Localizing, Modeling, and Drifting an Autonomous RACECAR

by

Corey H. Walsh

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2018

© Massachusetts Institute of Technology 2018. All rights reserved.

Author ........................................................................................................

Department of Electrical Engineering and Computer Science

May 29, 2018

Certified by ..............................................................................................

Dr. Sertac Karaman
Associate Professor
Thesis Supervisor

Accepted by ............................................................................................

Dr. Katrina LaCurts
Chairman, Masters of Engineering Thesis Committee
Localizing, Modeling, and Drifting an Autonomous RACECAR

by

Corey H. Walsh

Submitted to the Department of Electrical Engineering and Computer Science on May 29, 2018, in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, I discuss several topics which are core to the development of car-like autonomous systems, using the small scale autonomous RACECAR platform as a testbed. The ultimate goal of this work was to create an autonomously drifting car, which has been achieved with nonlinear trajectory optimization, and time-varying LQR trajectory tracking.

Topics include a discussion of the RACECAR platform, dynamics modeling of car-like systems, robot localization, autonomous RACECAR control, and the design of a custom RACECAR simulator. In developing a high performance particle filter localization algorithm, I designed and implemented a novel algorithm for high performance two-dimensional ray casting in occupancy grid maps, called the Compressed Directional Distance Transform (CDDT). Here, I will discuss the theory and characteristics of the CDDT algorithm, which has been implemented in RangeLibc, my open-source software library.

Thesis Supervisor: Dr. Sertac Karaman
Title: Associate Professor
Acknowledgments

First, I would like to thank Professor Sertac Karaman for advising me over the last two years. He gave me the resources and support that I needed to chase the problems that I found interesting, and encouraged me to push through the difficulties.

Next, I would like to thank my friends, particularly Austin, Yanni, and Tina, for looking out for me, and occasionally laughing my terrible puns. I especially would like to thank Hannah for her loving support, and for somehow never getting sick of hearing about my endeavors. Naturally, I’d like to thank my cats Asami and Jinkx, for being so soft and fluffy.

I would like to extend thanks and acknowledgment to the many people at MIT who have helped me throughout my thesis, including Ezra, Reo, Samir, Trevor, Martina, and Winter. I have also had some excellent professors along the way, who have given me the tools that I needed to pursue my goals. Without all of their help, my RACECAR would not be able to do sick tricks.

Last, but certainly not least, I would like to thank my family, and my parents, Ken and Tobi Walsh, for showing me an interest in engineering, and for helping turn my crazy ideas into reality.
# Contents

1 Introduction .................................................. 19
   1.1 Motivations .............................................. 19
   1.2 CDDT ..................................................... 20
   1.3 Outline .................................................. 20

2 Meet the RACECAR ............................................ 21
   2.1 Stock Configuration ..................................... 22
   2.2 Research Configuration .................................. 23
   2.3 Basic Software Architecture .............................. 24

3 Modeling the RACECAR ........................................ 25
   3.1 Coordinate Frames and other Notational Considerations ........ 25
   3.2 A Simple Geometric Steering Model ....................... 27
   3.3 A Car Model with Wheel Slip ................................
       3.3.1 Car-like Kinematics ............................... 29
       3.3.2 Modeling Wheel Forces ............................ 30
       3.3.3 System Identification ............................... 32

4 State Estimation ............................................... 37
   4.1 A Particle Filter Algorithm for State Estimation .......... 37
       4.1.1 Notation ........................................... 38
       4.1.2 Mathematical Formulation ......................... 39
       4.1.3 The Particle Filter Algorithm .................... 42
4.1.4 Our Implementation ........................................ 43
4.2 Motion Capture .................................................. 44

5 High Performance Ray Casting for Particle Filtering 45
  5.1 Problem Formulation and Notation ............................ 45
  5.2 Existing Algorithms for 2D Ray Casting ....................... 47
  5.3 The Compressed Directional Distance Transform .............. 47
    5.3.1 Direct CDDT Construction Algorithm .................... 50
    5.3.2 CDDT Query Algorithm ................................. 51
    5.3.3 Incremental CDDT Update ............................... 52
    5.3.4 Optimizations ........................................... 53
    5.3.5 Theoretical Analysis .................................... 56
    5.3.6 Experiments and Results ............................... 57

6 Controlling the RACECAR 63
  6.1 Trajectory Search and Pure Pursuit Tracking .................. 63
    6.1.1 Path Planning ......................................... 64
    6.1.2 Pure Pursuit Trajectory Tracking ....................... 65
    6.1.3 Variable Speed Pure Pursuit ........................... 66
  6.2 Autonomous Drifting with LQR and Trajectory Optimization .... 70
    6.2.1 Nonlinear Trajectory Optimization ....................... 71
    6.2.2 LQR Based Trajectory Tracking and Stabilization ....... 72
    6.2.3 Implementation and Results ........................... 74
  6.3 Parallel parking the RACECAR with Dynamic Programming ...... 81
    6.3.1 The Value Iteration Algorithm ......................... 82
    6.3.2 Problem Formulation ................................. 84
    6.3.3 Implementation and Results ........................... 87

7 Simulating the RACECAR 91
  7.1 Pose Simulation ............................................... 93
  7.2 Laser Scanner Simulation .................................... 93
List of Figures

2-1 A photograph of the V2 RACECAR configuration. .................. 21
2-2 A photograph of the V3 RACECAR configuration. .................. 22
2-3 A photograph of the research RACECAR configuration on two scales, used to determine the center of mass. ......................... 23

3-1 Diagram of an idealized RACECAR (left) with the corresponding bicycle based simplification (right). Here, the pictured steering angle $\delta$ is considered to be negative. $l_a$ and $l_b$ are the distances between the car’s center of mass and front and rear tires, respectively. ..................... 26
3-2 Illustration of the path arcs traveled for a variety of different steering angles, as predicted by the geometric Ackermann model. ............. 28
3-3 Illustration of a wheel’s slip angle. Image from [1]. ..................... 30
3-4 An image of the research car with its wheels centered on two protractors. This setup was used in the identification of the steering servo setpoint to steering angle gain parameter. We swept through the range of steering setpoints while marking the angle of each wheel. To account for nonlinearities in steering gain introduced by the Ackermann steering geometry of the Traxxas chassis, we use the average of the two measured steering angles in determining the steering gain. ............... 33
3-5 Scatter plot demonstrating observed front tire slip angles versus $F_{f,\text{lat}}$. Best fit lines demonstrated for both parameterized tire force models considered (Pacejka and Arctan). ................................. 34
3-6 Scatter plot demonstrating observed rear tire slip angles versus $F_{r,lat}$.
Best fit lines demonstrated for both parameterized tire force models considered (Pacejka and Arctan). ........................................... 35

3-7 Scatter plot demonstrating observed rear tire slip ratio versus $F_{r,long}$.
Best fit lines demonstrated for both parameterized tire force models considered (Pacejka and Arctan). ................................. 35

4-1 Illustration of a high level view of particle filter localization. The known map, laser scans, and odometry information is fused into an estimated pose via the MCL algorithm. ........................................... 38

4-2 Illustration of the LiDAR sensor model used in our particle filter. The probability that any given distance is measured, given that the true distance to the nearest obstacle is 140px, is shown on the left. On the right, we demonstrate the same probability metric, for any possible ground truth distance. ........................................... 41

5-1 Occupancy grid map intrinsic coordinate system. © 2018 IEEE .......... 46

5-2 Illustration of ray casting. The query and collision points are shown in red and blue, respectively. © 2018 IEEE ................................. 46

5-3 Visualization of the (unpruned) CDDT data structure for the basement map with $\theta_{\text{discretization}} = 16$. Each row demonstrates the set of zero points for a particular map rotation. Darkness represents the number of elements in each CDDT bin - white areas have no elements, black areas have several hundred elements. Due to radial symmetry (see Section 5.3.4), only $\theta$ values between 0 and $\pi$ are shown. ................................. 47
5-4 Demonstration of the sawtooth response along the $\theta = 0$ direction. Source occupancy grid pictured on the left. The distance between each pixel and the nearest obstacle in the $\theta = 0$ (rightward) direction is demonstrated in image form on the top right. The row of the distance image for $y = 400$ (blue line in the distance image) is plotted in the bottom right, demonstrating the sawtooth signal. This signal, corresponding to an entire row in the lookup table, may be compressed simply by preserving the zero points at $x = 447, 517, 900, 989$, with a compression factor in the hundreds.

5-5 Visualization of the radial symmetry present in CDDT zero point extraction. On the left, we show a visualization of the same CDDT structure as in Fig. 5-3 but without exploiting radial symmetry. On the right, we show the CDDT slices corresponding to the angles $\pi$ through $\pi + 2.75$ flipped horizontally. Notice that this flipped version (highlighted in red) is an exact copy of the lower half (highlighted in blue). This redundancy is exploited by storing only the CDDT slices highlighted in blue, and inferring the red CDDT slices at query time.

5-6 Demonstration of reconstructing rows of two DDT slices from a single set of zero points. © 2018 IEEE

5-7 Function for generating morphological edge maps. The edge map is the difference between the original map, and an eroded map.

5-8 Example map left, edge map right. © 2018 IEEE
Example of a map which would result in many elements being projected into the same CDDT bin, even with edge-map preprocessing. In this case, 21 elements will be projected into the same CDDT bin for $\theta = 0$, but only two of those elements (the two black pixels) can possibly be used in a ray cast query with $\theta = 0$. CDDT pruning would help significantly in this case, reducing the number of elements in each CDDT bin to two (corresponding to the two black pixels). Cases similar to this one occur frequently in real maps, since walls often create continuous lines in real occupancy grid maps.

The two maps used for synthetic benchmark testing. Basement map shown on the left, synthetic map shown on the right.

Violin plots demonstrating histogram of completion time over a large number of queries for each ray cast method. The horizontal axis shows time in milliseconds, and vertical axis shows the number of queries that completed after exactly that amount of time. The left side demonstrates the observed performance with a random query sampling scheme. The right side demonstrates performance with a grid sampling scheme. Grid sampling appears to be more efficient for all methods, likely due to memory access pattern related caching effects. © 2018 IEEE.

Construction time and memory footprint of each method. Ranges stored in lookup table with 16 bit integers, initialized with ray marching. © 2018 IEEE.

Time required by the MCL update step of the particle filter algorithm with different ray casting methods over a two minute trial on the RACECAR (Right). In all cases, the particle filter maintains 1000 particles, resolving 61 ray casts per particle. $\theta_{\text{discretization}} = 120$ where applicable. Real-world position of the car at times marked P0-P5 shown in the map image to the left. © 2018 IEEE.
5-14 Maximum number of particles that can be maintained in real time (approx. 40Hz) on the NVIDIA Jetson TX1. Stata basement map, 61 ray casts per particle, \( \theta_{\text{discretization}} = 120 \) where applicable. © 2018 IEEE.

5-15 The effect of the \( \theta_{\text{discretization}} \) parameter on median positional and rotational localization error (top), and the number of particle filter divergences (bottom) during a five minute period of driving in a motion capture system. All other parameters are held constant. Notice that a \( \theta_{\text{discretization}} \) above around 100 bins results in error characteristics similar to the ray marching algorithm (which does not have an angular discretization). © 2018 IEEE.

6-1 Example preplanned trajectory overlaid on the MIT Stata Basement Loop occupancy grid map. The trajectory start point is depicted as a green sphere, and the end point is depicted as a red sphere. This particular trajectory was manually determined for use in a race controller.

6-2 Example trajectory with speed profile. Path is shown on the right, with circles of varying radius indicating planned traversal speed. Red circles indicates deceleration. Green circles indicates acceleration or constant speed. The maximum speed profile based on path curvature is shown on the upper left. The maximum speed profile based on both path curvature and acceleration bounds is shown on the bottom left. In this example, 5.0 m s\(^{-2}\) is chosen as a global speed limit to avoid excessively high speeds.

6-3 3D Scatter Plot relating tangential velocity, steering arc radius, and steering angle. Gathered with a V1 RACECAR, using a GoPro camera for data collection and custom video processing scripts to extract the ground truth velocity and steering arc radius corresponding to each steering angle.
6-4 Mapping between steering angle and maximum traversal speed. Based on the data from Fig. 6-3, and used in automatically generating maximum speed profiles for given trajectories.

6-5 Several different poses observed along the "straight" maneuver with the tvLQR controller.

6-6 Observed RACECAR trajectories with the straight trajectory over dirt. Multiple trials with initial state perturbations are demonstrated for the tvLQR controller. The small arrows indicate the car's start and end orientation. Notice that the system is able to nearly recover from even large initial state errors, despite only 1.56 seconds of activity. The open-loop controller makes no attempt at tracking error recovery. The mean terminal position error of the observed open-loop trials was found to be 0.36m, whereas for tvLQR the same statistic was found to be 0.2m, including trials with highly perturbed initial states. If one only considers tvLQR trials with small initial perturbations, this error is found to be 0.12m, representing a significant reduction as compared to open-loop.

6-7 Several different poses observed along the drifting park maneuver with the tvLQR controller. The rear end of the car begins sliding near the second half of the trajectory (e.g. the last four poses shown here). Notice the dirt on the ground, which reduces tire grip and increases the surface variation.
6-8 Observed RACECAR trajectories with the drifting park trajectory over dirt. Multiple trials with initial state perturbations are demonstrated for the tvLQR controller. In this example, the open-loop controller fails entirely, missing the intended parking spot by a large margin. This failure is largely due to the dirt induced loss of traction, and other model inaccuracies. On the other hand, the tvLQR controller is able to fairly reliably track the intended path, even in the case of significant initial state perturbations. We note that the car does not exactly reach the end of the trajectory. We primarily attribute this to systematic model error.

6-9 Several different poses observed along the "donut" maneuver with the tvLQR controller.

6-10 Observed RACECAR trajectories with the donut trajectory (without dirt on the ground). The open-loop controller fails to track the planned trajectory, over-rotating by roughly $1.36\pi$ radians and translating away from the origin. In contrast, the tvLQR controller is able to much more accurately track the planned path, controlling for excessive oversteer by counter-steering. We note that the car does not exactly reach the end of the trajectory. We primarily attribute this to systematic model error.

6-11 Parked RACECAR.

6-12 Motion primitives considered in the parallel parking motion model. Each line represents one action $\left(\delta, \psi\right)$. The radii of these arcs are determined via the empirically determined mapping presented in Fig. 6-13. Timestep duration, and thus the distance traveled along these path arcs, is a parameter of the value iteration algorithm presented here.

6-13 Empirically determined mapping between steering angle and path arc radius. The mapping implied by the geometric Ackermann model (as discussed in Section 3.2) is also pictured here for comparison.
6-14 The full 210x210 (5cm resolution) "East Campus kitchen" map used in testing (left). In this map, the green rectangle represents the parking goal region. On the right, several different slices ($\theta = 0^\circ, 90^\circ, 180^\circ, 270^\circ$) of the converged value function $J$ are demonstrated. Lighter colors represent higher cost, dark regions are low cost. The parking spot is black here, since it is the only zero cost region.  

6-15 A demonstration of real paths traversed by the RACECAR during operation. Black regions in the map represent occupied space, white regions are free space. Picture of the RACECAR represents the initial pose of the car. Green region is the goal position. Gray areas represent the region of space swept by the footprint of the car during traversal from the initial pose to the goal.  

7-1 Screen capture of the RViz while the custom RACECAR simulator is running. Pose history is shown with the red arrows. The white dots overlaid on the map are from the simulated laser scanner.  

8-1 Spring 2018 6.141 Class photo.  
8-2 Spring 2017 6.141 Class photo.  
8-3 Jan. 2016 IAP competition group photo. I am holding the rightmost car.  

9-1 Example Flame Graph of the particle filter developed here.
Chapter 1

Introduction

This thesis investigates the construction, modeling, state estimation, and control of a small-scale autonomous RACECAR.

1.1 Motivations

Since their introduction in the 18th century, cars have been developed and improved in terms of usability, safety, efficiency, performance, and more recently, autonomy. Today, cars are a core enabling factor in our cities, lifestyles, and livelihoods.

Unfortunately, this modern luxury comes at a cost, both in terms of human life, and environmental impact. Despite significant regulatory and technological improvements, which have had success in reducing vehicle related deaths since the mid twentieth century, cars are still a leading cause of death and injury world wide. Currently, approximately 270 million automobiles contribute to on average over 100 vehicle related deaths per day in the United States.

With the recent development and improvement in computer systems, advanced automation techniques can now be applied to the problem of increasing automobile safety and performance. In pursuit of this goal, the study and development of robotics techniques as they relate to car-like systems is of vital importance.

In this thesis, I investigate and develop the 1/10th scale RACECAR platform. Not only is this an advanced platform for car-like robotics research, it is also a powerful teaching tool currently being used to educate the next generation of engineers at MIT
and other academic institutions. Throughout this thesis, many of my endeavors were influenced not only be applicability to research interests, but also applicability to RACECAR-based educational programs.

1.2 CDDT

This thesis work introduces a new data structure for two-dimensional ray casting in occupancy grid maps, called the Compressed Directional Distance Transform, or simply CDDT. The CDDT data structure is designed with planar robot localization in mind, and exhibits fast queries, fast initialization, low memory usage, and the ability to perform incremental map updates. I have developed a C++ and Python library called RangeLibc\(^1\), which implements CDDT, as well as several other competing ray casting methods for the CPU and GPU. This algorithm was recently published as part of the 2018 IEEE International Conference on Robotics and Automation (ICRA) proceedings [2]. I include a discussion of CDDT here for completeness, though my paper on the subject provides a more rigorous treatment.

1.3 Outline

This thesis is organized as follows:

- Chapter 2 introduces the robotic RACECAR platform
- Chapter 3 presents two different RACECAR dynamics models
- Chapter 4 discusses methods for RACECAR state estimation
- Chapter 5 presents and evaluates a novel high performance ray casting algorithm developed in this thesis work
- Chapter 6 discusses several of the different methods of RACECAR control
- Chapter 7 introduces a RACECAR simulator
- Chapter 8 reviews a few educational programs based on the RACECAR
- Chapter 9 includes supplemental material
- Chapter 10 concludes this thesis with a review of the methods considered, and presents areas for future research

\(^1\)https://github.com/kctess5/range_libc
Chapter 2

Meet the RACECAR

Figure 2-1: A photograph of the V2 RACECAR configuration.

The RACECAR\textsuperscript{1} (i.e. Rapid Autonomous Complex-Environment Competing Ackermann-steering Robot) is a an open source 1/10th scale autonomous car platform for robotics research and education. It is built on a Traxxas RC car chassis, and integrates powerful sensing and computing hardware. The RACECAR was jointly developed by the BeaverWorks of the Lincoln Laboratory, the Department of Aeronautics and Astronautics, and the Laboratory for Information and Decisions Systems at the Massachusetts Institute of Technology.

\textsuperscript{1}http://racecar.mit.edu
2.1 Stock Configuration

Figure 2-2: A photograph of the V3 RACECAR configuration.

The stock RACECAR (as used in various educational programs) has undergone three revisions (V1, V2, and V3) over the course of this thesis. The first and second revisions were very similar, with the exception of a newer version of the motor controller and upgraded suspension. The second revision (pictured in Fig. 2-1) includes the following parts:

- Traxxas Slash 4X4 Platinum Edition chassis
- NVIDIA Jetson TX1 embedded computer
- Hokuyo UST-10LX 2D LiDAR
- Stereolabs ZED depth camera
- Vedder VESC-X motor controller
- SparkFun 9DoF Razor IMU M0
- AC1750 WiFi Dual Band Gigabit Router
- Logitech F710 Bluetooth Joystick controller

The third revision (pictured in Fig. 2-3) shares much with V2, with the following upgraded parts:
• NVIDIA Jetson TX2 embedded computer
• Velodyne VLP-16 LiDAR
• Enertion FOCBOX motor controller

2.2 Research Configuration

![Image of RACECAR configuration](image)

Figure 2-3: A photograph of the research RACECAR configuration on two scales, used to determine the center of mass.

RACECAR number 74 (pictured in Fig. 2-3) was used in much of this thesis. The car started out as a V2 model, but was significantly upgraded for research purposes. Upgrades include:

• CASTLE 1415 Sensored 2400KV Drive Motor
• Hitec HSB-9360TH Steering Servo
• Racers Edge LP2S8000100C 8000mAh 7.4V Lipo Battery
• Netgear XR500 Router (offboard)
• DLink DAP-1665 Access points (used as onboard as WiFi to Ethernet adapter)
These hardware modifications served to improve steering and throttle response, as well as to significantly reduce communication latency between the motion capture system and the car. In addition to these hardware upgrades, several changes were made to the configuration, including:

- Significantly tightening suspension to minimize body roll and weight transfer effects
- Removal of the drive shaft, rendering the car rear-wheel drive only
- Pouring epoxy in the rear differential to lock the rear wheels together, simplifying vehicle dynamics and inducing an oversteer condition where the rear wheels are prone to losing traction.
- Modifying the layout to lower center of gravity and achieve a nearly 50/50 front-back weight distribution
- Placing several motion capture markers on the vehicle

### 2.3 Basic Software Architecture

The onboard NVIDIA Jetson computer runs Ubuntu 16.04. The RACECAR platform includes a software architecture based on the Robot Operating System (ROS) [3] for interfacing with the various onboard systems. The core software components are:

- Nodes for commanding the motor controller from the Jetson via USB interface
- Drivers and ROS wrappers for the onboard LiDAR, camera, inertial measurement unit (IMU), and joystick
- A control multiplexer to allow multiple nodes to send drive commands concurrently with different priorities (manual control, safety controller, autonomous controller)
- Transformation frames describing the spatial configuration of the car

Two core contributions of this thesis (discussed at length below) to the RACECAR software platform are:

- A high-performance particle filter localization node for performing LiDAR based state estimation in known environments (see Chapter 4)
- A custom RACECAR simulator ROS node, replacing the original Gazebo [4] based simulator (see Chapter 7)
Chapter 3

Modeling the RACECAR

When working with any robotic system, it is often useful to describe how the state of that system evolves over time under control. This model typically takes the form:

\[ \dot{x} = f(x, u) \]

Vector-valued \( x \) is used to describe the system state, while vector-valued \( u \) describes control inputs. In the case of the RACECAR, \( u \in \mathbb{R}^2 \) since we can command both steering angle and throttle setpoints.

Here, I introduce two models often used to describe RACECAR dynamics.

3.1 Coordinate Frames and other Notational Considerations

Common notation used in describing the car’s dynamics:

- \( \delta \): steering angle setpoint in radians
- \( \psi \): throttle setpoint. Typically the units are in m/s, specifically the speed at which the rear tires would be moving relative to the body frame at the contact point between the tire and the wheel. This definition is convenient because it means that if the car is driving with constant throttle setpoint of 1, under no-slip conditions, the car would have a tangential velocity of 1 m/s.
Figure 3-1: Diagram of an idealized RACECAR (left) with the corresponding bicycle based simplification (right). Here, the pictured steering angle $\delta$ is considered to be negative. $l_a$ and $l_b$ are the distances between the car’s center of mass and front and rear tires, respectively.

It is useful to define a few separate coordinate frames for use in modeling and discussion. The first frame of interest is the world frame, which is the top level frame of interest. Typically in this work, the world frame is considered to be the coordinate system of the known map, and is stationary. The subscript "w" (e.g. $x_w$) will be used to denote when a spatial quantity is in the world frame.

The next frame is the car’s body frame. This frame is fixed with respect to the car. Its origin varies depending on the model in question, but is typically either the car’s center of mass (as demonstrated in 3-1), or the "base link" pose, which is located directly between the rear tires. We always consider the positive $x$ axis of the car’s body frame to be directly forwards of the car, such that if the car starts at the origin and drives straight, it accumulates a positive $x$ position. The positive $y$ axis in the body frame is always considered as directly leftwards from the car. The subscript "b" (e.g. $x_b$) will be used to denote when a spatial quantity is in the car’s body frame.
The final set of coordinate frames which are often useful in modeling car dynamics is the various wheel frames, centered on each tire. Since we typically consider the bicycle simplification, we usually consider only the frames of the front and rear tires in the bicycle model (we don’t consider left/right wheel frames). The rear tire frame is similar to the body frame, with the exception of a translation. The front tire’s frame is also translated with respect to the body frame, but it is also rotated by exactly the steering angle \( \delta \) radians. The subscript "fw" (e.g. \( x_{fw} \)) will be used to denote when a spatial quantity is in the front tire’s frame. Similarly, the subscript "rw" (e.g. \( x_{rw} \)) will be used to denote when a spatial quantity is in the rear tire’s frame.

### 3.2 A Simple Geometric Steering Model

One widely used model for car-like dynamics is the geometric Ackermann model [5], which arises from considering the kinematics of a bicycle under zero-slip wheel conditions. While this model is not physically accurate, it does capture the primary nonlinearities and nonholonomic constraints in the system and is useful in many (especially low-speed) contexts. Here, we consider the car’s \( x \) and \( y \) position to be specifically that of the base link point (directly between the rear tires).

\[
x = \begin{bmatrix} x_w \\ y_w \\ \theta_w \end{bmatrix} = \begin{bmatrix} \text{world base link x-position} \\ \text{world base link y-position} \\ \text{world orientation} \end{bmatrix}
\]

\[
x = \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \\ \tan(\frac{\delta}{l_a + l_b}}) \end{bmatrix} \times v = \begin{bmatrix} \text{world base link x-velocity} \\ \text{world base link y-velocity} \\ \text{yaw rate} \end{bmatrix}
\]

Here, \( v \) is the velocity of the car. In this case, since we are assuming that wheel slip is negligible, we assume that the velocity \( v \) is equal to the throttle setpoint \( \psi \). While in general, this is not a good assumption, it does keep the size of our state space to a minimum. The overall wheelbase length of the car is denoted \( l_a + l_b \).

Numerical integration of the above dynamics equations reveals that with constant
steering angle, the car travels along constant-curvature arcs tangent to the car, with some radius depending on steering angle (illustrated in Fig. 3-2). Often, it is useful to consider these "path arcs" directly (such as in Section 6.1.2) so that larger dynamics integration steps may be used. We note that the arc radius is defined as follows for a given steering angle $\delta$.

$$\text{path arc radius} = \frac{l_a + l_b}{\tan \delta}$$

### 3.3 A Car Model with Wheel Slip

Unfortunately, we cannot always assume that wheel slip is negligible. In conditions involving high speed, low traction, aggressive driving, or a combination thereof, the above discussed geometric model of car-like dynamics breaks down significantly. To cope with this, I consider a model of the RACECAR which incorporates slip dynamics.

Here, we model wheel-ground interactions as producing forces, and work through the kinematic effect of those forces. Throughout the derivation of this model, we will use the knowledge that the research version of the RACECAR is rear-wheel drive, with a locked rear differential. Notably, this fact arises in the longitudinal forces contributed by each tire, namely in our assumption that the front tires produce no longitudinal force (in the wheel frame).

This model is based on the skidding car model from Chapter 13 of [6], adapted to include lateral velocity. Thanks to Ezra A. Tal for help deriving this model.
3.3.1 Car-like Kinematics

Assume that, in each wheels’ coordinate frame:

- The rear tires produce some longitudinal force $F_{r,\text{long}}$ and lateral force $F_{r,\text{lat}}$ due to friction with the ground.
- The front tires produce some lateral force $F_{f,\text{lat}}$ due to friction, but do not produce a longitudinal force since they are not driven and spin freely (which is to say $F_{f,\text{long}} = 0$).
- The car has dimensions and mass as described in the bicycle model in Fig. 3-1.

Under these assumptions, we can derive a set of kinematic equations that describe the car’s behavior. First, we define a state space which can capture the important system dynamics. This state space includes the car’s position in space in the world frame, forwards and lateral velocities in the body frame, and the rear wheel speed. We omit actual steering angle because the research car has a powerful servo that can very quickly rectify the actual steering angle with the steering setpoint; also, we are unable to measure the real steering angle due to lack of wheel angle encoders. The rear wheel speed could potentially be omitted if one assumes that they can instantaneously command some wheel speed. In practice, we include wheel speed since large wheel speed change requires significant time and control effort, and we can directly measure the actual wheel speed via the drive motor hall sensor.

$$x = \begin{bmatrix} x_w,\text{cm} \\ y_w,\text{cm} \\ \theta_w \\ \dot{x}_b \\ \dot{y}_b \\ \dot{\theta}_w \\ v_{rw} \end{bmatrix} = \begin{bmatrix} \text{world x-position} \\ \text{world y-position} \\ \text{world orientation} \\ \text{forward velocity} \\ \text{lateral velocity} \\ \text{yaw rate} \\ \text{rear wheel velocity} \end{bmatrix}$$

Now, we may use kinematics to determine the derivatives of this state as a function of the current state, and the commanded controls: $\dot{x} = f(x, u)$. 
\[
\begin{bmatrix}
\dot{x}_b \cos(\theta_w) - \dot{y}_b \sin(\theta_w) \\
\dot{x}_b \sin(\theta_w) + \dot{y}_b \cos(\theta_w) \\
\dot{\theta}_w \\
\dot{y}_b \theta_w - F_{f,\text{lat}} \sin(\delta)/m + F_{r,\text{long}}/m \\
-\dot{x}_b \theta_w + F_{f,\text{lat}} \cos(\delta)/m + F_{r,\text{lat}}/m \\
(l_a \dot{F}_{f,\text{lat}} \cos(\delta) - l_b \dot{F}_{r,\text{lat}})/I \\
p_{\text{throttle}} \left( u_{\text{throttle}} - s_{rw} \right)
\end{bmatrix} = 
\begin{bmatrix}
\text{world x-velocity} \\
\text{world y-velocity} \\
\text{yaw rate} \\
\text{forward acceleration} \\
\text{lateral acceleration} \\
\text{angular acceleration} \\
\text{change in rear wheel speed}
\end{bmatrix}
\]

The first three velocity terms arise simply from a change of coordinate system between the world and body frames. The next three acceleration terms arise from considering conservation of momentum, and the contribution of the forces from both the front and rear tires (including a change of coordinate system from the front wheel frame into the body frame). The final term comes from the assumption that the change in rear wheel speed may be modeled as a proportional controller. This assumption is not strictly true, but it is reasonable since the low-level motor controller implements a proportional–integral–derivative (PID) controller. Modeling the integral and derivative terms of the low-level motor controller would require a larger state space, which we do not believe is worth the additional accuracy for our purposes.

### 3.3.2 Modeling Wheel Forces

![Illustration of a wheel’s slip angle. Image from [1].](image-url)
Whereas in the previous section (3.3.1), we could simply derive the car's dynamics based on simple kinematics, it is not so easy to determine the forces that each tire produces due to friction with the ground. These forces are highly nonlinear, and difficult to estimate.

One commonly considered model of tire forces [7] involves computing tire forces as a function of slip angle and slip ratio. Slip angle ($SA$) is the angle between a wheel's direction of travel and orientation. A slip angle of zero means that the wheel is aligned with the direction of travel. Slip ratio ($SR$) is the ratio between the speed of the tire's motion at the tire’s tread ($V_{\text{tire}}$), and the relative speed of the road longitudinal to the tire at the point of contact between road and tire ($V_{\text{road}}$). A slip ratio of zero means that the wheel is rolling at the same rate as the surface is moving underneath it. A positive slip ratio means that the tire is spinning faster than the road, and a negative slip ratio means that the tire is spinning slower than it travels over the road. Nonzero slip ratios or slip angles implies nonzero forces in the directions longitudinal or lateral to each tire, respectively.

Typically, slip ratio is defined as $SR = \frac{V_{\text{tire}} - V_{\text{road}}}{|V_{\text{road}}|}$, however, this definition becomes numerically unstable at low speeds, and is non-differentiable. To cope with these problems, we consider the following definition of slip ratio using a "soft" absolute value function $|x|_\epsilon$. The value of $\epsilon$ is largely arbitrary, but should be as small as possible without introducing numerical instability in the simulation or control systems.

\[
|x|_\epsilon = \sqrt{x^2 + \epsilon}
\]

\[
SR = \frac{V_{\text{tire}} - V_{\text{road}}}{|V_{\text{road}}|_\epsilon=0.05}
\]

Similarly, slip angle is typically considered as $SA = \tan^{-1}(\frac{V_{\text{lat},\text{tire}}}{|V_{\text{long},\text{tire}}|})$ where $V_{\text{lat},\text{tire}}$ and $V_{\text{long},\text{tire}}$ are the lateral and longitudinal velocities of the ground's motion in each tire's frame. Notice that a free rolling wheel has $V_{\text{lat},\text{tire}} = 0$, and thus $SA = 0$. This definition suffers from the same non-differentiability problems as does the standard definition of slip ratio. Thus, we once again consider the slightly modified form:
\[ SA = \tan^{-1}\left( \frac{V_{\text{lat,tire}}}{|V_{\text{long,tire}}|_{\epsilon=0.05}} \right) \]

Often, an equation called Pacejka’s Magic Formula \([7, 8]\) is used to relate the slip ratio/angle with forces generated. This formula is reproduced below. \(F\) represents a force, and \(x\) represents either a slip angle or ratio, depending on the use case (slip angle for lateral forces, slip ratio for longitudinal forces).

\[
F(x) = S_v + D \sin(C \tan^{-1}(B(x + S_h) - E(B \cdot (x + S_h) - \tan^{-1}(B(x + S_h)))))
\]

The use of this formula further reduces the problem of modeling wheel forces to determining the six constants \(\{B, C, D, E, S_v, S_h\}\) for each tire’s longitudinal and lateral directions. Since the research RACECAR is rear-wheel drive and we assume that \(F_{f,\text{long}} = 0\), there are 18 parameters to determine, six for each of \(\{F_{f,\text{lat}}, F_{r,\text{lat}}, F_{r,\text{long}}\}\). Similar logic would apply for either a front or all-wheel drive vehicle.

We also considered a simplified wheel model to address difficulties encountered in performing trajectory optimization with Pacejka’s Magic Formula, while still capturing the primary nonlinearities observed. The "Arctan" model is as follows, and requires four parameters \(\{A, B, C, D\}\) to be determined, rather than six.

\[
F(x) = A \ast \tan^{-1}(Bx + C) + D
\]

In some cases, we use a simple linear tire force model, or we consider multiple models at the same time, using different force models for different estimated forces (e.g. lateral vs longitudinal).

### 3.3.3 System Identification

The above defined model has several parameters which non-obvious values. The following parameters were directly measured.

- mass
- \(l_a\): distance from center of mass to front tires
- \(l_b\): distance from center of mass to rear tires
The following parameters were inferred in a fairly trivial manner:

- $I$: moment of inertia
- Throttle gain: conversion from drive motor revolutions per minute (RPM) to speed at the wheel contact point
- Steering gain: conversion from servo setpoint to real steering angle in radians

The remaining unknown parameters are those involved in wheel force computation. We used a motion capture system to directly estimate the forces generated by the RACECAR’s front and rear tires over the course of approximately twenty minutes of driving. By rearranging the above (Section 3.3.1) vehicle model, it is possible to express each of the three relevant forces ($F_{f,\text{lat}}, F_{r,\text{lat}}, F_{r,\text{long}}$) in terms of measurable quantities, as is demonstrated below. Additionally, using motion capture, we directly estimated the various slip ratios $SR$ and slip angles $SA$ which serve as inputs to the tire force model. These quantities allowed us to estimate wheel force parameters via least-squares fit with observed data (see Figures 3-5, 3-6, 3-7).
\[ F_{r,\text{long}} = m \ast \ddot{x} - m \ast \dot{y}_b \ast \dot{\theta} \quad \text{when } \delta = 0 \]

\[ F_{r,\text{lat}} = \frac{\dot{y}_b \ast m + \dot{x} \ast \dot{\theta} \ast m - I \ast \ddot{\theta}/l_a}{l_b/l_a + 1} \]

\[ F_{f,\text{lat}} = \frac{I \ast \ddot{\theta} + l_b \ast m \ast \dddot{y} + l_b \ast m \ast \dddot{x} \ast \dot{\theta}}{\cos(\delta) \ast (l_a + l_b)} \]

Figure 3-5: Scatter plot demonstrating observed front tire slip angles versus \( F_{f,\text{lat}} \). Best fit lines demonstrated for both parameterized tire force models considered (Pacejka and Arctan).

There are a few interesting observations to make about the motion-capture-based tire force data. Both lateral force scatter plots (Figures 3-5, 3-6) exhibit clusters of observations at large slip angle with less than typical forces. We observed that these clusters occur when the car enters a steady state "donut" maneuver where the rear tires are quickly and continuously sliding around the mostly stationary front tires. We believe this occurs due to unmodeled effects such as wheel deformation, vehicle suspension, weight transfer, or other inaccuracies arising from our bicycle model simplification.

Another interesting observation is that there are two noticeable trends in the rear
Figure 3-6: Scatter plot demonstrating observed rear tire slip angles versus $F_{r,lal}$. Best fit lines demonstrated for both parameterized tire force models considered (Pacejka and Arctan).

Figure 3-7: Scatter plot demonstrating observed rear tire slip ratio versus $F_{r,long}$. Best fit lines demonstrated for both parameterized tire force models considered (Pacejka and Arctan).
tire longitudinal force data (Fig. 3-7), with a slight vertical offset from each other. We determined that these two primary trends depend on the direction of the car’s motion. If the car is moving forwards, its trend appears to be shifted downwards by approximately 3 newtons as compared to the trend observed when the car is moving backwards. This effect is likely due to weight transfer effects between the front and rear tires due to acceleration, which slightly shifts loading on the rear tires.

Another note is that in all cases (Figures 3-5, 3-6, 3-7), many observations lie a considerable distance from the best fit curves. This error is some combination of natural stochasticity in road surface conditions, measurement error, and various unmodeled effects. Attempting to further improve the model to capture some of this unmodeled variation would most likely significantly increase the model complexity and/or state space size, and could render it less useful from a controls perspective.
Chapter 4

State Estimation

Determining a robot’s state in a known environment, also known as state estimation, is an important and challenging problem in the field of robotics. For humans, this ability comes naturally. Thanks to the most sophisticated computational structure known to humankind (the brain), and an extensive suite of biological sensors, humans have an impressive capacity for state estimation. In real time, and almost subconsciously, we are able to estimate both our own state, and the state of other objects in our vicinity. Unfortunately, for robots, this task is quite difficult.

For the RACECAR, the ability to estimate position and orientation in a known environment opens the doors to numerous higher-level planning and control techniques. In this work, I developed a fast LiDAR based particle filtering algorithm [9] for planar localization in a known occupancy grid map. Additionally, when onboard state estimation proved to be too computational and/or noisy for research purposes, I used an extremely accurate offboard motion capture system located in MIT’s building 31.

4.1 A Particle Filter Algorithm for State Estimation

One common solution to the robot localization problem in planar environments is the particle filter algorithm [9], which iteratively refines a set of pose hypotheses called particles. The general idea of the particle filter is to infer unobservable state variables in a system by recursively conditioning belief on directly observable system variables.
In our case, this means that we would like to infer the car’s (unobservable) position and orientation in a map, based on available sensor and control data.

While the particle filter is no longer considered state of the art in terms of robot localization, it does have some nice properties, such as the ability to capture non-linear system dynamics. Additionally, it is simple, easy to understand, and easy to implement. For these reasons, implementing a particle filter is the assignment in Lab 5 of MIT’s class 6.141. Much of this section is borrowed from the lab guide [10] for that assignment, which I developed in collaboration with other 6.141 staff members. The particle filter discussed here is currently used as the staff solution for that lab assignment.

In this section, I briefly cover the mathematical backing behind the particle filter algorithm. For a more in-depth treatment, see [11] or [9]; I use similar notation and formulation.

4.1.1 Notation

The following notational conventions are used in this section. Adapted from the 6.141 lab 5 guide [10].
• Time is denoted as $t$. $t = 0$ is the initial time, and $T$ is the most recent time.

• We refer to position, state, and pose interchangeably, referring to location and orientation $X = (x, y, \theta)$ in the map coordinate frame in all cases.

• State at a time $t$ is denoted $X_t = (x_t, y_t, \theta_t)$

• Action at time $t$ is denoted $a_t$, and represents state derivatives $\frac{\partial X_t}{\partial t} = (\frac{\partial x_t}{\partial t}, \frac{\partial y_t}{\partial t}, \frac{\partial \theta_t}{\partial t})$

• Observations at time $t$ are denoted $o_t$. In this case, it is composed of the $N$ range measurements $\{r_0, r_1, r_2, ..., r_{N-1}\}$ recorded by the LiDAR scanner onboard the RACECAR. Each of these measurements corresponds to a distance reading at a particular angle to the LiDAR’s frame. For the Hokuyo LiDAR onboard the V1 and research RACECAR, $N = 1081$.

4.1.2 Mathematical Formulation

The Monte Carlo Localization (MCL) algorithm [9] seeks to estimate the probability distribution of the current state over all possible states, conditioned on all available current and past data (sensor readings, control commands, etc), as shown in Eq. Eq. 4.1. Typically, in the pose tracking case, which we are most interested in, it is assumed that the initial state $X_0$ is known. In practice, we provide this initial state manually, but in general, it could be estimated via a global localization scheme.

$$\text{bel}(X_T) = P(X_T|a_0...T, o_0...T, X_0)$$

(Eq. 4.1)

Two key theorems are used in the derivation of the particle filter. First is the Markov assumption [12]: that future transition probabilities only depend on the previous state. This effectively states that if we know $X_t$, then state $X_{t+1}$, as well as any future observations, do not depend on any $X_k \forall k < t$. Another important theorem is Bayes’ rule [13]:

$$P(A|B) = \frac{P(B|A) * P(A)}{P(B)}$$
In the particle filter, we use both our sensor readings and odometry/control data to recursively update our belief distribution \( \text{bel}(X_T) \), starting from \( \text{bel}(X_0) \). Without loss of generality, we may assume that our sensor readings and odometry arrive in an alternating fashion.

\[
\text{bel}(X_T) = P(X_T|o_T, a_T, o_{T-1}, a_{T-1}, ..., X_0) \quad (\text{Eq. 4.2})
\]

Applying Bayes’ Rule to Eq. 4.2 yields the following, in which \( \eta \) is some normalizing constant that ensures the belief sums to one:

\[
\text{bel}(X_T) = \eta * P(o_T|X_T, a_T, o_{T-1}, a_{T-1}, ...) * P(X_T|a_T, o_{T-1}, a_{T-1}, ...) \quad (\text{Eq. 4.3})
\]

Now, we may apply our Markov assumption to Eq. 4.3 to remove the dependency of the first term on past observations.

\[
\text{bel}(X_T) = \eta * P(o_T|X_T) * P(X_T|a_T, o_{T-1}, a_{T-1}, ...) \quad (\text{Eq. 4.4})
\]

Invoking the Law of Total Probability [14] by integrating over the state space at time \( T - 1 \) reveals that:

\[
P(X_T|a_T, o_{T-1}, ...) = \int P(X_T|X_{T-1}, a_{T-1}, o_{T-2}, ...) * P(X_{T-1}|a_{T-1}, o_{T-2}, ...) \, dX_{T-1} \quad (\text{Eq. 4.5})
\]

Eq. 4.5 may be simplified greatly with two important observations. The first is that we may once again apply the Markov assumption to the left half of the integral. The second is that the right half of the integral is simply the belief distribution at the previous time step, \( \text{bel}(X_{T-1}) \).

\[
P(X_T|a_T, o_{T-1}, ...) = \int P(X_T|X_{T-1}, a_{T-1}) * \text{bel}(X_{T-1}) \, dX_{T-1} \quad (\text{Eq. 4.6})
\]
Substituting Eq. 4.6 into Eq. 4.4 yields the following recursive formula, which is the primary reason that the MCL algorithm is frequently referred to as a "recursive Bayes' filter."

\[
bel(X_T) = \eta \ast P(o_T|X_T) \ast \int P(X_T|X_{T-1}, a_{T-1}) \ast bel(X_{T-1}) \, dX_{T-1} \tag{Eq. 4.7}
\]

Eq. 4.7 contains two important terms that we must define. \(P(o_T|X_T)\) is known as the "sensor model," or "observation model," and it defines how likely it is to receive a particular observation given that the system is at state \(X_T\). \(P(X_T|X_{T-1}, a_{T-1})\) is called the "motion model," and it defines how likely the system is to be at state \(X_T\), assuming that the previous state \(X_{T-1}\) and action \(a_{T-1}\) are as given.

One commonly used [9, 11, 15] definition of the sensor model relies on a probabilistic model of range sensors, which defines the probability of reading any particular sensor measurement, given the true distance to an obstacle. This "beam" model typically includes terms for "short" measurements (e.g. an unmapped obstacle), "true" measurements, and "long" measurements where the sensor erroneously returned a maximum range measurement. Our sensor model is illustrated in Fig. 4-2. The use of the beam model relies on ray casting in order to determine predicted ground truth obstacle distances, given any pose in the map.
4.1.3 The Particle Filter Algorithm

When it comes to converting the mathematical basis of the MCL algorithm into code, there is some question as to how the belief distribution $\text{bel}(X)$ should be represented. Storing belief "exactly" is computationally infeasible since it involves storing a probability distribution over an infinite number of possible states. Even discretizing this space into an $K$-dimensional table, where $K$ is the dimension of our state space, is computationally challenging since this table would have to be quite large even for small values of $K$. Instead, the particle filter algorithm approximates the belief distribution as a set of $m$ weighted samples called "particles." MCL directly operates on this particle distribution to constantly maintain an up-to-date state estimate. Here, $w_T^i$ represents the probability weight of the corresponding hypothesis $X_T^i$.

$$\text{bel}(X_T) \approx \{X_T^i, w_T^i\} \forall 0 \leq i < m$$

In this section, as a notational convenience, we will refer to the particle distribution as $X_T$, which (more rigorously) represents the belief over the state space, $\text{bel}(X_T)$. Specific particles from the distribution will be referred to as $X_T^i$.

```
function MCL_update(X_{T-1}, a_{T-1}, o_T):
    X_T = {}
    for i in range(m):
        # Draw a pose from the old particle distribution, according to the old weights.
        # These samples implicitly represent the prior pose belief $\text{bel}(X_{T-1})$
        X_{T-1}^i \sim X_{T-1}
        # Apply the motion model to update the pose of an old particle
        # $\text{Ex}[]$ represents the expected value, so this is a maximum likelihood estimate
        X_T^i = \text{Ex}[X_T \mid X_{T-1}^i, a_{T-1}]
        # Weight the new particle according to the sensor model
        w_T^i = p(o_T \mid X_T^i)
        # Add the new pose and weight to the new distribution
        X_T = X_T \cup \{X_T^i, w_T^i\}
```
end for

\[ X_T = \text{normalize}(X_T) \]

return \( X_T \)

end function

function Particle_Filter():

\# Initialize the particle distribution based on known initial pose

\[ X = \text{bel}(X_0) \]

while true

\[ a_{T-1} = \text{get\_last\_odometry()} \]

\[ o_T = \text{get\_last\_sensor\_readings()} \]

\[ X = \text{MCL\_update}(X, a_{T-1}, o_T) \]

\# Inferred pose is the expected value of the belief distribution

\[ \text{pose} = \mathbb{E}[X] \]

end while

end function

A slight detail here is that the functions \text{get\_last\_odometry()} and \text{get\_last\_sensor\_readings()} must account the actual ordering of the received measurements. For example, if odometry is received at a higher rate than laser scan messages, multiple odometry messages should be combined via dead-reckoning.

4.1.4 Our Implementation

My implementation of the particle filter is available online\(^1\) as part of the RACECAR software stack. An illustrative video of our implementation in action is available online\(^2\) as well. The code is written in Python with ros\text{py}, Numpy, and RangeLibc. For performance, all particle operations are performed with vectorized Numpy, and sensor model evaluation (which relies on ray casting) is performed with RangeLibc\(^3\). Since most of the expensive computation is performed in this manner, the particle

---

\(^1\)http://github.com/mit-racecar/particle_filter

\(^2\)http://rayban.vision/#localization_video

\(^3\)https://github.com/kctess5/range_libc
filter demonstrates a high level of performance. It is able to maintain thousands of particles in real time, with an incoming laser scan data rate of 40Hz.

The particle filter implementation is designed to be simple, easy to understand, and efficient. The code is not designed to incorporate many of the common particle filter improvements such as KLD sampling [15] or global localization methods (such as in Mixture-MCL [11]), but rather to demonstrate the effectiveness of the standard MCL algorithm. Our code is comprised of less than 1000 lines of Python, including utility functions, visualization logic, comments, and other supporting code. The actual MCL update, sensor model, and motion model are comprised of only a few hundred lines of code.

Our particle filter has been used in practice by 27 student teams, over two iterations of MIT’s class 6.141, and has proven to be quite effective (although parameter tuning is sometimes required for best performance). Several research groups have also had success running this code, including MIT’s Distributed Robotics Laboratory, and University of Pennsylvania’s Real-Time and Embedded Systems Lab (MLAB).

4.2 Motion Capture

In some cases, particularly when derivatives of the RACECAR’s state are required, the above developed localization algorithms prove insufficient. In these cases, we use a camera-based motion capture system produced by OptiTrack\(^4\), located in MIT’s building 31. One notable area of where future research would be appropriate is in reducing, or at least characterizing, the reliance of the control approaches discussed here on precise state estimation.

The use of external motion capture poses some of its own engineering challenges. Notably, minimizing latency involved in transferring pose tracking data from the offboard capture system to the onboard control algorithms is often critical to high performance operation. To this end, we use high performance wireless network components, as discussed above in the RACECAR hardware section (Chapter 2).

\(^4\)https://optitrack.com
Chapter 5

High Performance Ray Casting for Particle Filtering

The above discussed implementation of the particle filter algorithm is highly dependent on a fast ray casting algorithm. Ray casting itself is a complex operation, often dependent on map occupancy or geometry, and an evaluation of the sensor model may require tens of ray casts. Many effective particle filters maintain thousands of particles, updating each particle tens of times per second. As a result, millions of ray cast operations may be resolved per second, posing a significant computational challenge for resource constrained systems.

In this thesis work, I developed a novel data structure for performing fast approximate ray casting, called the Compressed Directional Distance Transform, or CDDT. In this section, I will discuss the mechanics and characteristics of this data structure. For a more complete treatment of the discussion in this section, see my paper on the subject [2], which shares much of the same material presented here.

5.1 Problem Formulation and Notation

We define the problem of ray casting in occupancy grids as illustrated in Fig. 5-2. We assume a known occupancy grid map in which occupied cells have value 1, and unoccupied cells have value 0. Given a query pose \((x, y, \theta)_{\text{query}}\) in map space, the
ray cast operation finds the nearest occupied pixel \((x, y)_{\text{collide}}\) which lies on the ray starting at the position \((x, y)_{\text{query}}\) pointing in the \(\theta_{\text{query}}\) direction, and returns the Euclidean distance \(d_{\text{ray}}\) between \((x, y)_{\text{query}}\) and \((x, y)_{\text{collide}}\).

\[
d_{\text{ray}} = \left\| \begin{pmatrix} x \\ y \end{pmatrix}_{\text{query}} - \begin{pmatrix} x \\ y \end{pmatrix}_{\text{collide}} \right\|_2
\]

We denote the discretized query pose as \([ (x, y, \theta)_{\text{query}} ]\). A \(\theta\) slice through a lookup table (LUT) is a 2D subset of the full 3D LUT, in which the value of \(\theta\) is held constant and \(x, y\) are varied. The number of discrete \(\lceil \theta \rceil\) values is denoted \(\theta_{\text{discretization}}\). Fig. 5-1 demonstrates our chosen coordinate system. This choice of coordinate system arises primarily from historical reasons - the first version of our implementation used this system since it matches an intrinsic image pixel coordinate system [16]. Subsequent versions kept the same coordinate system for compatibility.
5.2 Existing Algorithms for 2D Ray Casting

Existing commonly used algorithms for ray casting in two-dimensional occupancy grid maps include Bresenham’s Line algorithm [17] and ray marching [18]. Both algorithms work by iteratively checking points in the map, starting at the query point, and moving in the ray direction until an obstacle is discovered.

One common strategy for accelerating the ray casting procedure in particle filters is to precompute a three-dimensional lookup table (LUT) which stores the expected ranges for each discrete \((x, y, \theta)\) state [9]. While this is simple to implement and does result in large speed improvements as compared to ray casting, it can be prohibitively memory intensive for large maps and/or resource constrained systems, as noted in [11]. In a 2000 by 2000 occupancy map, storing ranges for 200 discrete directions would require over 1.5GB. While this memory requirement may be acceptable in many cases, it scales with the area of the map - a 4000 by 4000 map would require over 6GB for the same angular discretization, which is larger than the random-access memory onboard many mobile robots.

5.3 The Compressed Directional Distance Transform

![Visualization of the CDDT Data Structure](image)

Figure 5-3: Visualization of the (unpruned) CDDT data structure for the basement map with \(\theta_{\text{discretization}} = 16\). Each row demonstrates the set of zero points for a particular map rotation. Darkness represents the number of elements in each CDDT bin - white areas have no elements, black areas have several hundred elements. Due to radial symmetry (see Section 5.3.4), only \(\theta\) values between 0 and \(\pi\) are shown.
In this thesis work, with the guidance of my thesis advisor, I developed a new data structure for 2D ray casting called the Compressed Directional Distance Transform (CDDT). This data structure was created with particle filter localization specifically in mind. As such, goals include fast initialization and query times, ability to quickly modify the map, and low memory usage. These goals are attained at the expense of angular precision, which as I will demonstrate, is acceptable for particle filtering.

Although the three-dimensional table used to store precomputed ray cast solutions in a discrete state space is inherently large, it is highly compressible, which can easily be seen by visual inspection of LUT slices (see Fig. 5-4). Our novel data structure is designed to compress this redundancy, while still allowing for fast queries in near-constant time. The CDDT algorithm can seen as a lossless map compression algorithm, which allows distance queries to be quickly resolved in compressed form.

The key realization which enables CDDT is as follows. In the slice of the three-dimensional LUT (described in 5.2) which corresponds to \( \theta = 0 \), each row forms a sawtooth signal, as demonstrated in Fig. 5-4. Any obstacle along the row results in a distance value of 0 for the overlapping LUT element (pixel). Additionally, each pixel in the row is one unit further from the nearest obstacle than its neighbor in the \( \theta = 0 \) direction, so its value simply increments that of its neighbor. In recognizing this pattern, one may losslessly encode each row by storing only the "zero points" where the signal drops to zero (see Fig. 5-4).

While a sawtooth pattern is present in the cardinal directions \( (\theta = 0, \frac{\pi}{2}, \pi, \frac{3\pi}{2}) \), the same cannot be said for arbitrary directions \( \theta \) in the 3D lookup table. When the ray cast direction does not align with one of the axes, discretization effects cause this pattern to break down. However, if we are willing to accept a small amount of interpolation error, this property may be restored via a simple trick. Rather than ray casting in the \( \theta \) direction, rotate the map by \(-\theta\), and then perform the ray cast in the \( \theta = 0 \) direction on the rotated map. It is easy to see that (allowing for small interpolation artifacts), these two operations are equivalent. However, the operation which includes a rotation preserves the sawtooth property for arbitrary ray casting directions. One may avoid adverse interpolation artifacts (such as introducing gaps
Figure 5-4: Demonstration of the sawtooth response along the $\theta = 0$ direction. Source occupancy grid pictured on the left. The distance between each pixel and the nearest obstacle in the $\theta = 0$ (rightward) direction is demonstrated in image form on the top right. The row of the distance image for $y = 400$ (blue line in the distance image) is plotted in the bottom right, demonstrating the sawtooth signal. This signal, corresponding to an entire row in the lookup table, may be compressed simply by preserving the zero points at $x = 447, 517, 900, 989$, with a compression factor in the hundreds.

in originally solid walls) via a conservative nearest-neighbors interpolation scheme.

This map rotation trick is the core of the CDDT algorithm. The sawtooth property implies a simple and effective compression scheme whereby each row in a rotated map is compressed simply by storing the zero points. Coincidentally, these zero points correspond exactly to locations of obstacles in the rotated map, so construction of our data structure is a simple and efficient matter of enumerating the x-positions of all obstacles in every row of the rotated map (see Section 5.3.1 or [2] for details).

In contrast to our approach, constructing a full lookup table typically requires ray casting from every discrete state in the table, a very computational operation.\(^1\) Given

\(^1\)As an aside, if the full three-dimensional lookup table is desired, it may be efficiently generated using our map rotation approach, accepting small interpolation artifacts associated with map rotation.
a set of zero points, one may quickly determine distance from any point along the line to the nearest zero point via simple binary search in logarithmic time (see Section 5.3.2 for details). These properties result in a ray casting algorithm which has fast precomputation time, query time, and low memory usage. Additionally, we will demonstrate below that this data structure may be updated incrementally in response to map modifications. Many optimizations arise from this formulation, which will be discussed further in later subsections.

Similar to the three-dimensional lookup table, we consider a discrete set of \( \theta \) directions in the construction of the full CDDT. This angular discretization is the primary source for error in the CDDT ray casting algorithm, since ray casting queries falling between the discretization grid must be rounded to the nearest discrete neighbor. One could improve accuracy by ray casting in both neighboring directions and interpolating the result. However, we do not perform this interpolation since it would generally require two ray casts per query, and in the context of particle filtering, the gains in ray casting accuracy are not worth the additional computational cost (as we demonstrate in Fig. 5-15).

Below, it is useful to consider the concept of a directional distance transform (DDT), which stores the distance to the nearest obstacle in a particular direction \( \theta \) for each possible \([x, y]\) in the map. The full CDDT data structure can be seen as a collection of compressed directional distance transforms, each of which allows for fast ray casting in a particular direction. The notation \( DDT_\theta \) refers to the DDT corresponding to some \( \theta \) ray casting direction. A single DDT is analogous to a single \( \theta \) slice of a three-dimensional lookup table.

### 5.3.1 Direct CDDT Construction Algorithm

Here, we denote the \( 3 \times 3 \) transformation matrix used to project map-frame coordinates into the frame of the various DDTs as \( P_{DDT_\theta} \). While the projection is primarily a rotation, in our implementation a translation is also applied to ensure that the projected \( y \) coordinate is non-negative for use in indexing the correct bin.
edge_map ← morphological_edge_map(map)

Initialize $\theta_{\text{discretization}}$ empty CDDT slices

for $\theta \in \{[\theta]\}$ do

define each occupied pixel $(x, y) \in \text{edge}_\text{map}$ do

$(x, y)_{DDT_\theta} = P_{DDT_\theta} \ast \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$

for each CDDT bin overlapping with $y_{DDT_\theta}$ do

bin.append($x_{DDT_\theta}$)

end for
end for

for each CDDT bin do

# Initialize successor/predecessor structure

bin = initialize_bin_structure(bin)

end for
end for

5.3.2 CDDT Query Algorithm

function ray_cast($((x, y, \theta)_q)$)

$(x, y)_{DDT_{\theta_q}} = P_{DDT_{\theta_q}} \ast \begin{pmatrix} x_q \\ y_q \\ 1 \end{pmatrix}$

bin ← zero points in row $y_{DDT_{\theta_q}}$ of CDDT slice $\theta_q$

$x_{\text{collide}} = \text{smallest element } x_{\text{collide}} > x_{DDT_{\theta_q}} \in \text{bin}$

return abs($x_{DDT_{\theta_q}} - x_{\text{collide}}$)

end function
5.3.3 Incremental CDDT Update

Since each element in the scene corresponds to a predictable set of zero points in the CDDT, if the map changes, one may insert or remove zero points accordingly, as outlined by the algorithm below. For efficiency, we recommend using a B-tree data structure to store zero points. B-trees provide asymptotically logarithmic insertion, deletion, and query runtime, as well as good cache characteristics in practice.

If one uses the morphological pre-processing steps during initial CDDT construction, special care must be taken in the incremental obstacle deletion routine. Specifically, if edge obstacles are removed, previously occluded non-edge pixels may be revealed, and must therefore be inserted in order to retain data structure consistency. This process is not prohibitively expensive since it only requires checking the eight adjacent pixels. Incremental map modifications are generally incompatible with the "pruned" variant of CDDT (see Section 5.3.4), as the exhaustive pruning operation makes ensuring data structure consistency during obstacle deletion non-trivial.

```cpp
function update_obstacle(int x, y, bool occupied)
    map[x][y] ← occupied
    for θ ∈ {⌊θ⌋} do
        (x, y)_{DDT_θ} = P_{DDT_θ} * \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}
        for each CDDT bin overlapping with y_{DDT_θ} do
            if occupied do
                bin.insert(x_{DDT_θ})
            else do
                bin.remove(x_{DDT_θ})
            endif
        end for
    end for
end function
```
5.3.4 Optimizations

Several important optimizations arise from the CDDT framework. In this section, we will discuss the primary optimizations available and implemented within RangeLibc.

Radial symmetry

Figure 5-5: Visualization of the radial symmetry present in CDDT zero point extraction. On the left, we show a visualization of the same CDDT structure as in Fig. 5-3 but without exploiting radial symmetry. On the right, we show the CDDT slices corresponding to the angles $\pi$ through $\pi + 2\pi$ flipped horizontally. Notice that this flipped version (highlighted in red) is an exact copy of the lower half (highlighted in blue). This redundancy is exploited by storing only the CDDT slices highlighted in blue, and inferring the red CDDT slices at query time.

Extracting zero points from each row of the DDT introduces rotational symmetry. In fact, this rotational symmetry can be visually observed by visualizing the CDDT data structure, as in Fig. 5-5. From one set of zero points, two rows of the DDT can be reconstructed - the rows corresponding to $\theta$ and $\theta + \pi$ for any particular $\theta$ (as shown in Fig. 5-6). Therefore, one only need compute and store CDDT slices for $0 \leq |\theta| < \pi$ and the DDT for $0 \leq |\theta| < 2\pi$ may be inferred by reversing the binary search direction, resulting in a factor of two reduction in memory usage.

Rotational symmetry may be further exploited in scenarios where ray casts are performed radially around a single point. While traversing the data structure to resolve a ray cast query $(x, y, \theta)$, the ray cast for $(x, y, \theta + \pi)$ may be resolved with
a single additional memory read. Once a search algorithm is used to discover the index \( i \) of \( x_{\text{collide}_\theta} \) in the CDDT, the index of \( x_{\text{collide}_{\theta+\pi}} \) is simply \( i - 1 \) as in Fig. 5-6. For example, this symmetry can be used in robots with laser scanners sweeping angles larger than 180° to reduce the number of data structure traversals required to compute the sensor model by up to a factor of two.

Map Preprocessing

```python
function morphological_edge_map(map)
    strel = 3x3 solid block
    return map - (map ⊕ strel)
end function
```

Figure 5-7: Function for generating morphological edge maps. The edge map is the difference between the original map, and an eroded map.

The memory usage of the CDDT data structure is linearly correlated with the number of occupied grid cells in the source map. Recognizing this correlation, it is important to discard as many occupied grid cells as possible without affecting ray casting solutions. Discarding unneeded occupied cells is especially important in degenerate cases, such as when large regions of the map are completely filled in. We
find that a simple map preprocessing scheme is quite effective in handling such cases, and in reducing memory usage for typical occupancy grid maps.

Figure 5-8: Example map left, edge map right. © 2018 IEEE

Consider a 3x3 block of obstacles in an otherwise empty map, as in Fig. 5-8. The center obstacle will never be the nearest neighbor in any ray casting query, because any such query would first intersect with one of the surrounding obstacles. To exploit this fact, one can use an "edge map" for CDDT generation without loss of generality. To generate the edge map, a morphological function may be used, as defined in Fig. 5.3.4. To ensure correct results with this optimization, one must check if the query point overlaps with an obstacle in the source map to avoid ray casting from the middle of removed obstacles.

**CDDT Bin Pruning**

While map preprocessing is a fast and effective method of removing unnecessary elements in the CDDT data structure, it is possible to further remove elements without affecting ray casting performance.

Figure 5-9: Example of a map which would result in many elements being projected into the same CDDT bin, even with edge-map preprocessing. In this case, 21 elements will be projected into the same CDDT bin for $\theta = 0$, but only two of those elements (the two black pixels) can possibly be used in a ray cast query with $\theta = 0$. CDDT pruning would help significantly in this case, reducing the number of elements in each CDDT bin to two (corresponding to the two black pixels). Cases similar to this one occur frequently in real maps, since walls often create continuous lines in real occupancy grid maps.

Consider a line of obstacles aligned along the X-axis, such as the map shown in
Fig. 5-9. Every element in this line will be projected into a single zero point bin in the $\theta = 0$ CDDT slice. However, the middle elements of the line will never result in a collision for similar reason as demonstrated in Fig. 5-8. Any ray cast from points on the line of obstacles will return early in a source occupancy grid check, and any ray cast from non-overlapping points collinear in the $\theta$ or $\theta + \pi$ directions will intersect either the first or last obstacle in the line. Therefore, in the $\theta = 0$ CDDT slice, one may discard the zero points corresponding to the elements in the middle of the line of obstacles without introducing error. This form of optimization is simple to compute in the cardinal directions, but non-trivial for arbitrary $[\theta]$ not aligned with an axis. Rather than analytically determining which obstacles may be discarded, it is simpler to prune the data structure by ray casting from every possible state, discarding any unused zero point. We refer to the pruned data structure as PCDDT. It’s possible that a more computationally efficient pruning scheme may exist, which would not require this brute force approach. We recommend the development of computationally efficient pruning methods as an area of future research.

Pruning does significantly increase pre-computation time. However, the reduction of memory usage is worthwhile for static maps (see Fig. 5-12). In addition to memory reduction, we find that pruning slightly improves runtime performance, likely as a result of improved caching characteristics, and the reduced number of zero points to search.

5.3.5 Theoretical Analysis

Here, we consider the theoretical algorithmic complexity characteristics of the CDDT algorithm. This section is reproduced from [2].

In this section, we refer to the width and height of the source occupancy grid map as $w$ and $h$, respectively. We refer to the diagonal length across the map as $d_{w,h} = \sqrt{w^2 + h^2}$. The CDDT algorithm requires the original map data to check for overlaps between $(x, y)_{query}$ and obstacles, prior to searching CDDT bins. Additionally, for each occupied map pixel, a total of $\theta_{\text{discretization}}$ float values are stored in the CDDT bins. Thus, the memory usage of the CDDT data structure is $O(n * \theta_{\text{discretization}} +$
\( w \times h \) where \( n \) is the number of occupied pixels in the edge map. Since we must sort each bin after CDDT construction, pre-computation time is at worst \( O(n \times \theta_{\text{discretization}} + d_{w,h}^2 \times \theta_{\text{discretization}} \times \log(d_{w,h})) \) for the same definition of \( n \). In practice we find precomputation time is similar to \( O(n \times \theta_{\text{discretization}} + d_{w,h} \times \theta_{\text{discretization}}) \) since each CDDT bin has a small number of elements on average, as evidenced by the high demonstrated compression ratio.

The pruning operation described in 5.3.4 reduces memory requirement, with a computational cost of \( O(w \times h \times \theta_{\text{discretization}} \times O(\text{calc\_range}_{CDDT})) \). The precise impact of pruning on memory usage is scene dependent, and difficult to analyze generally.

The ray cast procedure has three general steps: projection into CDDT coordinate space, the search for nearby zero points, and the computation of distance given the nearest zero point. The first and last steps are simple arithmetic, and therefore are theoretically constant time. The second step requires a successor or predecessor query on the CDDT bin structure. As previously discussed, the number of zero points in each CDDT bin tends to be small and is bounded in map size. Thus, at worst the search operation using either a sorted vector or B-tree requires \( \log(d_{w,h}) \) which is a small constant value for a fixed size map. Therefore, for a given map size, our algorithm provides \( O(1) \) query performance.

When using a B-tree for storing zero points, the cost of toggling an occupancy grid cell’s state is \( O(\theta_{\text{discretization}} \log k) \) where \( k \) is the number of elements in each associated CDDT bin. Using the same argument of bin size boundedness for fixed size maps, the cost of grid cell update becomes \( O(\theta_{\text{discretization}}) \) for maps of fixed dimension. In any case, this cost is generally not prohibitive for real-time performance in dynamic maps for reasonable choice of \( \theta_{\text{discretization}} \) (see Fig. 5-15).

5.3.6 Experiments and Results

We implemented the above described CDDT algorithm, along with several alternative ray casting algorithms, in the open source C++ package called RangeLibc\(^2\). In this

\(^2\)https://github.com/kctess5/range_libc
section, we present some of our findings related to the performance of the various algorithms we implement in RangeLibc. All synthetic benchmarks were performed on a computer with an Intel Core i5-4590 CPU @ 3.30GHz with 16GB of 1333MHz DDR3 ram, running Ubuntu 14.04.

For a more complete algorithmic evaluation, see my paper on the subject [2], which shares much of the same material presented here.

Synthetic benchmarks

Figure 5-11: Violin plots demonstrating histogram of completion time over a large number of queries for each ray cast method. The horizontal axis shows time in milliseconds, and vertical axis shows the number of queries that completed after exactly that amount of time. The left side demonstrates the observed performance with a random query sampling scheme. The right side demonstrates performance with a grid sampling scheme. Grid sampling appears to be more efficient for all methods, likely due to memory access pattern related caching effects. © 2018 IEEE
We evaluate algorithm performance in two synthetic benchmarks, using two maps. The first "grid" benchmark computes a ray cast for each point in a uniformly spaced grid over the three-dimensional state space. The second "random" benchmark generates ray queries uniformly at random. The so-called "Synthetic" map was created with Adobe Photoshop, whereas the "basement" map was created via a simultaneous localization and mapping (SLAM) algorithm on the RACECAR while navigating the Stata basement (map images shown in Fig. 5-10).

Performance in the particle filter

We also evaluated the performance of each of the considered ray casting methods in the context of particle filtering. Testing the ray casting methods within a particle filter...
is important because memory access pattern related effects can have a large impact on observed performance. In all particle filter benchmarks, we use the NVIDIA Jetson TX1 embedded computer onboard the RACECAR platform. We use a single thread for computing the Monte Carlo update step, though it could be easily parallelized across multiple threads for additional performance.

It is interesting to note in Fig. 5-14 that LUT provides very fast performance in the particle filter, roughly 3.4 times faster than PCDDT. We believe this is due to the tightly clustered memory access pattern of a well-localized particle filter, which yields a good low-level cache hit rate in the lookup table.

Figure 5-13 demonstrates the time required for evaluating the sensor model over a two minute period of driving the RACEAR around the MIT basement. This figure demonstrates the near constant time performance of both CDDT and the lookup table. In contrast, both Bresenham’s line and ray marching exhibit highly environment dependent runtimes. Variable sensor model evaluation time is problematic because the number of particles must be set in consideration of peak runtime, not just mean runtime.

The effect of angular discretization on particle filtering

Considering that both CDDT and the lookup table use a discrete angular resolution, one obvious question is how this discretization affects particle filtering accuracy. To quantitatively characterize discretization effects, we used a motion capture system to gather ground truth state information for comparison with the state inferred by our particle filter. We autonomously drove the RACECAR around our motion capture environment (using the Pure Pursuit controller discussed in Section 6.1.3) for a period

| Max particles maintained at 40Hz with 61 rays/particle |
|---------------------|----------------|----------------|----------------|----------------|
| BL      | RM    | CDDT  | PCDDT | LUT  |
| 400     | 1000  | 2400  | 2500  | 8500 |

Figure 5-14: Maximum number of particles that can be maintained in real time (approx. 40Hz) on the NVIDIA Jetson TX1. Stata basement map, 61 ray casts per particle, $\theta_{\text{discretization}} = 120$ where applicable. © 2018 IEEE
Figure 5-15: The effect of the $\theta_{\text{discretization}}$ parameter on median positional and rotational localization error (top), and the number of particle filter divergences (bottom) during a five minute period of driving in a motion capture system. All other parameters are held constant. Notice that a $\theta_{\text{discretization}}$ above around 100 bins results in error characteristics similar to the ray marching algorithm (which does not have an angular discretization).

Our testing framework provided a ground truth pose to the particle filter when the particle distribution significantly diverged from the motion capture data, allowed us to test extremely coarse $\theta$ discretizations which cause frequent particle filter divergence. We present the number of times the ground truth position was provided in Fig. 5-15.

Our findings regarding the affect of angular resolution match our intuition on the matter - the MCL algorithm is very robust to sensor noise and map inaccuracies, so it follows that it would be similarly robust to discretization artifacts.
Conclusion

We have shown that the proposed CDDT algorithm may be used to accelerate sensor model computation onboard mobile robots when localizing in two-dimensional occupancy grid maps. Beyond our theoretical and experimental evaluation, the CDDT algorithm has been successfully used (in the particle filter discussed here) by well over a hundred students in MIT’s 6.141.

It is worth noting that the CDDT algorithm is not well suited for parallel execution in graphics processing hardware. The algorithm does not have a very predictable memory access pattern, and thus is primarily constrained by the time required for random memory access. We did implement CDDT in CUDA for use on the Jetson TX1’s onboard GPU, but found only a small speedup, which did not justify the added complexity. It is possible that further work to optimize our GPU implementation could yield larger speedup. When a GPU is available, we recommend using the ray marching algorithm (a GPU implementation is included in RangeLibc), since it has a better suited memory access pattern for parallel execution.
Chapter 6

Controlling the RACECAR

Throughout this work, I implemented several different approaches to RACECAR control and path planning. In this section, I discuss the various methods that were tested.

Broadly speaking, these methods fall into two categories: trajectory-based, and global. In the trajectory-based methods, the goal is to find a feasible path from some start point A to some goal point B. Once a path is found, a separate path tracker is often used to navigate the path. In global methods, we find for every possible system state, an action which makes progress towards the goal. A mapping between state and action is typically called a "policy." If a given policy ensures that from any start point the minimal cost (for example, traversal time) is incurred along the way to the goal, that policy is called an "optimal policy."

6.1 Trajectory Search and Pure Pursuit Tracking

One common method of planning and control for autonomous cars uses low-dimensional paths (that is, paths which do not include full high-dimensional state information such as body frame velocities or wheel speeds), paired with a trajectory tracking system which handles the low-level car control, as long as it is feasible to do so. We consider one such system, which pairs a search based planner with the Pure Pursuit tracking algorithm.
6.1.1 Path Planning

Often, traditional search algorithms, such as A* [19] are used to determine feasible paths. Alternatively, many people use randomly exploring methods such as RRT [20] (or variants thereof) to quickly explore potentially high-dimensional search spaces. In my work with the RACECAR, I implemented an A* variant called Space-Time Exploration Guided Heuristic Search [21] for solving the path planning problem in large environments such as the MIT basement system. Since path planning using this algorithm is not the focus of this thesis, I urge interested readers to consult the literature on the subject [21].

I also considered manually planned trajectories, generated using a custom ROS node which integrates with RViz’s built-in point selection tools.
6.1.2 Pure Pursuit Trajectory Tracking

The pure pursuit algorithm [22] is a widely used technique for approximately following preplanned trajectories under car-like dynamical constraints. The algorithm proceeds by iteratively determining a "lookahead point" along the followed trajectory, and applying the control necessary to travel the arc passing through the point and tangent to the car. Pure pursuit can be thought of as a "follow the carrot" algorithm where the robot is always pursuing a moving target which is some fixed distance relative to the car ahead along the path.

The pure pursuit approach reduces the problem of tracking a preplanned trajectory to one of determining the proper lookahead point. While this may sound like a simple problem, it is in fact somewhat challenging, and there are many implementations possible, each of which handles various degenerate cases differently. The general method which we typically use and recommend is the same as the method presented in [22], which we will describe here for completeness. This method has several desirable properties, such as simplicity, and the ability to work anywhere along the path without a temporal dependency.

Algorithm for finding the Pure Pursuit Lookahead Point

1. Find nearest point to the car along the trajectory

2. Starting at the nearest point to the car, proceed forwards along the trajectory until a point along the trajectory is found that is exactly $d_{\text{lookahead}}$ meters away from the car.

There are several special cases that must be handled:

- If the nearest point along the path is greater than some threshold distance $d_{\text{reacquire}}$ away from the car, give up and stop. Typically, the path planner would be subsequently invoked in this scenario.

- If the car is between $d_{\text{reacquire}}$ and $d_{\text{lookahead}}$, use the nearest point as the lookahead point
• If there is no point forwards along the path that is $d_{\text{lookahead}}$ meters away from the car (e.g. near the end of the path), use the last point along the trajectory as the lookahead point. Alternatively, one may employ some separate controller to navigate to the very end of the trajectory.

• If there’s an obstacle in the arc connecting the car to the lookahead point, stop. Typically, the path planner would be subsequently invoked in this scenario.

6.1.3 Variable Speed Pure Pursuit

![Figure 6-2: Example trajectory with speed profile. Path is shown on the right, with circles of varying radius indicating planned traversal speed. Red circles indicate deceleration. Green circles indicate acceleration or constant speed. The maximum speed profile based on path curvature is shown on the upper left. The maximum speed profile based on both path curvature and acceleration bounds is shown on the bottom left. In this example, 5.0 m s$^{-2}$ is chosen as a global speed limit to avoid excessively high speeds.](image)

When attempting to quickly follow a trajectory, it is often useful to be able to vary drive speed in order to drive fast in the long straight sections, and slow in the sharp corners. I implemented an extension to the pure pursuit algorithm that allows for trajectory speed profiles to be automatically generated and executed. This system allowed my car to complete the Stata basement loop in roughly 35 seconds. With more parameter tuning, this time could likely be further reduced using the same
controller. Here, I discuss the specifics of my implementation. This approach is not rigorously inspired by a mathematical backing, rather, it is a practical method that was designed from an engineering standpoint to make the car drive faster. I find that this approach is simple to implement and tune as an extension of Pure Pursuit, and it works well in practice. For a more principled approach, I recommend considering other methods, such as Model Predictive Control (MPC) [23].

The general idea of my approach is to assume that the car exactly follows the path, and then consider, at each point along the path, the maximum rate of travel that would allow the car to turn sharply enough to pass through the next waypoint. I experimentally created a mapping between path arc radius and maximum rate of travel by driving the car in circles with varied speeds and steering angles, and measuring the traveled path arc radius, shown in Fig. 6-4.

Once the preplanned path is augmented with maximum rate information, a forwards and backwards pass through the trajectory ensures that acceleration and deceleration bounds are respected. The space-indexed speed profile represents an upper bound on the car’s safe travel speed, so, in practice, we attempt to travel slightly slower than this profile, to ensure a margin of error.

In this particular project, I used the manually planned trajectory shown in Fig 6-1. Alternatively, any search algorithm such as A* or RRT* could be used. The approach described here works best when the input trajectories are smooth.

**Trajectory resampling**

The first step in my variable speed Pure Pursuit algorithm is to resample the trajectory so that each waypoint along the piecewise linear trajectory is approximately 1 meter apart. Resampling ensures that the distance between consecutive waypoints is small so that the Ackermann path arcs considered in generating the maximum speed profile closely follow the trajectory.
Maximum speed profiling

The next operation in generating a speed profile is to step along the resampled trajectory, one waypoint at a time, and determine an upper bound on the speed that would allow the car to safely travel through the next waypoint (allowing for understeer at high speeds). I use data collected (see Fig. 6-3) on the real car to determine the maximum speed for each required path arc radius. My mapping between path arc radius and maximum travel speed is presented in Fig. 6-4. An example speed profile after the maximum speed determination step is shown on the top left of Fig. 6-2.

Acceleration bounds

Once the trajectories have been labeled with maximum traversal speed for each waypoint, it is necessary to make two more passes in order to ensure that real-world acceleration and deceleration bounds are satisfied. Each pass through the speed profile must simply ensure that the maximum slopes present are within the car’s measured acceleration and deceleration bounds, clipping the profile as necessary. An example speed profile which takes into account acceleration bounds is shown in Fig. 6-2.
Figure 6-4: Mapping between steering angle and maximum traversal speed. Based on the data from Fig. 6-3, and used in automatically generating maximum speed profiles for given trajectories.

**Variable speed trajectory execution**

Following a preplanned trajectory with variable speed proceeds in much the same manner as the pure pursuit algorithm presented in Section 6.1.2. The key differences are as follows:

- The lookahead distance $d_{\text{lookahead}}$ is linearly scaled with current drive speed according a preset $p_{\text{lookahead}}$ parameter.

- The throttle command is set as the minimum between the preplanned speed at the nearest trajectory point, and the maximum traversal speed given the path arc radius required for navigating to the found lookahead point. This conservative approach ensures the car doesn’t attempt to drive excessively fast when it must turn harder than expected by the trajectory.
6.2 Autonomous Drifting with LQR and Trajectory Optimization

The ultimate goal of this thesis was to create a RACECAR controller that could "drift." Drifting is a challenging driving maneuver in which oversteer is intentionally initiated, and then controlled via steering and throttle inputs such that a large slip angle is maintained in the rear tires. This technique allows the driver to take advantage of the highly nonlinear sliding-friction dynamics present at high slip angles.

Drifting is not generally the optimal control method - faster laps around a race track are usually achieved by maintaining the in-traction driving regime. However, there are cases where controlling drifts is necessary, such as in low traction settings when driving on loose dirt, ice, or rain water. Additionally, in rear-wheel drive vehicles, drifts may be unintentionally initialized if too much throttle is applied in turns.

Typically, autonomous car control systems attempt to stay in-traction with low slip angles. This decision simplifies vehicle dynamics, since linear approximations of tire forces may be used, and remaining in-traction is typically the optimal strategy under good driving conditions. Since autonomous cars will eventually be expected to operate in inclement weather conditions, we believe that it is important to investigate the design of autonomous controllers which can operate through high slip angle drifting regimes. Additionally, there is a certain "cool factor" in autonomously sliding a car sideways, which captures ones’ imagination.

In pursuit of this goal, we discovered that autonomous drifting is quite challenging. Not only do the typical car-like nonholonomic driving constraints come into play, but also the vehicle dynamics are highly nonlinear at large slip angles, and especially complicated at the transitions between the high-traction and low-traction regimes.

In this section, we discuss a control technique which relies on nonlinear trajectory optimization, and LQR trajectory tracking. This combination allows for the planning and execution of drifting maneuvers through the use of a high-dimensional dynamics model which models the large-slip-angle case. We implemented and tested this controller for the RACECAR platform as a final project [24] for MIT’s class 6.832.
6.2.1 Nonlinear Trajectory Optimization

In this section, the trajectories considered differ from trajectories in the previous section (6.1.2) in that they contain high-dimensional full state and control information. In our case, we consider the 7D model and 2D controls presented in Section 3.3.1, so our trajectories are matrices of dimension $N \times 9$, where $N$ is related to the length of the trajectory.

Generating these trajectories is nontrivial, since they must both travel between the specified start and end states, and obey system dynamics and control constraints (unlike the lower-dimensional trajectories previously considered). We pose the trajectory generation problem as the solution to a constrained nonlinear optimization problem using the direct transcription method (see Section 11.3.1 "Direct Transcription" in [25]). This formulation allows the use of high performance optimization tools (such as the open source Ipopt [26] or the commercial package Snopt [27]).

Constraints considered in the optimization problem include:

- System dynamical constraints between consecutive states in the trajectory
- Start and end state constraints
- Control constraints, such as absolute and rate-of-change control limitations
- Global constraints on the bounds of each dimension (e.g. maximum possible $x$ or $y$ coordinates)

The optimization can also minimize some cost metrics (such as control effort), but we found that simply requiring that trajectories satisfy the various constraints (without providing a cost) resulted in a much faster solve time.

These constraints (paired with our highly non-linear dynamics) make this optimization non-convex and prone to dynamically infeasible local minima. To mitigate this non-convexity, we provide initial guesses at the solution by progressively considering more complex dynamics models, starting with a linear tire model, and moving towards the nonlinear tire models (presented in Section 3.3.2).
To solve more complex planning problems than those tested here (such as those with obstacles), one could first consider a lower-dimensional search algorithm, such as those discussed in Section 6.1.1. The solution to the lower-dimensional problem could serve as an initialization point for a higher-dimensional problem. Alternatively, the problem could be hierarchically subdivided, and multiple smaller high-dimensional trajectory optimization problems could be solved for different sections of the path (this would be in some sense similar to an MPC [23] solution).

6.2.2 LQR Based Trajectory Tracking and Stabilization

The Linear Quadratic Regulator (LQR) is a commonly used control technique which provides optimal control for linear systems with quadratic cost functions. Despite being formulated for linear systems, LQR can often prove useful even for controlling nonlinear systems, such as the RACECAR, by using linear approximations to the system dynamics.

LQR provides a cost minimizing controller for systems of the form \( \dot{x} = Ax + Bu \), with quadratic cost functions of the form \( x^TQx + u^TRu \), where Q and R are symmetric positive semidefinite matrices. Assuming that the system dynamics implied by the A and B matrices are controllable, the solution to the algebraic Riccati equation [28] implies an optimal control law. A more complete derivation of the LQR theory is provided in [25].

In this work, we use LQR as a feedback controller to stabilize the RACECAR around trajectories discovered via nonlinear trajectory optimization. This technique allows the car to track preplanned trajectories despite model inaccuracies and/or system disturbances (such as misaligned initial conditions). Here, we will show how LQR may be used for trajectory tracking.

In order to adapt LQR to the trajectory tracking case, we consider a time-varying version of system dynamics, parameterized by time \( t \). This control approach is often referred to as time-varying LQR (tvLQR) since the matrices A and B change with time. In general, Q and R may also change with time, but we do not vary Q and R in our implementation.
Since we wish to stabilize about the preplanned trajectory, we use LQR to determine a corrective factor with respect to the plan. Thus, we consider a relative coordinate system.

\[ \bar{x}(t) = x(t) - x_0(t), \quad \bar{u}(t) = u(t) - u_0(t), \quad \dot{\bar{x}}(t) = \dot{x}(t) - \dot{x}_0(t) \]

In the above equations, \( x_0(t) \) represents the planned state-space trajectory, while \( u_0(t) \) represents the planned control trajectory. With this relative coordinate system, we can use LQR to express the residual controller: \( \bar{u}^*(t) \) which represents a corrective factor. By simple substitution, we can observe that \( u^*(t) = \bar{u}^*(t) + u_0(t) \). Also, note that the open-loop controller may be written simply as \( \bar{u}^*(t) = 0 \), which implies that the preplanned control is applied with no corrective factors.

To solve LQR, we must first determine a linear approximation to our system dynamics, so that we may write:

\[ \dot{\bar{x}}(t) \approx A(t) \ast \bar{x}(t) + B(t) \ast \bar{u}(t) \]

To determine the \( A \) and \( B \) matrices, we should consider the definition of \( \dot{x} \). For notational convenience in the following equations, we omit the time index (e.g. \( x(t) \) is written simply \( x \)).

\[ \dot{x} = \dot{x} - \dot{x}_0 = f(x, u) - f(x_0, u_0) \]

In general, a linear approximation of \( \dot{x} \) breaks down, since \( f \) is a nonlinear function. However, we may locally approximate the nonlinear function \( f \) by considering the Taylor expansion around \( \bar{x}(t) = 0, \bar{u}(t) = 0 \), which corresponds to a linearization about \( x_0(t), u_0(t) \).

\[ f(x, u) \approx f(x_0, u_0) + \frac{\partial f(x_0, u_0)}{\partial x}(x - x_0) + \frac{\partial f(x_0, u_0)}{\partial u}(u - u_0) \]

Therefore, by substitution:
\[
\dot{x} \approx \frac{\partial f(x_0, u_0)}{\partial x}(x - x_0) + \frac{\partial f(x_0, u_0)}{\partial u}(u - u_0) = \frac{\partial f(x_0, u_0)}{\partial x} \bar{x} + \frac{\partial f(x_0, u_0)}{\partial u} \bar{u}
\]

Thus, our time-varying A and B matrices are simply the Jacobian of the system dynamics with respect to state and control perturbations about the planned trajectory. The Jacobian may either be determined via an exact formulation of the derivatives of system dynamics, or by finite differences (i.e. slightly perturbing the state \(x\) and control \(u\) about \(x_0\) and \(u_0\)). In the above derivation, we considered time to be continuously variable. In practice, we consider discrete times \(t\), interpolating along the planned trajectory in our controller’s main loop.

### 6.2.3 Implementation and Results

We implemented and tested the above described LQR trajectory tracking controller for the RACECAR. This section discusses some of the details of our implementation, and our results.

In computing LQR control for preplanned trajectories, we considered the 7D dynamics and 2D control model of the RACECAR, as presented in Section 3.3.1. Although it is possible to determine exact derivatives for this model, for convenience and avoidance of math-related bugs, we used a finite-difference-based method for computing the A and B matrices. We used a combination of linear and Arctan tire models (see Section 3.3.2), since we found that this combination worked well in practice for both trajectory optimization and LQR.

Full state estimation (including pose and body frame velocities) was performed via the motion capture system available in MIT’s building 31. Our runtime system allowed us to select and initialize trajectories via the RACECAR’s wireless joystick controller, and to set the motion capture coordinate frame origin to the car’s pose.
Engineering challenges

In implementing this controller, we encountered and solved many engineering challenges, including system latency and control fidelity.

One such challenge was the real-time wireless transmission of motion capture data from the offboard motion capture system to the RACECAR. We found that using the onboard Jetson WiFi, our communication latency was highly variable, at times in excess of 100ms. This problem was solved by integrating a high-end gaming router and a dedicated onboard WiFi-to-Ethernet adapter.

Another challenge was in the low-level motor control. The standard RACECAR platform uses a sensorless drive motor, with open-loop low-level motor control. We found (especially at low throttle settings) this form of motor control was unreliable: it would sometimes fail to start moving immediately, and throttle settings of less than 0.5 m/s would generally have no effect. This problem inspired us to install a higher quality sensored brushless drive motor, so that we could use closed-loop field-oriented control [29] in the motor driver.

Finally, we had to ensure that our own code was highly performant, so as to minimize the latency between receiving location data and applying control. For efficiency, we implemented the entire runtime software architecture as a single C++ executable.

Results

Our controller was able to successfully control the car under high-slip conditions. We tested several trajectories, all precomputed via nonlinear trajectory optimization.

Straight Path

- Start state: (0.0, 0.0, 0.0)
- End state: (2.0, 0.0, 0.0)
- Time allowance: 1.56 seconds
Figure 6-5: Several different poses observed along the "straight" maneuver with the tvLQR controller.

Figure 6-6: Observed RACECAR trajectories with the straight trajectory over dirt. Multiple trials with initial state perturbations are demonstrated for the tvLQR controller. The small arrows indicate the car’s start and end orientation. Notice that the system is able to nearly recover from even large initial state errors, despite only 1.56 seconds of activity. The open-loop controller makes no attempt at tracking error recovery. The mean terminal position error of the observed open-loop trials was found to be 0.36m, whereas for tvLQR the same statistic was found to be 0.20m, including trials with highly perturbed initial states. If one only considers tvLQR trials with small initial perturbations, this error is found to be 0.12m, representing a significant reduction as compared to open-loop.

This trajectory required quickly driving forwards in a straight line, and coming to a stop two meters in front of the start. Especially with dirt on the ground, the car exhibited significant "torque steer" where it would rotate to one side when significant throttle was applied. We found that the tvLQR controller was quite effective in controlling this unwanted rotation. Additionally, the controller did a good job of recovering from significant start-state perturbations, despite only being active for
1.56 seconds. In contrast, the open-loop controller consistently undershot the planned trajectory (likely due to the reduced traction due to dirt on the floor) and "torque steered" to the left under high acceleration.

**Drifting Park**

Figure 6-7: Several different poses observed along the drifting park maneuver with the tvLQR controller. The rear end of the car begins sliding near the second half of the trajectory (e.g. the last four poses shown here). Notice the dirt on the ground, which reduces tire grip and increases the surface variation.

- Start state: \((0, 0, 0, 0)\)
- End state: \((-0.5, 1.7, 1.5\pi)\)
- Time allowance: 2.64 seconds

The parking maneuver required that the car quickly go from the origin to a "parking spot." The close proximity of the parking spot, combined with the short time provided, necessitate an oversteer slide near the end of the trajectory. While the car typically did not arrive exactly at the goal state, we found that it was quite consistent regardless of start-state perturbations. This systematic error seems to imply some model inaccuracy relating to the car's high-slip-angle sliding dynamics, which is
Figure 6-8: Observed RACECAR trajectories with the drifting park trajectory over dirt. Multiple trials with initial state perturbations are demonstrated for the tvLQR controller. In this example, the open-loop controller fails entirely, missing the intended parking spot by a large margin. This failure is largely due to the dirt induced loss of traction, and other model inaccuracies. On the other hand, the tvLQR controller is able to fairly reliably track the intended path, even in the case of significant initial state perturbations. We note that the car does not exactly reach the end of the trajectory. We primarily attribute this to systematic model error, especially problematic near the end of the trajectory where feedback control cannot compensate for the error.

Donuts

- Start state: \((0,0,0,0,0)\)
- End state: \((0,0,0,4.0\pi)\)
- Time allowance: 4.8 seconds

The "donut" maneuver required that the car travel from the origin, and then back to the origin, but with two full rotations. In fact, this maneuver is not a "donut" in the traditional sense, which would have the car's rear tires spinning in circles around mostly stationary front tires. Nevertheless, this is an interesting test case, because it incurs a large slip angle in the rear tires as the car quickly rotates. Our tvLQR controller is able to correct for model inaccuracies which cause the open-loop controller to quickly diverge from the planned path.
Figure 6-9: Several different poses observed along the "donut" maneuver with the tvLQR controller.

Figure 6-10: Observed RACECAR trajectories with the donut trajectory (without dirt on the ground). The open-loop controller fails to track the planned trajectory, over-rotating by roughly $1.36\pi$ radians and translating away from the origin. In contrast, the tvLQR controller is able to much more accurately track the planned path, controlling for excessive oversteer by counter-steering. We note that the car does not exactly reach the end of the trajectory. We primarily attribute this to systematic model error.

**Qualitative Results**

Our experience with this controller provides a few insights. We found that the
The approach is surprisingly robust to initial-state perturbations, as is demonstrated by Figures 6-6, 6-8, and 6.2.3. This robustness is surprising primarily because the use of linear approximations to the nonlinear dynamics in the tvLQR controller. One might expect that in the presence of large perturbations, the linear approximation would fail and the car would do something highly suboptimal. While failure did occur in some cases, it appears that the linear approximation, when paired with high rate feedback, is quite robust.

While we did have success with the tvLQR approach, we discovered that it had some pitfalls. Namely, it requires both an accurate dynamics model, and full state information. In the case of the RACECAR, both of these requirements are difficult to fulfill. As we have seen in the modeling section (3.3.1), the car has highly nonlinear dynamics with many complex second-order effects, such as weight transfer. Additionally, full state information can be difficult to acquire, both in our case, and in the general case. In future work, it would be interesting to pair this control strategy with onboard state estimation, such as the particle filter presented in this thesis.

Even if a model is fairly accurate, any systematic error can be problematic in trajectory execution, especially in underactuated systems. For example, if the planned trajectory relies on the rear wheels sliding a certain amount, but the rear tires do not slide as much as anticipated, the underactuated nature of car-like dynamics limit system controllability and make recovery challenging or impossible. We encountered problems with systematic error on some trajectories in which the car would under-rotate near the end, such as in the parking or donut maneuvers.

In our experience, if model inaccuracies must be accepted, then it can be helpful to make the model a "less extreme" version of reality, rather than a more extreme version. For example, if the model predicts the car's maximum acceleration rate to be 10 m s\(^{-2}\), but in fact it is only 9 m s\(^{-2}\), then the car will be physically unable to track trajectories which rely on the predicted maximum acceleration. On the other hand, if the model's maximum predicted acceleration is 8 m s\(^{-2}\), then the car will easily be able to track the planned paths while still remaining within the car's limits. Similarly, if the car was oversteering more than predicted by the model, it could correct with
steering, whereas correcting understeer is more challenging. In future work, it would be interesting to attempt to generate trajectories which are strict "underestimates" of the car’s abilities, so the planned trajectory remains within the car’s controllable region.

Despite some pitfalls, our experiments have shown that this control method is powerful, and we believe that it has large potential as a drifting controller. Since the method’s primary drawbacks seem to arise from model inaccuracy, this might serve as a good basis for a hybrid model-based/learning-based approach such as the method proposed in [30].

6.3 Parallel parking the RACECAR with Dynamic Programming

Figure 6-11: Parked RACECAR

One approach to global optimal robotic control involves computing an approximate solution to the Bellman Equation via an algorithm known as value iteration [31]. This method is significantly different from the planning methods discussed above, in that it seeks to define a global policy for moving from any point in space to some goal
state, while incurring the minimal "cost" along the way. This approach is extremely attractive since it is (resolution) complete [6], and once computed, may be efficiently evaluated at any state, with the expectation of near optimal results. However, it is often difficult to apply in practice, due to the fact that its algorithmic complexity and memory requirements are exponential in the size of the state space.

In this work, and as a final project for MIT’s 6.231 'Dynamic Programming and Optimal Control,’ I implemented a value-iteration-based controller for parallel parking the RACECAR in a constrained environment. This technique allows the RACECAR to quickly, safely, and robustly travel from any starting pose in a mapped environment to a predetermined parking spot. Here, I will briefly discuss this approach and the results obtained. Much of this section was presented previously in my final paper for 6.231 [32], it is included here for completeness.

I used the value iteration algorithm to numerically determine a "value function", which defines the cost of the car being in any given state. Under the following problem formulation, this value function implies a time-optimal policy for navigating into a parking spot, which respects car-like nonholonomic driving constraints. At runtime, I estimate the car’s state using my previously discussed particle filter implementation (Section 4.1.4) and apply the control which results in the minimal-cost future state.

To maintain computational tractability in this effort, I make several simplifying assumptions. Namely, I assume state transitions are deterministic, the environment is known ahead of time, and that the car’s state is known at each time. I also assume that the car is traveling slow enough that the geometric car model (see Section 3.2) accurately describes vehicle dynamics. While these assumptions are not strictly valid in real operating conditions, they are not so limiting as to prevent practical operation.

### 6.3.1 The Value Iteration Algorithm

In order to determine the optimal value function, it is necessary to solve the Bellman equation. Here, \( J(x) \) is a representation of the value function.

\[
J(x) = \min_{u \in U(x)} [g(x, u) + J(F(x, u))] \tag{Eq. 6.1}
\]
In this equation, $x$ represents any state in the state space. $g(x, u)$ represents the "stage cost," which is the cost of performing an action $u$ while at state $x$. $J(x)$ represents the expected "cost-to-go" from an initial state $x$ to the terminal state. $F(x, u)$ represents the system dynamics, providing the expected future state, given that a control $u$ is applied while at state $x$. $U(x)$ represents the set of actions that are available at state $x$.

It is important to note that this is a recursive formulation, which implies that the cost of being at a state is the cost of being at some state, plus the cost of controls at that state, plus the cost of being at the future state, which is achieved by applying the chosen control at that state. There exists some "policy" $\mu(x)$ which maps from every state in the state space to the optimal control action at that state in order to minimize the function $J(x)$. This minimizing policy is known as the "optimal policy" and is denoted $\mu^*(x)$. The minimal cost from each initial state is denoted $J^*(x)$ and arises from applying the optimal policy at all states:

$$J^*(x) = g(x, \mu^*(x)) + J(F(x, \mu^*(x)))$$  \hspace{1cm} (Eq. 6.2)

Value iteration (VI) numerically determines the optimal value function $J^*(x)$ by iteratively refining an initial estimate until further refinement yields no improvement in cost for any state $x$. Value iteration is a powerful approach, because there are strong guarantees that the iteration will eventually terminate at the globally optimal policy (allowing for discretization induced error). These guarantees, and a more complete derivation are shown in [31].

The value iteration algorithm enumerates all possible actions, for a given state $x$, and picks the action which minimizes the total expected cost, effectively evaluating Eq. 6.1 one state at a time. This process is often referred to as a "backup" of the state $x$. The full VI algorithm proceeds by backing up every state $x$ using a (potentially suboptimal) value function estimate $J(x)$ in order to form an improved estimate $J'(x)$. When $J'(x) = J(x)$, no further iteration will have any affect so the algorithm terminates.
6.3.2 Problem Formulation

With the definitions of state space, cost \((g(x, u))\), action \((U(x))\), and motion model \((F(x, u))\), the value iteration algorithm proceeds in a formulaic fashion, as described above. The definitions used in parallel parking the RACECAR are provided below.

State Space

In this section, I consider a four-dimensional state space.

\[
x = \begin{bmatrix}
x_w \\
y_w \\
\theta_w \\
\psi
\end{bmatrix} = \begin{bmatrix}
\text{world x-position} \\
\text{world y-position} \\
\text{world orientation} \\
\text{throttle}
\end{bmatrix}
\]

Velocities are omitted because inclusion thereof would necessitate additional dimensions, which would result in the value-iteration-based method described here being computationally infeasible.

In value iteration, the chosen state space is generally discrete, except for in a few special cases where an exact solutions exists. I found, in practice, that a relatively small value of angular discretization is required for good performance, roughly 40 or more bins (although this number was not rigorously derived). For simplicity, the \(x, y\) dimensions are discretized to the same resolution as the input map (often 5cm resolution). For throttle, I only consider three values: \{-1, 0, 1\}, representing slow forwards, stopped, and slow backwards, respectively.

Action Model

For simplicity, at each state, I consider a discrete set of available actions \((\delta, \psi)\) representing a choice of steering angle and throttle, respectively. The set of steering angles considered is given by discretizing the range between the minimum and maximum steering angles, always including the option with \(\delta = 0\). At each non-terminal state, the set of throttle options \(\psi\) includes \{-1, 1\} representing slow forwards and
backwards motion, respectively. For terminal states, the only action available is
\((\delta = 0, \psi = 0)\), representing a stopping action.

**Motion Model**

![Motion Primitives Considered in the Parallel Parking Motion Model](image)

Figure 6-12: Motion primitives considered in the parallel parking motion model. Each line represents one action \((\delta, \psi)\). The radii of these arcs are determined via the empirically determined mapping presented in Fig. 6-13. Timestep duration, and thus the distance traveled along these path arcs, is a parameter of the value iteration algorithm presented here.

In my parallel parking-controller, I assume that wheel slip is not a major factor, in order to reduce the size of the state space. To maintain the accuracy of assumption, I limit the car to slow driving speeds. I consider a slight adaptation of the geometric steering model discussed in Section 3.2. Namely, I use an empirically determined mapping between steering speed and path arc radius. This mapping is developed in a similar manner as in Section 6.1.3 and is presented in Fig. 6-13.

**Cost Function**

Since we are interested in determining a time-optimal policy for parallel parking, the cost function is straightforward. Goal states (states that are within the parking spot) are considered to have zero cost. Each action is assigned a cost equal to the control timestep, which represents the time spent performing that action. With this
formulation, the value function has a real world meaning: the expected amount of time required to travel from any state to the parking spot.

To prevent the car from attempting to rapidly change directions, as in a K-point turn with large values of K, I add an additional cost that represents the time required to switch driving direction. This term is the reason that throttle (ψ) is included in the state space.

To prevent wall collisions, states which would collide with obstacles in the environment are assigned a very high cost. The specific value of this cost is somewhat irrelevant, so long as it is very high compared to the worst case parking maneuver duration (otherwise the algorithm might decide it’s worthwhile to "drive through" an obstacle as a short cut!). In our implementation, we arbitrarily chose an obstacle cost of 100000.

One could potentially define an additional smaller cost for non-colliding states which are near obstacles to encourage the car to use paths far from obstacles where possible. However, doing so would affect the time-optimality of the solution, and I did not find it necessary in my implementation.
6.3.3 Implementation and Results

![Figure 6-14: The full 210x210 (5cm resolution) "East Campus kitchen" map used in testing (left). In this map, the green rectangle represents the parking goal region. On the right, several different slices ($\theta = 0^\circ, 90^\circ, 180^\circ, 270^\circ$) of the converged value function $J$ are demonstrated. Lighter colors represent higher cost, dark regions are low cost. The parking spot is black here, since it is the only zero cost region.]

I implemented the above described system in the C++ programming language. My code is available online\(^1\). Here, I will discuss a few details of my implementation, and the results attained.

**Value Function Representation**

While the state space is discretized here, for a correct numerical solution, one must be able to evaluate cost at any continuous state. Thus, the value function is represented as a three-dimensional tensor. A bilinear interpolation scheme is used to evaluate values at states $(x, y, \theta)$ which do not lie on the grid, thus providing a continuous approximation which becomes exact as the distance between neighboring states approaches zero. As in the state space, an absorbing boundary condition is used in the $x$ and $y$ dimensions, while a periodic boundary condition is used for the $\theta$ dimension. The throttle $\psi$ axis is not interpolated, since there are only three discrete choices considered.

\(^1\)https://github.com/kctess5/dynamic_parking
Since the optimal policy is implied by the value function, it is not necessary, in practice, to store the policy explicitly. Instead, the optimal action may be determined for any given state by performing a single backup on that state and choosing the cost minimizing action \((\delta, \psi)\).

Optimizations

One major optimization which significantly accelerates the computation of the converged value function is to limit the set of states backed up at each iteration. Specifically, if a state is not within the maximum distance traveled during one control step of a state which changed cost on the previous iteration, it can be skipped without loss of generality. This optimization yields a large speed improvement by exploiting the spatial relationships present in the state space, reducing the number of "useless" backups which do not change value of the backed up state.

Additional smaller optimizations are applied, such as the precomputation of the motion model’s motion primitives in order to avoid expensive trigonometric operations during value iteration.

Results

In my implementation, value function precomputation typically takes between 10 seconds and 5 minutes, depending on scene complexity, size, and discretization. We performed testing in a small region of a student lounge in the East Campus dorm at MIT (a map of this environment is available in Fig. 6-14). The free space in the kitchen is small enough that the robot is unable to make a full circle without crashing, necessitating K-point turns. With a \(210 \times 210 \times 60\) discretization, considering 22 different actions, using a step size of 3 pixels (along the path arc curves), VI converges in 166 iterations after 234 seconds, backing up roughly 151 million states. Without applying the optimization described above in Section 6.3.3, the same number of iterations would have required 878 million state backups. Therefore, in this case, the aforementioned optimization results in an 82.8% reduction in computation. An
Figure 6-15: A demonstration of real paths traversed by the RACECAR during operation. Black regions in the map represent occupied space, white regions are free space. Picture of the RACECAR represents the initial pose of the car. Green region is the goal position. Gray areas represent the region of space swept by the footprint of the car during traversal from the initial pose to the goal.

I found that this approach worked well for parking the car, but it did require a few modifications along the way to ensure good performance. This method is highly dependent on an accurate model, since the time-optimal solution often requires control near the edge of the system’s capabilities. In the original implementation, I used the Ackermann model of car dynamics. However, this did not perform very well since the model predicted that the car’s turning arc radius was smaller than reality near the extremes of steering angle setpoints (see Fig. 6-13). This systematic error meant that in some cases, especially near discontinuities in the value function, the policy relied on a non-feasible maneuver, which often resulted in the car getting "stuck" as it went back and forth between states with directly conflicting controls (for example, two states with opposite throttle setpoints). Correcting the model to better match reality significantly mitigated this problem, and improved parking speed and robustness. It is possible that adding a cost to extreme controls would discourage the system from

\(^2\)http://rayban.vision/#parallel_parking_video
operating near its limits, preserving some control authority for cases where there is a large incentive for extreme controls.

Additionally, localization error can be problematic. If the localization estimate oscillates back and forth across a discontinuity in the implied policy (for example, one state which requires reversing, and another state which requires going forwards), the car can similarly get "stuck." This problem was mitigated by performing a single value iteration backup using the system's current state, allowing the car to incorporate the cost of direction reversal at runtime - it would only switch directions if there was a large cost incentive in doing so. This technique could be extended to include state uncertainty information by choosing the expected value of cost minimizing action over the state space belief, rather than just considering the maximum likelihood pose estimate. However, I did not find this necessary in achieving robust parking behavior.

While it is clear that this is a powerful approach, it is limited by its dependency on an accurate model and state estimation, and the requirement that the system's state space be small enough that a sufficiently-fine state discretization is not computationally prohibitive.
Chapter 7

Simulating the RACECAR

Figure 7-1: Screen capture of the RViz while the custom RACECAR simulator is running. Pose history is shown with the red arrows. The white dots overlaid on the map are from the simulated laser scanner.

In developing software for the RACECAR, or more generally any robotic system, it is often useful to algorithmically simulate the system’s behavior in software. Simulation allows new software components to be rapidly tested at a low cost, without some of the real world problems that can make unbiased algorithmic evaluation difficult. In Robot Operating System (ROS), it is often possible to simulate a robot’s
low-level interfaces such that higher level software is unaware that it is in a computer simulation. This ensures that (at least nominally) if the software is tested and runs in simulation, it will perform similarly in reality.

In order to put the RACECAR into "The Matrix" [33] (i.e. a simulated environment), one must simulate the following (or at least whatever subset of the following is used by higher level software):

- The car’s position and orientation in a virtual environment
- Laser scanner messages, usually provided by the onboard laser scanner
- Wheel speed and steering angle based odometry, usually based on motor driver state information
- Inertial measurement readings, usually provided by the onboard inertial measurement unit
- Camera frames, usually provided by the onboard ZED camera

When I started working with the RACECAR, the above features were provided by Gazebo [4], a dedicated graphical physics-based robot simulator. While this system worked and was highly featured, we found that it was excessively computational for the task at hand, especially since it typically entailed concurrently running RViz and Gazebo - two heavyweight graphical applications. This problem was exacerbated by the fact that many researchers and students working with the RACECAR run the development suite inside a Linux virtual machine on portable laptop hardware. In practice, many users were seeing simulation rates of less than 5Hz, which is unacceptable from a development standpoint.

Not only was the Gazebo-based system highly computational, it was also lacking in flexibility since the car dynamics model could not be easily changed. We also encountered strange bugs that would occasionally require restarting the simulator after the car got "stuck" or somehow flipped upside down.

To address these performance and usability concerns, I wrote a new simulator from scratch in the Python programming language and the Rospy ROS API. This new simulator is, in practice, a fairly standard ROS node, it just happens to provide most of the facilities that would normally be provided by the real car’s sensors or drive train. Since the code is a standard ROS node, it is familiar and simple to modify or configure,
which makes this new simulator useful for ever-changing research tasks. Since this simulator sits at a lower level than our teleoperation (manual joystick control) and multiplexer nodes, those features of the core RACECAR software architecture work in simulation "out of the box".

The new simulator has proven itself very useful both in research and in the Spring 2018 iteration of 6.141, in which the simulator was included in the software development suite provided to approximately 80 students.

7.1 Pose Simulation

For maintaining the car’s position and orientation, rather than using a full physics simulator, I elected to use the same type of simplified dynamics models as discussed in Section 3. At startup, the virtual car is initialized to some configurable start state. From there, the chosen dynamics model is used to forward simulate the car’s state as a function of received control input and prior state via Euler integration. Multiple dynamics models have been implemented in the simulator, including the models discussed in Section 3, and slight variants thereof.

To visualize the car’s position, the simulator publishes a transformation frame between the world frame and the car’s "base_link" coordinate frame. Publishing this transformation allows researchers to visually inspect the car’s position by adding a polygonal RACECAR model (included in the original simulator) to RViz. This functionality is demonstrated in Fig. 7-1.

7.2 Laser Scanner Simulation

The LiDAR scanner is one of the most useful sensors onboard the RACECAR, and as such, it is important to be able to faithfully reproduce these measurements in simulation. To achieve this goal, the simulator uses ray casting (see Section 5) from the car’s simulated pose into the provided environment occupancy grid map. The computed range measurements are converted into a ROS "sensor_msgs/LaserScan"
message and published on the configured LiDAR topic.

7.3 Inertial Measurement Unit Simulation

Inertial measurement sensor readings are simulated for arbitrary dynamics models using finite differences on the car's pose at consecutive simulation timesteps. Finite difference accelerations are combined with a simulated gravitational force, and the result is published as a ROS "sensor_msgs/Imu" message.

7.4 Collision Detection

It is important that in simulation, the car is not able to simply drive through obstacles in its environment, since that would be deeply unrealistic. To accomplish detect collisions, we consider a 3D RACECAR configuration space [34] using a rectangular model of the car and the provided simulation environment map. This precomputed configuration space provides a computationally inexpensive collision check function that may be referenced at each simulation timestep. If a collision is detected at a particular timestep, the car's velocity is set to zero. Additionally, to prevent the car from "getting stuck" in a wall, the dynamics model evaluation which resulted in the collision is undone so that the car remains in free space. These collision mechanics are not perfectly accurate (for example, it doesn't consider rebound effects), but it works well in practice.

7.5 Extra Features

To aid in algorithmic analysis, we implemented the following additional features:

- Manual drag, rotate, and place tools for graphical car positioning via RViz
- Full simulation state publishing (useful for working with high-dimensional dynamics models like the 7D model from Section 3.3.1)
- A ROS interface for algorithmically setting (teleporting) the car's full state
- Publishing a transformation frame which visualizes front wheel steering angle
Chapter 8

Education with the RACECAR

My first foray into the world of RACECAR development was through an MIT Independent Activities Period (IAP) program which involved racing the cars around the Stata basement loop. This program was critical to my personal development, since it was my first taste of what would become a two year adventure. Acknowledging the importance of this kind of program in introducing students to robotics research, I have remained involved in the teaching side of the RACECAR, acting as a teaching assistant for MIT’s course 6.141.

8.1 6.141 ’Robotics: Science and Systems’

Figure 8-1: Spring 2018 6.141 Class photo.

"Robotics: Science and Systems", also known as 6.141, or RSS, is a lab class at MIT focused on the RACECAR. I was involved in 6.141 first as a student in Spring
2016, and then again as a teaching assistant in Spring 2017 and 2018. Throughout the class, teams of five or six learn about general robotics concepts, and apply those concepts both in simulation and on the real car. Many students who take 6.141 go on to pursue higher education or employment in robotics.

8.2 Other RACECAR-based Programs

In addition to 6.141, the RACECAR makes appearances in other programs including the Lincoln Labs BeaverWorks summer program for high school students and an IAP course at MIT. My first interaction with the RACECAR was during the IAP course in January of 2016.

All of these educational programs are important because in addition to introducing students to an high-quality research platform, they provide a testbed and use case for RACECAR software development. Many of the software components discussed here were developed in support of educational programs, and were improved via exposure to students.
Chapter 9

Supplemental Material

9.1 Development Tools

Throughout this thesis, several development tools proved invaluable. Here, I would like to briefly discuss my development setup in hopes that current and future roboticists may be inspired to refine their own workflows.

Sublime Text

Sublime Text has been my editor of choice for several years. I highly recommend it! The ability to control multiple cursors has proven extremely useful time and time again. To anyone who uses Sublime, who has not yet learned all the "hidden" features, I recommend reading the book *Sublime Text Power User* ([https://sublimetextbook.com](https://sublimetextbook.com)) cover to cover. Here’s a short list of Sublime plugins that I have found extremely useful:

- Predawn theme (with Monokai color scheme and small tabs), since it is so easy on the eyes: [https://github.com/jamiewilson/predawn](https://github.com/jamiewilson/predawn)

- Origami, since it allows arbitrarily tiled text editing windows: [https://github.com/SublimeText/Origami](https://github.com/SublimeText/Origami)
The Robot Operating System (ROS)

Needless to say, ROS has been integral to many of the projects discussed in this thesis. ROS is an excellent open source project, with a wealth of associated tooling (such as RViz) maintained by a large community of roboticists.

Numpy, Scipy, and Matplotlib

Without these, I would be lost, and my Python programs would be slow. Numpy is an excellent package which adds support for arbitrarily-dimensioned matrices. Numpy also serves as the basis for a large variety of high-quality packages such as Scipy and Matplotlib, which implement many useful tools for scientific computing and visualization. Numpy, and the associated ecosystem, has served me well as a Matlab alternative throughout my thesis.

SSHFS

In robotics, we often must work with source code on a variety of networked computers. My personal solution to this problem has been to mount the various remote systems locally via SSHFS, allowing me to access various remote files using my native text editor. Whenever working with a RACECAR, my first step was generally to mount its file system with SSHFS.

Flamegraphs

Figure 9-1: Example Flame Graph of the particle filter developed here.

Flame Graphs are an excellent visual tool developed by Brendan Gregg \(^1\) for debugging a program’s runtime performance. We have created an online guide for

\(^{1}\)http://www.brendangregg.com/flamegraphs.html
creating Flame Graphs in Rospy\textsuperscript{2}, based on the method we used in profiling and optimizing the particle filter implementation described here.

**Callgrind + KCachegrind**

These two tools have been my dream team for debugging and improving the runtime performance of my C++ programs. There are alternative profiling tools for C++, but I can recommend this combination of tools as highly usable.

**i3 Window Manager**

i3WM\textsuperscript{3} has been my Linux window manager of choice for a few years. It is lightweight and easy to use, which makes it excellent for running on the various operating systems that I must run on resource constrained systems, especially those without access to GPUs for window transparency composting.

**VMWare Fusion**

VMWare Fusion \textsuperscript{4} (provided courtesy of MIT IS&T) allows me to run my robotics development virtual machine on my host operating machine of choice. While this is certainly not the only virtualization solution, it has worked for me.

**GNU/Linux**

This list would not be complete without including the Linux kernel and the GNU toolset. These systems are critical in most modern robotics systems, both for runtime execution and system development tasks, and the RACECAR is no different. Many thanks to the hard-working people who contribute to making the Linux ecosystem what it is today.

\textsuperscript{2}https://mit-racecar.github.io/2017/03/21/flame

\textsuperscript{3}https://i3wm.org

\textsuperscript{4}https://www.vmware.com/au/products/fusion.html
9.2 Videos

Over the course of this thesis, I created several videos demonstrating the various functionality discussed here. These videos are available on my website (http://rayban.vision/#videos). Here’s a list!

- Drifting the RACECAR:
  http://rayban.vision/#drifting_video

- Autonomous Parallel Parking with Value Iteration:
  http://rayban.vision/#parallel_parking_video

- Real-Time Particle Filter Robot Localization on the GPU:
  http://rayban.vision/#localization_video

- Fast Autonomous RC Car - Pure Pursuit Trajectory Tracking:
  http://rayban.vision/#pure_pursuit_video

- CDDT Video Abstract (submitted to ICRA 2018):
  http://rayban.vision/#icra_cddt_video

- Simulating the RACECAR:
  http://rayban.vision/#racecar_simulation_video
Chapter 10

Conclusion and Future Work

This thesis has presented several different topics which are essential in the development of autonomous systems, such as model identification, state estimation, control, and simulation. Additionally, a novel algorithm for two-dimensional ray casting has been presented. Towards my goal of drifting the RACECAR, a method for controlling the highly nonlinear system dynamics encountered at high slip angles has been developed and evaluated.

There are several areas which would be good candidates for future research. Foremost is the integration of our time-varying LQR trajectory tracking approach with onboard state estimation, which would remove the dependency on an expensive and stationary motion capture system, and would allow a much broader range of potential applications. Using onboard state localization would require a method for estimating pose derivatives with reasonably low amounts of noise and an end-to-end pose estimation latency under 50ms.

Additionally, it would be interesting to investigate the use of machine learning in system identification and/or control. One idea that comes to mind is formulating a dynamics model based on a library of observed motion capture data, perhaps via K-nearest neighbors lookup [35]. Another idea is to combine the existing time-varying LQR controller with a learning based module which may discover and compensate for the systematic errors present in the original dynamics model, as proposed in [30].

There are several features which would be good additions to the developed par-
ticle filter code. For example, some form of global localization (perhaps as used in Mixture-MCL [11]) would allow the system to recover from tracking divergence without requiring manual intervention.

To improve our novel ray casting method, it would be useful to investigate computationally inexpensive methods of pruning the CDDT data structure, especially methods which would be compatible with online map modification. Faster initialization of the pruned CDDT variant would increase the method’s applicability.

A useful extension of the RACECAR simulator proposed here would be to simulate color and depth camera images, available on the real platform via onboard camera systems. This would allow use of the simulator in developing camera based algorithms, which is especially important in RACECAR hardware variants which do not include a laser scanner.

I hope that this thesis serves as a useful reference and guide for future researchers.
Bibliography


[33] The Matrix. Film, 1999. Lana Wachowski (Director), Lilly Wachowski (Director), Joel Silver (Producer). Warner Bros.
