghz processor and 4GB of memory using Scala 2.9.1. The asynchronous nature of the simulator leads to small amounts of nondeterminism as the exact timing of the navigation loop can affect the results. To establish the importance of these effects and to produce reliable results, each test was run 50 times, and for each metric the mean and standard deviation was computed over all trials.

5.2.1. Parameters. The motion planner has several parameters which can be tuned. We list the values used for the demonstrations in Table B.1. Chapters 3 and 4 give more explanation of the variables.

5.3. Metrics

In the following demonstrations, we evaluate our system on various metrics which reflect smoothness, path-following, safety, and speed. They are described here.

5.3.1. Smoothness. Smoothness is an important requirement of an autonomous navigation system meant to be used by humans, and satisfying it places some restrictions on the range of available actions. The aim is to provide comfort for the

passengers by limiting longitudinal (in the direction of the motion) and lateral acceleration. Table B.2 lists the effects of acceleration on the human body according to the ISO 2631-1 standard governing mechanical vibration and shock as cited in [17]. A comfortable drive should limit overall root mean squared acceleration a_w to $< 1 \text{ m/s}^2$.

We can measure this in terms of changes in v and θ over time. We can define the acceleration vector as

(5.1)
$$\vec{a} = \frac{d\vec{v}}{dt} = \frac{dv}{dt}u_T + v\frac{d\theta}{dt}u_N$$

where \vec{v} is the velocity vector, v is the scalar velocity in the direction of the car's motion (i.e., the number displayed by the speedometer) and u_T and u_N are unit vectors which respectively lie tangent and normal to heading angle θ . Given that, we can define a_w as

$$(5.2) a_w = \sqrt{k_x^2 a_{wx}^2 + k_y^2 a_{wy}^2}$$

where a_{wx} and a_{wy} are the components of acceleration (as defined in Equation (5.1)) with regards to the x and y axes respectively, and k_x and k_y are multiplicative factors with set as $k_x = k_y = 1.4$ to conform to the ISO standard.

For the two components of acceleration, a_t and a_l as well as their combination a_w we report two values: the mean (denoted as $\overline{a_t}$, $\overline{a_l}$, $\overline{a_w}$) and the maximum (max a_t , max a_l , max a_w).

5.3.2. Path following. Given our ideal path (as generated in Section 3.3) we the agent's path-following effectiveness it in terms of position and θ . Due to the smoothness properties of Cardinal splines, closely following it should produce a smooth, efficient ride.

We measure path following as the Euclidean distance for position and absolute

angular distance for heading angle. Given ideal positions \mathbf{p}_i and actual positions \mathbf{q}_i we define the positional divergence as the sum of euclidean distances:

(5.3)
$$\Delta p = \sum_{i=1}^{n} |\mathbf{p}_i - \mathbf{q}_i|$$

and given ideal heading angles (directions of the vector tangent to the ideal path) θ_i and actual heading angles ι_i we define the heading angle divergence as in Equation (4.4):

(5.4)
$$\Delta \theta = \sum_{i=1}^{n} |atan2(sin(\theta_i - \iota_i), cos(\theta_i - \iota_i))|.$$

5.3.3. Safety. We measure safety as the minimum distance from any obstacle during a run, as even a single close encounter with a pedestrian is dangerous. For all pedestrian position sets $P \in \mathcal{P}$ and for all points $\mathbf{p}_i \in P$ we find the minimum Euclidean distance from the agent positions \mathbf{q}_i :

$$(5.5) \hspace{3.1em} s=\min\left\{|\mathbf{p}_i-\mathbf{q}_i|\ |\ i\in[1,n], p\in P\right\}.$$

This metric does a better job of presenting the danger posed by the vehicle than other aggregates like mean closest distance over all pedestrians.

5.3.4. Speed. We consider maximum and mean velocity:

$$(5.6) v_{max} = max\{v_i\}_1^n$$

$$\bar{v} = \frac{\sum_{1}^{n} v_i}{n}.$$

Both of these measures are problematic. Mean velocity is highly dependent on the details of the scenario (specifically, the stops necessary to accommodate pedestrians or follow road rules) and deriving an ideal mean velocity for a specific scenario is difficult. Maximum velocity does not suffer from this issue, but is not very representative of the performance across the test. Note that the two main issues with

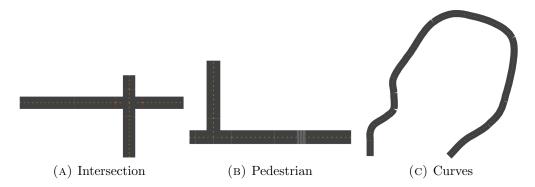


FIGURE 5.1. The three scenarios used to evaluate the agent.

higher velocities are the increased stopping time and the fewer motion planning operations per distance covered both of which give the car less reaction time.

5.4. Demonstrations and results

We evaluated our motion planner on three scenarios, each of which demonstrates a different challenging aspect of driving. The scenarios are shown in Figure 5.1. Intersection, described in Section 5.4.1, requires the agent to navigate a four-way stop and make a left turn. Pedestrian, Section 5.4.2, demonstrates pedestrian motion prediction and avoidance. Curves, Section 5.4.3 involves navigating a course with curves of varying degrees. Together these scenarios show the motion planner solving common and difficult navigation problems.

For each scenario, we show the ideal and actual trajectories of an example run as well as a series of graphs showing the evolution of the metrics over time. The aggregate data from all trials and scenarios is listed in Table 5.1. The subsections that follow describe each scenario in more depth, while the data are analyzed together in Section 5.5.

5.4.1. Intersection. This scenario demonstrates two challenges: smoothly stopping at a stop sign (denoted as a red dot in Figure 5.2) and making sharp

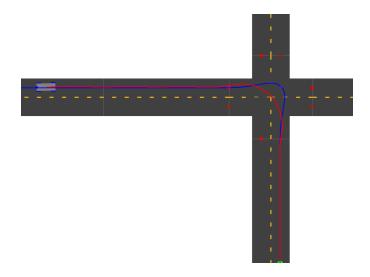


FIGURE 5.2. The intersection scenario. The red dots represent stop signs, while the red line gives the agent's path through the map.

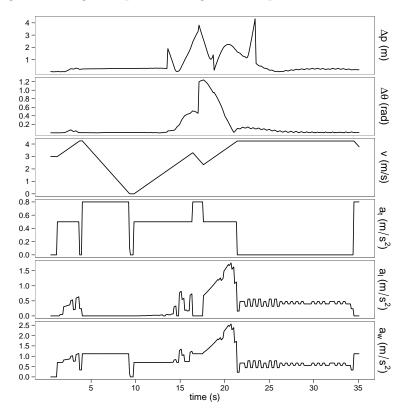


Figure 5.3. Results from a run through the intersection scenario.

CHAPTER 5. METHODS AND DEMONSTRATIONS 5.4. DEMONSTRATIONS AND RESULTS left-hand turns. Figure 5.3 presents six graphs relating the metrics to time from the start of the run. From top to bottom they give Δp , the distance between the closest point on the ideal curve and the agent's position, $\Delta\theta$, the angular distance between the vector tangent to the ideal path at that point and the agent's heading angle, v, longitudinal velocity, a_t , tangent (longitudinal) acceleration, a_l , lateral acceleration, and a_w , the root mean square acceleration. For this scenario the maximum speed was set to 4 m/s, or about 9 mph. In each trial the agent began at the green dot with heading angle $\pi/2$, i.e., in the direction of the positive yaxis. The car had no difficulty following the path for the straight segment. At about the four second mark it starts slowing down, stopping just before the stop sign. It then begins executing the turn, first by turning a bit to the right then attempting the left turn. Due to the placement of the path point (pink dot) in the intersection, the ideal curve is sharp and dynamically unfeasible. However this does not present a problem for the agent, which makes the sharpest turn it can until it reaches the next straight part of the ideal path curve, at which point it quickly corrects itself to return to the ideal curve. The motion is natural, aside from a brief point at about 18 seconds where the agent gets momentarily stuck due to the lane restrictions, causing it to slow for a moment until it once again finds a feasible maneuver. See Section 5.5 for more analysis of the agent's performance.

5.4.2. Pedestrian. This scenario demonstrates pedestrian motion prediction and avoidance (in this case by coming to a stop) and making sharp right turns. Figure 5.4 shows agent and pedestrian trajectories from a run and Figure 5.5 shows the metrics over time. The agent begins at the green dot (on the far right) with initial heading angle $\theta = \pi$ and velocity v = 3 m/s with $v_{max} = 4$ m/s. Two deterministic pedestrians are present. One, denoted by a red dot with a thin red trajectory line, moves along the road until it gets to the crosswalk (the

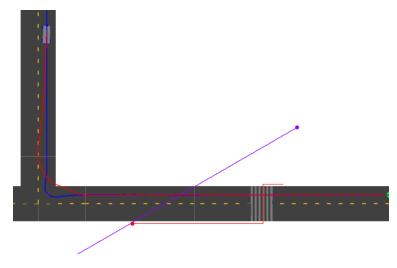


FIGURE 5.4. The pedestrian scenario. The car must slow to avoid hitting the two pedestrians (red and purple dots with associated trajectories) as well as make the tight right turn. The thick blue line gives the ideal car trajectory and the thick red line is the car's actual trajectory.

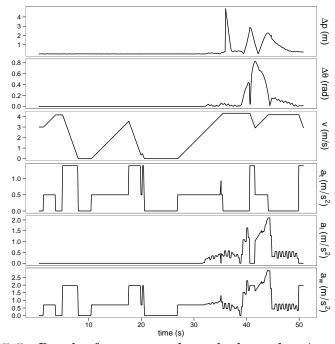


FIGURE 5.5. Results from a run through the pedestrian scenario.

striped lines) at which point it turns to cross (Figure 5.6). The agent detects its change in heading angle, predicts its future trajectory (Figure 4.3) and rules out



FIGURE 5.6. At 4 seconds, the red pedestrian turns into the crosswalk.

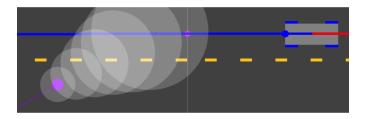


FIGURE 5.7. At 14 seconds, a pedestrian crosses the road diagonally. Each set of possible positions at a future time t is represented as a translucent circle.

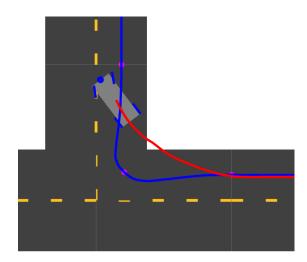


FIGURE 5.8. At 40 seconds the car finishes its turn, overshooting the ideal path before correcting itself, as can be seen by the orientation of the wheels).

all possible actions, causing it to brake. Once the pedestrian moves out of the lane, after the 10 second mark, the car begins to accelerate again, only to encounter the purple pedestrian (Figure 5.7). Given its heading angle, the agent again

CHAPTER 5. METHODS AND DEMONSTRATIONS 5.4. DEMONSTRATIONS AND RESULTS determines that there is a high probability of collision and slows down to avoid it. For the right turn (Figure 5.8, the ideal path is again dynamically unfeasible, so the agent approximates it, overshooting the ideal path before correcting. At about 40 seconds there is a sharp spike in Δp , the difference between the ideal position and the actual position of the agent but with no corresponding increase in $\Delta \theta$. This results from a limitation of the closest point finding algorithm presented in Section 4.1. Due to the sharpness of the curve, for a fraction of a second it chooses the wrong point s^* along the ideal path, as the algorithm can fall into local minima with sharply non-linear functions. This does not affect navigation materially due to briefness and because the tangent vector at that point is very close to that of the correct point. This scenario was designed to show obstacle prediction and avoidance in typical (the red pedestrian) and somewhat adverse (the purple pedestrian) situations. In both, the agent is able to predict their motion and safely avoid them, approaching no closer than 4.5 m from either.

5.4.3. Curves. This scenario demonstrates path following around a road with several tight curves (Figure 5.9) of varying degrees. The error correction system (described in Section 4.3) is crucial for success here, without which small errors on each turn can accumulate, driving the car off of the road. We ran two sets of tests on this course, one with a maximum velocity $v_{max} = 9$ m/s and another with $v_{max} = 4$ m/s, shown in Figures 5.11 and 5.12 respectively. Even at the faster speed the agent is able to successfully navigate the course in every case, straying only 0.5 meters on average from the reference path. However, the limitations of the velocity control system lead to very high lateral accelerations as the car does not slow down around curves. In the slower run through lateral accelerations are more manageable, peaking around 1.2 m/s². Note that to make scales comparable between the two runs, both were limited to 60 seconds. Due to

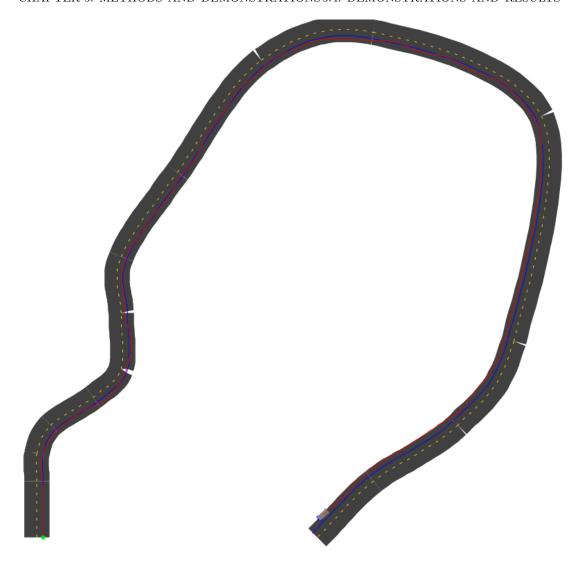


FIGURE 5.9. Curves scenario, with $V_{max} = 9$ m/s. The agent must navigate around a course with several tight curves. The blue line gives the ideal trajectory of the agent while the red line gives its actual trajectory.

the length of the course the agent at the slower speed was able to make it through about half the course in that time span. In both cases, there is a large spike in the Δp graph, at 55 and 30 seconds for slow and fast respectively. This has the same cause as the spike in the pedestrian scenario: a temporary mis-estimation of s^* .

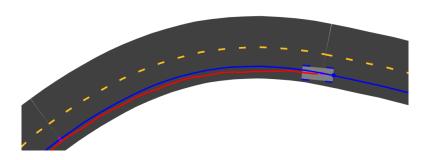


FIGURE 5.10. The agent navigating the upper left part of the curves scenario at about 29 seconds. The agent undershoots the curve a bit, correcting itself as it enters the straightaway.

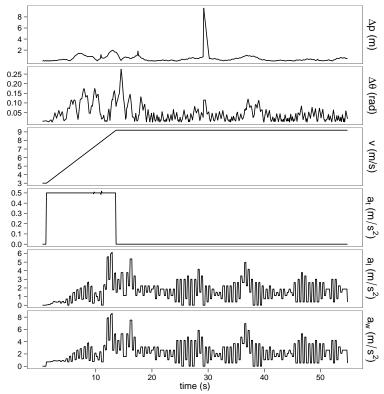


FIGURE 5.11. Results from a run through the curves scenario at a relatively high speed (9 m/s), which results in very high lateral accelerations.

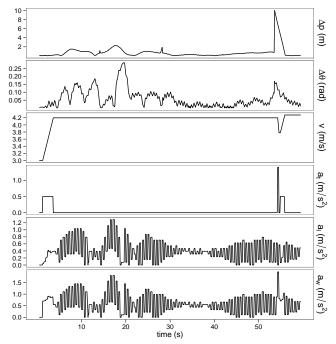


FIGURE 5.12. Results from a run through the curves scenario at a slower speed (4 m/s). Given the 60 second time limit, approximately half the course was traversed.

5.5. Analysis

Of the four metric groups discussed in Section 5.3, the data show good results for the final three: path-following, safety and speed. At its worst, in the slow curves scenario, the agent strays only 0.75 m on average from the ideal path. This behavior can be seen in the trajectories recorded during each scenario. At high speeds on the curve scenario, the agent does even better, straying only half a meter on average. These results show the strength of the error correction mechanism, which causes the agent to return to the reference path after over or undershooting the ideal curve. This behavior can be seen in Figures 5.8 and 5.10. Looking at the path following over time, in the curves scenario Δp stays relatively constant, aside from the outlying spikes. The error in other two scenarios is almost all accounted for in the turns, resulting from the inability for the car's to follow the prescribed

	intersection	pedestrians	curves (fast)	curves (slow)
$\overline{a_t}$	0.34 (0.01)	0.39 (0.02)	0.12 (0.01)	0.04 (0.01)
$\max a_t$	0.83(0.11)	1.41(0.02)	1.04(0.43)	1.40(0.00)
$\overline{a_l}$	0.34(0.01)	0.21(0.03)	1.58 (0.03)	0.40(0.01)
$\max a_l$	1.81 (0.18)	2.10(0.13)	6.43(0.33)	1.35(0.07)
$\overline{a_w}$	0.86(0.02)	0.81(0.04)	2.28(0.04)	0.60(0.01)
$\max a_w$	2.62(0.24)	2.96(0.18)	9.03(0.46)	1.98(0.07)
Δp	0.65 (0.05)	0.38(0.23)	0.55(0.02)	0.75(0.02)
$\Delta \theta$	0.13(0.01)	0.06(0.03)	0.05(0.00)	0.06(0.00)
\overline{v}	3.15(0.03)	2.36(0.19)	8.42(0.03)	4.19(0.03)
$\max v$	4.26(0.03)	4.26(0.03)	9.25(0.06)	4.27(0.04)
<i>S</i>		4.51 (0.13)	• ,	. ,

TABLE 5.1. Aggregated data from 50 runs of each demonstration. Means over all runs are listed with standard deviations in parentheses. The safety metric s is only applicable to the pedestrian scenario and is therefore omitted for the others.

path. However, after completing the curve in both instances it is able to correct itself and return to the path. $\Delta\theta$ has a similar evolution. At worst it is on average 0.13 radians, or about 2% of a circle. During the sharp turns in the first two scenarios it peaks at about 1.2 radians, or 20%, but is again quickly corrected.

Velocity and safety are simpler to evaluate. In the case of the former, the agent is able to traverse the curves scenario even at the relatively high speed of 9 m/s (about 20 mph), while in the others speeds were limited to 4 m/s (about 9 mph) to allow the agent to make the sharp turns at the intersections. In the pedestrian scenario, the safety metric s was 4.5 m, which is clearly a "safe distance," the only guidance given by US traffic laws.

The smoothness story is more complicated. The tangent (longitudinal) acceleration a_t is generally reasonable, maxing out at 1.4 m/s² in the pedestrian scenario. This is about the acceleration felt when going from surface road speeds

(25 mph) to highways speeds (65 mph) in 13 seconds. However the lateral acceleration a_l is less reasonable. This is particularly true on the fast curves scenario, where the higher speeds result in lateral accelerations of nearly 6.5 m/s², well into the "extremely uncomfortable" range according to Table B.2.

Another concern is the high-frequency oscillations sometimes produced by controller, as can be seen in the plots of $\Delta\theta$. This is often a problem with control systems that incorporate feedback. In our case, even as the agent gets very close to the correct θ it will generally be high or low by a small amount. This gets fed back into the next iteration of the navigation loop, which causes an overcorrection, leading to a cycle of small adjustments around the correct value.

CHAPTER 6

Discussion and future work

The previous chapter demonstrated the motion planner's effectiveness in several scenarios that we believe reflect difficult planning problems. In every case the agent was able to successfully navigate the course while maintaining safety. The system performs particularly well in regards to path-following: in the full curves test it manages to average just 0.55 m distance from the reference path and 0.05 radians from its tangent vector. Where it deviates, as in the tight turns in the intersection and pedestrian scenarios, it does so because the reference path is impossible for an Ackermann vehicle to follow. These results improve on the controller described in [21], which strays as far as 8m in an environment similar to the curves scenario. Our acceleration data also compare favorably to controllers in [17] and [21], which achieve similar results only by limiting velocity. The slow curves scenario takes this approach, leading to a much better maximum of 1.35 m/s², with only 0.4 m/s² on average.

This advantage may be due to the use of a hierarchical state space in our controller, which permits a large sampling distance. By looking further ahead, we are able to respond more optimally to future changes in path orientation. Similarly, by using a constant sampling distance rather than time, performance is largely independent of velocity and thus the same parameters could be used in each of our scenarios.

Our safety system, which works by state generation and elimination, allows simple integration with obstacle prediction systems. While the one presented here is simplistic, it is effective in our limited tests. In the pedestrian test the agent maintains a minimum distance of about 4.5 m from both pedestrians, even though they behave recklessly by walking in front of the car. However further tests are needed to demonstrate effectiveness with arbitrary pedestrians. Realistic pedestrian simulation is a significant research project in and of itself (e.g., [15], [10]), and the ethical and legal concerns of testing in real-world environments makes this a challenging problem.

The high-frequency oscillations sometimes produced by our controller are problematic. In other controller systems, these have issues have been approached by carefully tuning the weights of the different components (e.g., [25]) of the error term, and a similar approach may be effective here. Another possibility is to apply a smoother to the control loop [3], however our attempts at this lead to instability, producing lower-frequency oscillations that eventually drove the car off the road. Removing or scaling down the feedback component entirely produced good results in some tests but not others, particularly the curve scenario. Without feedback the agent's actions can get progressively more divergent from the reference path, leading eventually to failure.

6.1. Future work

The two most pressing issues discussed in the previous section are the high lateral accelerations and the oscillatory behavior of the control system. A more adaptable velocity control system would help solve the former. The current velocity controller has only two options: accelerate at the maximum allowed rate or brake at the maximum allowed rate. Instead, the agent should decelerate when executing curves to a velocity that would produce lateral acceleration below 1 m/s^2 . This could possibly be accomplished by integrating acceleration into the

action space and including lateral acceleration into the cost function.

Improving the obstacle motion prediction system would also increase the applicability of the planner. Some work has been done in this area in the context of pedestrians (e.g., [18], [14]) using machine learning to predict future obstacle positions. Vehicle motion predictions are also very important for a real car. Fortunately, the basic approach of our motion planner (ruling out likely invalid trajectories based on collision probabilities) seems easily extensible to better sources of collision probabilities.

6.2. Conclusion

In this thesis we described a motion planning system for autonomous vehicles with several novel features. By generating an ideal, smooth path and evaluating dynamically feasible vehicle trajectories against it, our planner is capable of producing smooth, efficient driving, showing that arclength-parameterized Cardinal splines can be effective in this domain. By using a hierarchical state space which allows two choices of steering angle, we extended the range of planning. And by controlling the orientation drift through the addition of an error term, our agent is able to very closely follow the reference path and correct for small missteps.

We developed an obstacle prediction system and integrated it into the motion planner by ruling out actions leading to states with a high probability of collision. In the same manner we developed a system for controlling lane behavior. In both cases, the desired behaviors were observed, while serving as examples of how more sophisticated systems might be integrated into the planning infrastructure.

We demonstrated the effectiveness of our planner in several simulated scenarios that require solutions to challenging driving problems, and in each our planner successfully navigated to its destination while behaving safely around pedestrians and closely following the reference path. By implementing our simulator asynchronously and in real-time, we showed that our planner is usable in the performance-sensitive area of robotics. However, testing in simulation, while necessary, is not sufficient to prove effectiveness in the physical world of robotics, and future work should include implementation in a robotic vehicle.

Bibliography

- [1] Akka. http://akka.io/.
- [2] The scala programming language. http://www.scala-lang.org/.
- [3] K. H. Ang, G. Chong, and Y. Li. Pid control system analysis, design, and technology. Control Systems Technology, IEEE Transactions on, 13(4):559 – 576, july 2005.
- [4] M. Buehler, K. Iagnemma, and S. Singh. The DARPA Urban Challenge: Autonomous Vehicles in City Traffic (Springer Tracts in Advanced Robotics). Springer, 1 edition, Nov. 2009.
- [5] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In C. Schmid, S. Soatto, and C. Tomasi, editors, *International Conference on Computer Vision & Pattern Recognition*, volume 2, pages 886–893, INRIA Rhône-Alpes, ZIRST-655, av. de l'Europe, Montbonnot-38334, June 2005.
- [6] A. Ess, K. Schindler, B. Leibe, and L. Van Gool. Object detection and tracking for autonomous navigation in dynamic environments. *Int. J. Rob. Res.*, 29(14):1707–1725, Dec. 2010.
- [7] D. Ferguson, M. Darms, C. Urmson, and S. Kolski. Detection, prediction, and avoidance of dynamic obstacles in urban environments. In 2008 IEEE Intelligent Vehicles Symposium, pages 1149–1154, 2008.
- [8] D. Ferguson, T. M. Howard, and M. Likhachev. Motion planning in urban environments. J. Field Robot., 25(11-12):939–960, Nov. 2008.
- [9] B. Guenter and R. Parent. Computing the arc length of parametric curves. *Computer Graphics and Applications, IEEE*, 10(3):72 –78, May 1990.
- [10] S. J. Guy, J. Chhugani, S. Curtis, P. Dubey, M. Lin, and D. Manocha. Pledestrians: a least-effort approach to crowd simulation. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '10, pages 119–128, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.
- [11] E. Haines. *Graphics Gems IV*, chapter Point in Polygon Strategies, pages 24–46. Academic Press, 1994.
- [12] J. Hancock, M. Hebert, and C. Thorpe. Laser intensity-based obstacle detection. In In Proceedings of the IEEE Conference on Intelligent Robots and Systems, 1998.
- [13] P. E. Hart, N. J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. Systems Science and Cybernetics, IEEE Transactions on, 4(2):100–107, July 1968.
- [14] S. Imagerie, A. Dizan, V. Govea, M. Augustin, L. Prsident, M. Roland, M. Michel, M. Wolfram, B. Examinateur, M. Hiromichi, Y. Examinateur, M. Olivier, A. Examinateur, M. Christian, and L. Directeur. Incremental learning for motion prediction of pedestrians and vehicles.
- [15] I. Karamouzas, P. Heil, P. van Beek, and M. Overmars. A predictive collision avoidance model for pedestrian simulation. In A. Egges, R. Geraerts, and M. Overmars, editors, *Motion in Games*, volume 5884 of *Lecture Notes in Computer Science*, pages 41–52. Springer Berlin / Heidelberg, 2009.

BIBLIOGRAPHY BIBLIOGRAPHY

[16] S. Koenig and M. Likhachev. D* lite. In Proceedings of the AAAI Conference of Artificial Intelligence (AAAI), pages 476–483, 2002.

- [17] L. Labakhua, U. Nunes, R. Rodrigues, and F. S. Leite. Smooth trajectory planning for fully automated passengers vehicles: Spline and clothoid based methods and its simulation. In J. A. Cetto, J.-L. Ferrier, J. M. Costa dias Pereira, and J. Filipe, editors, *Informatics in Control Automation and Robotics*, volume 15 of *Lecture Notes in Electrical Engineering*, pages 169–182. Springer Berlin Heidelberg, 2008.
- [18] F. Large, D. Vasquez, T. Fraichard, and C. Laugier. Avoiding cars and pedestrians using velocity obstacles and motion prediction. In *Intelligent Vehicles Symposium*, 2004 IEEE, pages 375 379, june 2004.
- [19] M. Likhachev and D. Ferguson. Planning long dynamically feasible maneuvers for autonomous vehicles. *Int. J. Rob. Res.*, 28(8):933–945, Aug. 2009.
- [20] M. Likhachev, D. Ferguson, G. Gordon, A. T. Stentz, and S. Thrun. Anytime dynamic A*: An anytime, replanning algorithm. In Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS), June 2005.
- [21] K. Macek, R. Philippsen, and R. Siegwart. Path following for autonomous vehicle navigation with inherent safety and dynamics margin. In *Intelligent Vehicles Symposium*, 2008 IEEE, pages 108 –113, june 2008.
- [22] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. Numerical Recipes 3rd Edition: The Art of Scientific Computing. Cambridge University Press, New York, NY, USA, 3 edition, 2007.
- [23] I. J. Schoenberg. *Cardinal Spline Interpolation*. Society for Industrial and Applied Mathematics, Philadephia, PA, 1973.
- [24] G. Seetharaman, A. Lakhotia, and E. Blasch. Unmanned vehicles come of age: The darpa grand challenge. *Computer*, 39(12):26 –29, dec. 2006.
- [25] S. Skogestad. Simple analytic rules for model reduction and pid controller tuning. *Journal of Process Control*, 13(4):291 309, 2003.
- [26] A. Stentz and I. C. Mellon. Optimal and efficient path planning for unknown and dynamic environments. *International Journal of Robotics and Automation*, 10:89–100, 1993.
- [27] C. Urmson, J. Anhalt, D. Bagnell, C. Baker, R. Bittner, M. N. Clark, J. Dolan, D. Duggins, T. Galatali, C. Geyer, M. Gittleman, S. Harbaugh, M. Hebert, T. M. Howard, S. Kolski, A. Kelly, M. Likhachev, M. McNaughton, N. Miller, K. Peterson, B. Pilnick, R. Rajkumar, P. Rybski, B. Salesky, Y.-W. Seo, S. Singh, J. Snider, A. Stentz, W. R. Whittaker, Z. Wolkowicki, J. Ziglar, H. Bae, T. Brown, D. Demitrish, B. Litkouhi, J. Nickolaou, V. Sadekar, W. Zhang, J. Struble, M. Taylor, M. Darms, and D. Ferguson. Autonomous driving in urban environments: Boss and the urban challenge. J. Field Robot., 25(8):425–466, Aug. 2008.
- [28] H. Wang, J. Kearney, and K. Atkinson. Arc-length parameterized spline curves for realtime simulation. In in Proc. 5th International Conference on Curves and Surfaces, pages 387–396, 2002.

APPENDIX A

Ancillary algorithms

```
function subdivide(a, b, \bar{s}) begin
 2
               // If a and b have gotten too close we return to prevent infinite loops
 3
              if |a-b| < \epsilon then return -1
 4
               // Find the midpoint of the segment [a, b]
 5
               mid \leftarrow (a+b)/2
               // Find the corresponding possible position
 6
 7
               \tilde{s} \leftarrow arclength(0, mid)
               // If \tilde{s} is very close to \bar{s}, we've found the solution at mid
 8
 9
              if |\tilde{s} - \bar{s} < \epsilon| then return mid
10
               // Otherwise we recurse on either the left or right subinterval
11
              if \tilde{s} > \bar{s} return subdivide(a, mid)
12
               else return subdivide(mid, b)
13
          end
```

FIGURE A.1. The recursive subdivision algorithm used to find the position of the point with arclength \bar{s} along a segment of a spline. a and b is the range of the search (initially, 0 and 1), ϵ is a small number and arclength(c,d) is a function that uses Equation (3.4) to find the distance along the spline from c to d.

```
1
           function point_in_poly(x, y, xs, ys, length) begin
 2
                 c \leftarrow \mathsf{false}
 3
                 i \leftarrow 0
 4
                 j \leftarrow length-1
 5
                 while i < length
 6
                       a \leftarrow ys[i] > y \neq ys[j] > y
                       b \leftarrow x < (xs[j] - xs[i]) * (y - ys[i]) / (ys[j] - ys[i]) + xs[i]
 8
                      if a and b
 9
                            c \leftarrow !c
10
                      endif
11
                      j \leftarrow i
                      i \ \leftarrow i{+}1
12
13
                 endw
14
                 return c
15
           end
```

FIGURE A.2. Given a point (x,y) and length points with x-coordinates xs and y-coordinates ys which describe a polygon, returns true if the point is contained within the polygon and false otherwise.

APPENDIX B

Tables and values

Name	Description	Value
$\overline{\tau}$	Tension param for Cardinal splines	0.1
A_{max}	Maximum positive acceleration	0.5 m/s^2
Br_{max}	Maximum braking acceleration	-1.4 m/s^2
ϕ_{max}	Maximum steering angle	0.5
ϕ_{min}	Minimum steering angle	-0.5
ϕ_ϵ	Discretization of ϕ	0.02
v_{max}	Maximum velocity	4 m/s unless noted
D_s	Sampling distance	7 m

Table B.1. Parameter values used for the demonstrations in Chapter 5.

Overall acceleration	Consequence
$a_w < 0.315 \text{ m/s}^2$	Not uncomfortable
$0.315 < a_w < 0.63 \text{ m/s}^2$	A little uncomfortable
$0.5 < a_w < 1 \text{ m/s}^2$	Fairy uncomfortable
$0.8 < a_w < 1.6 \text{ m/s}^2$	Uncomfortable
$1.25 < a_w < 2.5 \text{ m/s}^2$	Very uncomfortable
$a_w > 2.5 \text{ m/s}^2$	Extremely uncomfortable

TABLE B.2. Human perceptions of acceleration as specified by the ISO 2631-1 standard, where a_w is the overall root mean squared acceleration.