Predictive Modeling and Socially Aware Motion Planning in Dynamic, Uncertain Environments

by

Yu Fan Chen

Submitted to the Department of Aeronautics and Astronautics on August 20, 2017, in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Abstract

Advances in sensor technologies and computing power have spurred a surge of interest in autonomous vehicles, such as indoor service robots and self-driving cars. The potential applications of such vehicles are predicted to have far-reaching impacts on human mobility and the economy at large. While there has been significant progress in the past decade, reliable, fully autonomous navigation remains challenging, particularly in environments that entail frequent interactions with other dynamic agents. Specifically, safe and time efficient navigation may require (i) predictive modeling of agents with unknown intents (e.g., goals), and (ii) cooperative collision-free motion planning. These issues are not only hard research problems individually, but also tightly coupled since the nearby agents' motion could be affected by the vehicle's choice of action.

This work focuses on the interplay between prediction and planning, and presents novel algorithmic approaches while considering various challenges arising from perceptual and computational limitations. First, a motion modeling framework is developed, which learns from data a set of commonly exhibited local motion patterns and the associated transition probabilities. This framework is designed to work with real data from onboard sensors, such as noisy position measurements and fragmented trajectory tracks due to sensor occlusion. Second, a multi-query path planning algorithm is presented, which computes a domain-specific similarity metric by learning the map's geometry. The algorithm not only enables quick local re-planning in response to frequent changes in the environment, but also allows for finding homotopically distinct paths at the route level. Third, a method for decentralized multiagent collision avoidance is developed, which uses reinforcement learning to generate a computationally efficient policy that encodes cooperative behaviors. Moreover, this approach is extended to capture subtle human navigation norms, such as passing on the right and overtaking on the left. The proposed methods are tested on hardware, and are shown to enable fully autonomous navigation at the average human walking pace through a pedestrian-rich environment.

Thesis Supervisor: Jonathan P. How
Title: R. C. Maclaurin Professor of Aeronautics and Astronautics
Acknowledgments

Time flies. It’s hard to believe that five years had gone by as the memories from my first days at MIT are still fresh. While the journey is certainly arduous, which contained a few too many late night working sessions, it was also fun and exciting, particularly when robots sprang into life with the implementation of new ideas. It was a truly unforgettable journey, and I would like to thank those who helped make this experience rewarding and enjoyable.

First and foremost, I’m grateful to my advisor, Professor Jonathan P. How, for his guidance and support. Throughout my time as a graduate student, I’m struck by Jon’s the uncanny ability to ask just the right questions that would lead to a deeper understanding of a research problem. In particular, I recall many meetings where his questions would first gracefully point to flaws in my ideas, and then help me see the limitations of the existing literature. I’m grateful to Jon for teaching me the importance of finding/understanding the cause of a problem before jumping to a solution.

I would also like to thank my thesis committee, Professor Nicholas Roy and Professor Sertac Karaman, for their invaluable input and feedback. I’m grateful to my external evaluator, Dr. Andrea Censi, and my thesis readers, Dr. Miao Liu and Dr. Golnaz Habibi, for taking the time to go through the document and providing useful feedback to improve the clarity of the writing.

I am incredibly grateful to all my lab mates at the Aerospace Controls Lab for being there with me every step along the way. In particular, I would like to thank Justin Miller for patiently helping me with all the hardware problems, Michael Everett for building the Jackal and running the late night experiments, Miao Liu for teaching me many useful tools for statistical learning, Shih-yuan Liu for building a well-thought-out software architecture and helping me with my next career step, Mark Cutler for mentoring me when I was a junior member of the lab, Kemal Ure for helping me with many machine learning problems, Shayegan Omidshafiei for the memories from the conference trips, Brett Lopez for his sense of humor, Luke Johnson for his insights
into almost all research problems, Beipeng Mu for her high spirit and optimism, and Jack Quindlen for his passion and knowledge in anything that flies. I couldn’t have asked for a better group of friends.

I would like to thank my parents for their unconditional and unwavering support. They have always encouraged me to follow my passion even when it leads me thousands of miles away from home. I am also thankful to my twin brother, with whom I had many interesting conversations and shared a laugh about the PhD experience. After going to college in different countries, we were fortunate to be in the same city again for graduate school. It was great to see our relationship grew from childhood companionship (with a touch of rivalry) into a lifelong friendship.

To all my friends from MIT and UofT, thanks for being there through the ups and downs. Finally, I would like to thank Ford Motor Company for supporting my research.
Contents

1 Introduction .......................................................... 15
  1.1 Motivation ......................................................... 16
    1.1.1 Challenges ................................................ 17
  1.2 Literature Review ............................................... 19
    1.2.1 Predictive Motion Modeling ............................... 19
    1.2.2 Path Planning in Dynamic, Uncertain Environments .... 21
    1.2.3 Decentralized Multiagent Collision Avoidance .......... 23
    1.2.4 Socially Aware Navigation ............................... 25
  1.3 Summary of Contributions .................................... 27

2 Background .......................................................... 31
  2.1 Gaussian Processes ............................................. 31
  2.2 Sparse Coding .................................................. 33
  2.3 Diffusion Maps .................................................. 35
  2.4 Reinforcement Learning ........................................ 38
    2.4.1 Deep Reinforcement Learning ............................ 39
  2.5 Hardware Platforms ............................................ 39
    2.5.1 Hardware Setup ........................................... 40
    2.5.2 Perception System ....................................... 41
  2.6 Summary ........................................................ 44

3 Predictive Motion Modeling ....................................... 45
  3.1 Problem Formulation ........................................... 47
4.3.1 Path Planning for a Nonholonomic Vehicle .......................... 91
4.3.2 Path Planning on Multiple Floor Levels ......................... 92
4.3.3 Performance Comparison ........................................ 92
4.3.4 Hardware Experiment ........................................... 94
4.4 Summary ..................................................................... 96

5 Decentralized Collision Avoidance and Socially Aware Navigation in Dynamic Environments

5.1 Problem Formulation .................................................. 102
5.1.1 Sequential Decision Making ..................................... 102
5.1.2 Reinforcement Learning ......................................... 104
5.2 Collision Avoidance with Deep RL ................................. 106
5.2.1 Parametrization ...................................................... 106
5.2.2 Generating Paths Using a Value Network ..................... 107
5.2.3 Training a Value Network ....................................... 108
5.2.4 Incorporating Kinematic Constraints ......................... 111
5.3 Social Norms ............................................................. 113
5.3.1 Characterizing Social Norms .................................... 113
5.3.2 Inducing Social Norms ........................................... 115
5.4 Scaling Up to Multiagent Systems .................................. 117
5.4.1 Approach I: Using a Two-agent Value Network ............ 119
5.4.2 Approach II: Training a Multiagent Network ............... 119
5.5 Results .................................................................... 122
5.5.1 Computational Time ............................................... 122
5.5.2 Performance Evaluation of CADRL ......................... 124
5.5.3 Socially Aware Navigation of SA-CADRL ................. 127
5.5.4 Hardware Experiment ........................................... 130
5.6 Summary .................................................................... 133

6 Conclusions and Future Work .............................................
6.1 Future Work ............................................................. 139
**List of Figures**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Autonomous ground vehicles</td>
<td>16</td>
</tr>
<tr>
<td>1-2</td>
<td>Challenges for autonomous navigation in dynamic environments</td>
<td>17</td>
</tr>
<tr>
<td>1-3</td>
<td>Related work on decentralized multiagent collision avoidance</td>
<td>24</td>
</tr>
<tr>
<td>1-4</td>
<td>Related work on socially aware motion planing</td>
<td>26</td>
</tr>
<tr>
<td>2-1</td>
<td>Hardware platforms</td>
<td>40</td>
</tr>
<tr>
<td>2-2</td>
<td>Calibrating the onboard sensors</td>
<td>42</td>
</tr>
<tr>
<td>2-3</td>
<td>Block diagram for the perception system</td>
<td>43</td>
</tr>
<tr>
<td>3-1</td>
<td>Finding motion patterns from a trajectory dataset</td>
<td>46</td>
</tr>
<tr>
<td>3-2</td>
<td>Augmenting the activeness variables</td>
<td>51</td>
</tr>
<tr>
<td>3-3</td>
<td>Visualization of a motion pattern</td>
<td>63</td>
</tr>
<tr>
<td>3-4</td>
<td>Two real world trajectory datasets</td>
<td>67</td>
</tr>
<tr>
<td>3-5</td>
<td>Effects of the activeness variables in the ASNSC framework</td>
<td>68</td>
</tr>
<tr>
<td>3-6</td>
<td>Visualizing dictionary atoms</td>
<td>69</td>
</tr>
<tr>
<td>3-7</td>
<td>Metrics for evaluating prediction quality</td>
<td>70</td>
</tr>
<tr>
<td>3-8</td>
<td>Comparison between ASNSC and DPGP</td>
<td>70</td>
</tr>
<tr>
<td>3-9</td>
<td>Sensitivity analysis of the <em>thres</em> parameter</td>
<td>72</td>
</tr>
<tr>
<td>4-1</td>
<td>Dynamic elements in a mapped environment</td>
<td>76</td>
</tr>
<tr>
<td>4-2</td>
<td>Diffusion map as a coordinate transformation</td>
<td>80</td>
</tr>
<tr>
<td>4-3</td>
<td>Choosing the truncation parameter <em>k</em> when constructing of a diffusion map.</td>
<td>81</td>
</tr>
<tr>
<td>4-4</td>
<td>Effect of a time scale parameter on diffusion distance</td>
<td>82</td>
</tr>
</tbody>
</table>
4-5 Visualization of the diffusion distance .................................. 82
4-6 Path planning in a static environment ....................................... 85
4-7 Path planning in a dynamic environment ................................. 87
4-8 Finding paths from different homotopic classes ....................... 89
4-9 Path planning for a nonholonomic vehicle ............................ 91
4-10 A multi-floor domain .......................................................... 93
4-11 Path planning on multiple floors ......................................... 94
4-12 Performance statistics on random test cases ......................... 95
4-13 Autonomous navigation using diffusion maps ....................... 96

5-1 An unsafe scenario for interacting with a dynamic agent .......... 100
5-2 An RL policy ................................................................. 104
5-3 Convergence of a Deep RL policy ..................................... 108
5-4 Training trajectories generated by ORCA .......................... 110
5-5 Training the value network ............................................... 111
5-6 A rotational kinematic constraint ....................................... 112
5-7 Symmetries in multiagent collision avoidance ..................... 113
5-8 Indications of a navigation convention emerging from the CADRL policy 114
5-9 A norm inducing reward function ................................... 116
5-10 SA-CADRL policies exhibiting socially aware behaviors ...... 118
5-11 Multiagent trajectories produced by CADRL ....................... 120
5-12 A network structure for multiagent scenarios ..................... 121
5-13 SA-CADRL policies generalized to multiagent scenarios ....... 123
5-14 Performance comparison on a crossing scenario .................. 126
5-15 A four-agent test case ..................................................... 127
5-16 Performance comparison on random test cases ................. 128
5-17 Navigating around non-CADRL agents ........................... 129
5-18 Hardware testing environments .................................... 130
5-19 Hardware experiment snapshots .................................... 131
5-20 Visualization of a ground vehicle’s sensor data .................. 132
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Literature review on socially aware motion planning</td>
<td>27</td>
</tr>
<tr>
<td>3.1</td>
<td>Prediction performance comparison between ASNSC and DPGP</td>
<td>71</td>
</tr>
<tr>
<td>4.1</td>
<td>Computational time for finding diffusion maps</td>
<td>92</td>
</tr>
<tr>
<td>4.2</td>
<td>Performance statistics on random test cases</td>
<td>97</td>
</tr>
<tr>
<td>5.1</td>
<td>CADRL’s performance statistics</td>
<td>135</td>
</tr>
<tr>
<td>5.2</td>
<td>SA-CADRL’s performance statistics</td>
<td>135</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The past decade has seen significant improvements in sensor technologies and tremendous growths in computing power. For example, the emergence of 3D Lidars and inexpensive RGB-D cameras allows for accurately capturing depth information from the environment. Also, palm-sized mobile computing platforms can now deliver more than 1.5 Teraflops of performance (e.g., Jetson TX2). Such advances have spurred research interests in perception and planning algorithms, which in turn, led to increased levels of autonomy in mobile robots. For instance, utilizing the rich sensory information, robots can localize to within a few centimeters in known maps [1, 2], as well as detect and track a variety of objects [3–6]. Given such exciting progress in research, self-driving cars [7, 8] and service robots [9, 10] (Fig. 1-1) are now attracting lots of commercial interests for their potential to bring about profound changes to human mobility patterns. In particular, the former could drastically reduce traffic fatality and free up time from driving; while the latter could introduce new personal mobility options and delivery services by tackling the last-mile problem in transportation systems.

A common theme for operating such ground vehicles is that negotiating traffic would require interacting with other dynamic agents in the environment. For example, it is important for self-driving cars to take into account other vehicles’ motion, and for delivery robots to avoid colliding with pedestrians. While there has been impressive progress in the past decade, fully autonomous navigation remains chal-
Figure 1-1: Autonomous ground vehicles. (a) shows a Google’s self-driving car prototype navigating autonomously on the street. (b) shows a delivery robot from Starship Technologies navigating on a busy sidewalk.

Challenging, particularly in uncertain, dynamic environments cohabited by other mobile agents. This is because the other agents’ intents and policies (i.e., goals and paths) are typically not known, and explicit communication of such hidden quantities are often impractical due to physical limitations. Also, to operate safely and efficiently in environments with pedestrians, it is important to characterize the subtle navigation rules demonstrated by humans, such as passing on right and overtaking on the left.

This thesis aims to develop a unified strategy for autonomous robot navigation in uncertain, dynamic environments, with a focus on predictive modeling and cooperative motion planning.

1.1 Motivation

Due to the presence of other dynamic agents, autonomous navigation of a robotic vehicle entails (i) predictive motion modeling of agents with hidden intents (e.g., goals), and (ii) collision-free motion planning. These issues are not only hard research problems individually [11–16], but also tightly coupled since the other agents’ motion could be affected by the vehicle’s choice of action. For instance, a service robot yielding to its right might induce an incoming pedestrian to pass on the robot’s left. The prediction and planning steps amount to reasoning about the joint motion of a vehicle and its neighbors, a process which is intellectually challenging and computationally expensive. An example of a robot’s decision-making process is illustrated in Fig. 1-2.
Detecting pedestrians

Inferring hidden intents

Inferring paths / hidden policies

Finding a safe path / action

Figure 1-2: Challenges for navigating in dynamic environments. A robot needs to first identify pedestrians from its onboard cameras using computer vision techniques, and then infer each agent's intended goals and motion patterns. The other agents' predicted motion would be subsequently used for finding a safe path for the robot. Not only are the inference steps difficult since the pedestrians' intents (e.g., goals) are typically not known, the entire decision-making pipeline is also computationally challenging for achieving real-time performance.

Here, a vehicle would need to identify the nearby agents from sensor inputs, infer each agent's intended goal and predict the corresponding path, and choose an action that is safe with respect to the other agents' predicted motion. This work seeks to enable safe and efficient autonomous navigation by focusing on the interplay between prediction and planning, while considering various challenges arising from perception and computational constraints.

1.1.1 Challenges

The development of autonomous vehicles brings together knowledge from many different fields, such as mechanical design, machine perception, mapping and localization, and planning under uncertainty. While one objective of this work is to develop an autonomous hardware platform for validating algorithmic approaches, the primary focus is on the planning aspect. Specifically, this thesis aims to advance the state-of-the-art by tackling the following challenges.

**Predictive Motion Modeling** To plan safe paths in dynamic environments, an autonomous robot needs to anticipate future behaviors of the nearby agents. A
promising direction for generating long term prediction is to learn a set of possible intentions (e.g., common goal locations) and motion patterns from past observations [11,17], such as trajectories tracked using onboard sensors. Yet, learning from such datasets is challenging because they often contain a large portion of incomplete trajectory fragments due to sensor occlusion and limited field of view. Also, due to computational constraints, it remains difficult to produce real-time predictions for all nearby dynamic agents in crowded environments.

**Path Planning in Dynamic, Uncertain Environments** While path planning in static environments has matured in the past decade, it is still difficult to account for the uncertainty in the other agents’ predicted motion. Specifically, to avoid colliding with other agents, the state space for the path planning problem is often expanded to include a time dimension [14], which would lead to a significant increase in computational complexity. More importantly, uncertainty from other agents’ motion prediction often grows quickly in time, making a large portion of the state space untraversable (unsafe) for classical path planning algorithms. This would cause the freezing robot problem [15], where a robot sits still since nowhere is deemed safe. Thus, it is important to develop a principled, computationally efficient method for handling such predictive uncertainty, one that would enable safe motion planning without being over-conservative.

**Socially Aware Navigation without Explicit Communication** Human navigation conventions should be considered when operating autonomous robots, particularly in environments that require frequent interactions with pedestrians. Specifically, the robots’ choice of action can induce changes in the nearby pedestrians’ motion, thereby allowing for cooperative motion planning [15]. While humans navigate around each other efficiently and effortlessly, the underlying mechanisms are largely unknown. Furthermore, given hardware limitations, explicit communication of each agent’s intentions is often not possible between robots and pedestrians. Thus, interacting with humans in a socially compliant manner, one that accounts for social norms such as
passing on the right and overtaking on the left, is practically important and in-
telegually challenging. Also, cooperative behaviors require reasoning about the joint 
state of the robot and the surrounding pedestrians, a process which can be much 
more computationally intensive than path planning for the robot alone.

1.2 Literature Review

There exists a vast literature on the control and coordination of autonomous vehicles. 
As this work is focused on the decision-making aspects, the following provides a brief 
survey on the works related to the aforementioned challenges.

1.2.1 Predictive Motion Modeling

Motion and behavior prediction is critical for ensuring the safety of autonomous 
vehicles. Filtering-based methods [18–20] are the most commonly used approach for 
their simplicity and low computational cost. In particular, these methods typically 
assume simple dynamical models, such as constant velocity or constant acceleration, 
and make prediction by propagating the system dynamics forward in time. However, 
the application of filtering-based approaches is often restricted to motion prediction 
on short time scales because they do not account for (i) interaction between dynamic 
agents, (ii) environmental structures, and (iii) the agent's underlying intentions (e.g., 
goal).

To address the aforementioned limitations, researchers have explored augmenting 
the dynamical models with additional terms to account for interaction between pedes-
trians in crowded environments. In particular, various techniques from the multiagent 
collision avoidance literature have been used to model such interaction [21–24]. For 
instance, the extended social forces model (ESFM) [21] treats pedestrians as repelling 
electrical charges, which is similar to the potential field methods. Also, the Bayesian 
reciprocal velocity obstacle (BRVO) [22] algorithm integrates a geometric collision 
avoidance model with a filtering-based algorithm. Albeit having shown improvement 
in prediction accuracy to filtering-based methods, these interaction models do not
consider global movement patterns nor capture subtle human behaviors (e.g., social norms). Moreover, prediction performance often depends heavily on tuning a large set of parameters, which is a difficult process because the parameters can vary significantly for different individuals [21].

A data-driven approach is commonly adopted for applications that require motion prediction over longer time scales (i.e., more than five seconds). This class of algorithms is typically provided with a trajectory dataset, from which commonly exhibited motion patterns can be learned in an offline learning phase [11, 12, 17, 25–27]. The set of motion patterns captures environmental structures and commonly exhibited behaviors, and thereby can produce more accurate predictions over longer time scales than filtering-based methods [11]. Learning-based methods can be broadly classified into two categories, Markovian-based and clustering-based.

Markovian-based methods [25, 27–29] learn a state transition model from the training trajectories, typically with a pedestrian's current position as the observed variable and the intended goal position as the hidden variable. The goal positions (hidden variables) are usually specified by domain experts or can be learned through an inverse reinforcement learning framework [26]. Conditioned on the current position and the inferred goal position, predictions are made by rolling the Markov chain forward in time. Since Markovian models are only conditioned on the current state, they can generate poor predictions if different motion patterns exhibit long overlapping segments [17]. Also, Markovian-based approaches can be susceptible to measurement noise [11].

Clustering-based methods [11, 17, 30–33] seek to find commonly exhibited motion patterns in the environment by first grouping the training trajectories into a few clusters, and then fitting a predictive model to each cluster. Gaussian Process (GP) regression [34] has been used for learning motion patterns [12] modeled as two dimensional velocity flow fields. The application of GPs addresses many of the shortcomings of Markovian-based methods by explicitly accounting for observation history and measurement noise. Clustering-based approaches have been shown to produce more accurate predictions than Markovian-based methods [11].
The prediction accuracy of clustering-based methods hinges on the ability to find a good clustering configuration in the training dataset. For this reason, various similarity metrics [31] and clustering techniques have been proposed [32, 33, 35]. Due to limitations in previous pedestrian tracking technologies, prior work has generally focused on learning from datasets collected using either motion capture systems [11, 13] or fixed wall-mount cameras [31], whose domains are typically restricted to the sensors' field of view. This work shows, in Chapter 3, that applying clustering-based methods to larger domains could be difficult. In particular, clustering-based methods implicitly assume that there exists a good clustering configuration in the dataset. However, this assumption might not be true in large domains because trajectories can be similar in one region and different in another, leading to ambiguity when finding clustering assignments.

1.2.2 Path Planning in Dynamic, Uncertain Environments

Planning feasible, collision-free paths in mapped environments is fundamental to many robotics applications. Although a map is often provided, there can be frequent changes in the environment due to the presence of other mobile agents. For example, in office buildings, furniture can often be shifted due to people's daily activities. Also, the nearby agents' motion needs to be considered for avoiding collision. To navigate efficiently and safely in such environments, a robot needs to leverage the knowledge of the map to find good paths quickly, and to re-plan locally when changes are detected.

A common strategy is to use single-query motion planning algorithms for finding feasible paths, and then adapt to local changes in the dynamic environments [36–39]. For example, grid-based D*-Lite [36] first uses A* to find the optimal path based on the current state (knowledge) of the map, and keeps track of the distance to goal for the set of visited states. Upon detecting local changes, such as an obstacle, D*-Lite would intelligently update the distance to goal for a set of impacted states (a subset of visited states), therefore without needing to re-plan the entire path. Similarly, building on RRT [40], RRTX is a sampling-based algorithm that allows for efficient
updates to the search tree when changes are detected in the environment [38]. Yet, planning from scratch for each new query (i.e., setting a goal for the vehicle) can be inefficient for applications that require repeated, on-demand motion planning.

Multi-query methods seek to improve computational speed by learning from the map or from past experience. For instance, sampling-based roadmap methods [40] pre-compute a topological graph for a more concise representation of the map. In addition, using previously planned paths, experience-based methods can pre-compute a better heuristic [41,42] or construct a path library [43] to solve for new queries more efficiently. However, these methods tend to bias toward previously sampled points or explored paths, and do not explicitly consider uncertainty in the environment.

An alternative to experience-based methods is to learn a potential function from the map that estimates a pairwise cost-to-go, which in turn, can serve as a heuristic to speed up single-query search algorithms. While efficient to query, classical potential field methods can suffer from having many local minima [40], thus limiting their usefulness for motion planning in complex environments. Researchers have explored finding potential functions that do not have local minima [44], such as by solving a steady-state heat equation on the map [45]. However, these potential functions need to be solved individually for each query to a different goal. Hence, computing a potential function is often much more expensive than finding a feasible path, because the former solves for every state with respect to the goal. Yet, a good cost-to-go function can be valuable for purposes beyond just finding a feasible path. For instance, a cost-to-go function can help guide a safe local exploration strategy [46] in an unknown environment. More generally, motion planning in the vicinity (i.e., perception range) of a robot often requires more sophisticated models, such as accounting for interaction with pedestrians [15] and modulating the robot’s speed [47,48]. Having a potential function could be useful for guiding such local search methods. This work will present a method in Chapter 4 that pre-computes and stores a potential function for finding/re-planning safe paths quickly in dynamic environments.
1.2.3 Decentralized Multiagent Collision Avoidance

The aforementioned methods are mostly designed for handling static uncertainty, such as moved furniture and closed doors. However, they are not well suited for handling dynamic uncertainty, such as pedestrians' motion. If agents can communicate their intents, such as goals or desired paths, then multiagent path planning methods can be applied. For instance, centralized algorithms [49-51] have been developed for finding a set of jointly collision-free paths, and message-passing algorithms [52] has been adapted for resolving pairwise conflicts. However, in many scenarios, explicit communication cannot be reliably established, such as when considering human-robot interactions. This work considers scenarios without communication, thus requiring the vehicle to navigate around other agents with hidden intents.

Existing work on decentralized collision avoidance can be broadly classified into cooperative and non-cooperative methods, as illustrated in Fig. 1-3. Non-cooperative methods handle dynamic obstacles by adding a time dimension to the path planning problem [41,53,54]. In particular, given the other agents' intended paths (e.g., generated using methods described in Section 1.2.1), the robot would find a feasible time-parametrized path that is collision-free with the others. In addition, to account for the uncertainty in the other agents' predicted motion, risk-aware path planning algorithms have been developed [11,14]. Yet, these methods rely on domain experts to set the thresholds/costs on the level of acceptable risk, which can be difficult to tune and often induce over-conservative behaviors. More importantly, as shown in Fig. 1-3(a), non-cooperative approaches do not consider interaction, such that the robot's choice of action can affect the surrounding agents' motion. This can lead to the freezing robot problem [55], in which a feasible/safe path might not exist for the robot if the other agents are assumed to be agnostic to the robot.

Cooperative methods [15,16,54,56-60] have been developed based on the observation that both the robot and the surrounding agents should share the responsibility for resolving collisions. These methods can be further classified into two sub-categories, reaction-based and trajectory-based. Reaction-based methods specify a set of interac-
Figure 1-3: Related work on decentralized multiagent collision avoidance. A red agent travels toward its goal, as marked by red cross, while avoiding collision with the blue agents. Non-cooperative methods (a) find a path that is collision-free with the other agents' predicted paths. Cooperative methods (b,c) anticipate changes in the other agents’ predicted paths due to the vehicle's motion. Reaction-based methods (b) compute a one-step action and expect reciprocal actions from the other agents. Trajectory-based methods (c) compute a set of jointly feasible paths for all agents in the environment.

In comparison, trajectory-based methods [15,16,60] explicitly account for evolution of future states by planning/anticipating a set of collision-free paths for every agent in the environment. For example, the interactive Gaussian Process model [15] finds a set of smooth paths connecting each agent’s current position and intended goal, as shown in Fig. 1-3(c). Trajectory-based methods often induce smoother paths than that of reaction-based methods because the former have a longer planning horizon [16]. However, trajectory-based methods can be computationally expensive (planning paths for all other agents) and often require information that isn’t available (other agent’s intended goal). Moreover, due to model and measurement uncertainty, the other agents’ actual paths might not conform to the predicted paths, particularly beyond a few seconds into the future. Thus, trajectory-based methods also need to
be run at a high rate, which exacerbates the computation problem.

1.2.4 Socially Aware Navigation

The preceding section reviews common strategies for multiagent coordination, which often assume reciprocity, such that every agent would follow the same set of rules for avoiding collision. Thus, if a system has full control over the entire fleet of vehicles, it would be fine to adopt a set of precise, intricate geometric rules. However, when considering autonomous navigation in pedestrian-rich environments, we cannot assume that the surrounding people would follow the same geometric rules. Hence, to achieve socially aware motion planning, additional care is needed to model human navigation behaviors.

Existing work on socially compliant navigation can be broadly classified as either *model-based* or *model-free*. Model-based approaches are typically extensions of multiagent collision avoidance algorithms, with additional parameters introduced to account for social interactions [21,22,61-63]. For instance, as shown in Fig. 1-4(a), to distinguish between human–human and human–robot interactions, the extended social forces model [21,62] augments the potential field algorithm with additional terms that specify the repulsive forces (i.e., strength and range) governing each type of interaction. Model-based methods are designed to be computationally efficient as they often correspond to intuitive geometric relations; yet, it is unclear whether humans do follow such precise geometric rules. In particular, the force parameters often need to be tuned individually, and can vary significantly for different pedestrians [21]. Also, it has been observed that model-based methods can lead to oscillatory paths [16,64].

In comparison, model-free approaches aim to develop a policy that emulates human behaviors by matching feature statistics, such as the minimum distance to pedestrians. In particular, Inverse Reinforcement Learning (IRL) [65,66] has been applied to learn a cost function from human demonstration (teleoperation) [67–69], and a probability distribution over the set of joint trajectories with nearby pedestrians [16,60]. For example, IRL is used to learn a mapping from sensors’ (e.g., camera images) input features to a navigation cost, which is turn, is used to plan
Figure 1-4: Related work on socially aware motion planning. A red agent travels toward its goal (marked by red cross) while avoiding collision with the blue agents. To imitate human behaviors, model-based methods (a) introduce new terms into the geometric rules for collision avoidance, such as specifying the attractive force from the goal and the repulsive forces between nearby agents. Model-free methods (b,c) formulate the navigation problem in general decision-making frameworks, and then attempts to learn the underlying policy from human demonstration (i.e., tele-operation). Inverse RL methods (b) first compute navigation costs in vicinity of the robot, and then plan a path with minimum cost. Max entropy optimization (c) can be used to compute the probability distribution over jointly feasible paths for all agents in the environment.

(a) Model-based (b) Learn a cost function (c) Learn distr. over paths

Summary The preceding has reviewed the key components for autonomous navigation in dynamic, uncertain environments, and it is worth pointing out the close relationship between motion prediction and path planning. On the one hand, the other agents' predicted motion can be used as inputs in non-cooperative path plan-
Table 1.1: Socially aware path planning in presence of dynamic obstacles.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Account for interaction</td>
<td>no</td>
<td>–</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Rely on intent prediction</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Able to learn from data</td>
<td>no</td>
<td>yes</td>
<td>–</td>
<td>yes</td>
</tr>
<tr>
<td>Consider social compliance</td>
<td>no</td>
<td>yes</td>
<td>–</td>
<td>yes</td>
</tr>
<tr>
<td>Appl. to multiagent coll. avoid.</td>
<td>–</td>
<td>–</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Computational complexity</td>
<td>medium</td>
<td>high</td>
<td>low</td>
<td>high</td>
</tr>
</tbody>
</table>

...ning algorithms [41]; and on the other hand, the path planning/collision avoidance algorithms can be modified to generate predictions [62]. The coupling between the two components is precisely due to the need to model interaction among agents in the environment. Moreover, socially aware motion planning can be viewed as adapting decentralized collision avoidance techniques to model how human navigate/interact with each other. A summary of the aforementioned methods is provided in Table 1.1.

1.3 Summary of Contributions

This thesis seeks an integrated solution approach to improve the safety and efficiency of autonomous navigation in dynamic, uncertain environments. Specifically, this work presents novel motion modeling techniques, efficient socially compliant motion planning algorithms, and develops the necessary sensing capabilities for building a functional hardware platform. Also, attention will be paid to the interconnections between these issues, such as considering predictive uncertainty in the design of a
path planner. In summary, the main contributions of this thesis are as follows.

- The development of augmented semi-nonnegative sparse coding (ASNSC) [71], a feature-based motion modeling and prediction framework. ASNSC finds a succinct and intuitive representation of mobile agents' trajectories as a sequence of commonly exhibited motion patterns. Both the motion patterns and the transition dynamics between them are learned from datasets collected by sensors onboard autonomous vehicles. Also, ASNSC is designed to handle (i) fragmented trajectories due to sensor occlusion, and (ii) environments much larger than a stationary sensor's field of view, where intents cannot be easily identified as entry/exit points of the scene. As will be explained in Chapter 3, ASNSC combines the merits of the Markovian-based and clustering-based methods reviewed in Section 1.2.1.

- The development of motion planning with diffusion maps (MPDM) [72], a method for enabling efficient re-planning in dynamic environments. This work seeks to pre-compute and store a potential function that encodes the distance between any pair of states in the map. A naive approach to solve the all-pairs-shortest-path problem would require $O(|V|^2)$ storage, which could be formidable for complex environments. To overcome this limitation, MPDM finds an efficient representation of the domain by learning the map's geometry. Moreover, the potential function is used to guide local re-planning around moving obstacles, and fast re-routing in case of perceived congestion.

- The development of multiagent collision avoidance with deep reinforcement learning (CADRL) [64], a decentralized collision avoidance algorithm that does not require explicit communication. Recall anticipating interaction with nearby agents is essential for generating collision-free paths, but such interaction can be difficult to model and expensive to simulate in real time. This work develops a cooperative navigation rule through deep reinforcement learning, which effectively offloads the expensive online computation (anticipating the other agents' motion) to an offline learning procedure. Further, the model is ex-
panded to characterize human navigation conventions [73]. As will be described in Chapter 5, CADRL achieves the performance of trajectory-based methods while attaining the low computational cost of reaction-based methods reviewed in Section 1.2.3.

- Hardware demonstrations on autonomous ground vehicles. The motion prediction algorithm ASNSC is implemented on an electric vehicle for generating accurate, real-time predictions of the surrounding pedestrians’ paths. The socially compliant interaction model CADRL is integrated with the proposed motion planning algorithm MPDM and implemented on an indoor ground robot. The robot is shown to navigate fully autonomously in various environments that require frequent interactions with pedestrians.
Chapter 2

Background

This chapter provides a quick review of the main machine learning techniques used in this thesis. In particular, the following summarizes the main computational steps and the important theoretical properties for each of the tools. Also, two robotic vehicle platforms are described, which will be used to validate the algorithmic approaches developed in later chapters.

2.1 Gaussian Processes

Gaussian Process (GP) [34] is a statistical model commonly used for nonlinear regression. It has become popular in the past decade for its ability to handle noise in the input and characterize uncertainty in its prediction. Also, GP is nonparametric, such that the number of parameters (model complexity) can grow with data. More formally, a GP is a collection of random variables, of which any finite subset is jointly Gaussian distributed. GP can be viewed as a probability distribution over functions $f : \mathbf{x} \to v$, that maps from $d$ dimensional vectors, $\mathbf{x} \in \mathbb{R}^d$, to a scalar output $v \in \mathbb{R}$, such that

$$
\mathbb{E}[f(\mathbf{x})] = m(\mathbf{x}) \tag{2.1}
$$

$$
\mathbb{E}[(f(\mathbf{x}) - m(\mathbf{x}))(f(\mathbf{x'}) - m(\mathbf{x'}))] = k(\mathbf{x}, \mathbf{x'}) + \sigma^2_n \delta(||\mathbf{x} - \mathbf{x'}||), \tag{2.2}
$$
where $m(x)$ is a mean function, $k(x, x')$ is a covariance function, and $\sigma_n$ is the measurement noise. The kernel function specifies the correlation between different input locations, and there has been a lot of research [34] on developing novel kernel functions to capture special structures (e.g., periodicity) in different regression problems. For simplicity and computational reasons, this work uses the square exponential kernel

$$k(x, x') = \sigma_s^2 \exp\left(-\frac{1}{2}(x - x')^T \Lambda^{-1}(x - x')\right),$$

(2.3)

where $\sigma_s^2$ is the signal variance and $\Lambda$ is a diagonal matrix which consists of the length scale parameters for each input dimension.

GP regression can be viewed as computing a posterior distribution from the prior (mean and covariance function) and data. Given data in the form of $n$ input-output pairs $D = \{(x_1, v_1), \ldots, (x_n, v_n)\}$, the joint distribution of the data with a new point $(x_*, v_*)$ can be written as

$$\begin{bmatrix} v \\ v_* \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \bar{v} \\ \bar{v}_* \end{bmatrix}, \begin{bmatrix} K + \sigma_n^2 I & k_* \\ k_*^T & k_{**} + \sigma_n^2 \end{bmatrix}\right),$$

(2.4)

where $\bar{v} = [m(x_1), \ldots, m(x_n)] \in \mathbb{R}^n$, $\bar{v}_* = m(x_*)$ are the mean function evaluated at the corresponding input locations, $K \in \mathbb{R}^{n \times n}$ is the data covariance matrix such that $K_{ij} = k(x_i, x_j)$, $k_* = [k(x_1, x_*), \ldots, k(x_n, x_*)] \in \mathbb{R}^n$ is the covariance between the data and the new point, and $k_{**} = k(x_*, x_*)$ is the variance at the new point. The regression problem can be solved for by finding the distribution at the new point conditioned on the data,

$$v_*|D \sim \mathcal{N}(\mu_*, \Sigma_*)$$

(2.5)

$$\text{s.t.} \quad \mu_* = \bar{v} + k_*^T (K + \sigma_n^2 I)^{-1}(v - \bar{v})$$

$$\Sigma_* = k_{**} + \sigma_n^2 - k_*^T (K + \sigma_n^2 I)^{-1}k_*.$$  

(2.6)

It is worth noting that GP regression has $O(|D|^3)$ computational complexity, because computing the posterior distribution requires inverting a matrix as shown
in (2.6). This could be a concern for many real-time applications, particularly when the number of data points is large. To address this issue, there has been various works on developing computationally efficient, sparse approximations of the kernel matrix [74,75].

### 2.2 Sparse Coding

Sensor technologies dictate how data is acquired and represented, which is often in a form not easily understandable/interpretable to human. For example, a digital camera registers an image as millions of pixel values, whereas human develops semantic descriptions, such as the type, position, relationship between different objects in the scene. Sparse coding is a class of algorithms that seeks to find a succinct, intuitive representation of the data. In this framework, each data sample $x_i \in \mathbb{R}^p$ is represented as a linear combination of a few basis vectors $d_k \in \mathbb{R}^p$, such that $x_i \approx \sum_k s_{ki} d_k$, where $s_{ki}$ is a set of sparse coefficients. In matrix form, the framework aims to solve the following optimization problem,

$$
\text{argmin}_{D,S} \|X - DS\|_F^2 + \lambda \sum_i \|s_i\|_r,
$$

(2.7)

where $X = [x_1, \ldots, x_l] \in \mathbb{R}^{p \times l}$ is the data matrix, $D = [d_1, \ldots, d_K] \in \mathbb{R}^{p \times K}$ is the dictionary matrix, $S = [s_1, \ldots, s_l] \in \mathbb{R}^{K \times l}$ is the corresponding coefficients, $\| \cdot \|_F$ is the matrix Frobenius norm, and $\lambda \sum_i \|s_i\|_r$ is a regularization term using the $l_r$ norm for $r = \{0, 1, \ldots\}$. Note that the goal is to solve for $D$ and $S$ simultaneously, which amounts to finding a representation in a new set of bases.

Attaining sparsity in $S$ is critical for achieving interpretability of the model. The obvious choice for imposing sparsity in (2.7) is to use the $l_0$ norm, which penalizes any nonzero coefficients. However, this is similar to the best subset selection problem, which is combinatorial in nature and computationally expensive to solve [76]. Thus, researchers have proposed using the $l_1$ norm [77–79], which has been viewed as a convex relaxation of the $l_0$ norm. More precisely, when $r = 1$, (2.7) is a bi-convex...
function, such that it is convex in \( S \) when holding \( D \) constant, and convex in \( D \) when holding \( S \) constant. Most algorithms [77–79] solve (2.7) by iterating between the dictionary and the coefficients.

Over the past decade, sparse coding algorithms have found successful applications in audio processing, image classification, segmentation, denoising, inpainting, and compression [77–79]. In fact, they are perhaps the best performing algorithms before the popularization of deep learning [80]. For example, sparse coding would typically learn an intuitive set of basis vectors, such as features corresponding to common geometric shapes (e.g., edges, lines) for representing images [79]. Much of sparse coding’s success has been attributed to the use of the \( l_1 \) norm and its tendency to induce sparse solutions. Yet, recent advances in optimization theory provide a new perspective to explain why sparse coding works well on noisy, real-world data. In particular, for solving the coefficients, it has been shown in [81] that

\[
\min_s \|x - Ds\|_2 + \lambda \|s\|_1 = \min_{s} \max_{\Delta \in \mathcal{U}} \|x - (D + \Delta)s\|_2, \tag{2.8}
\]

where \( \mathcal{U} = \{\Delta \in \mathbb{R}^{p \times k} : \max_{\xi} \frac{\|\Delta \xi\|_2}{\|\xi\|_1} \leq \lambda\} \) is an uncertainty set. By inspection of (2.7), when holding \( D \) constant, each column of \( S \) can be solved for independently and thereby individually. Thus, it suffices to show that solving for each column of \( S \) is equivalent to a robust linear regression formulation. Also, note the regularization parameter \( \lambda \) determines the magnitude of the allowed noise perturbation. Astute readers may have noticed that (2.7) and (2.8) are different by a square factor. In fact, it has been shown in [82] that these are equivalent up to a constant in the regularization factor,

\[
\min_s \|x - Ds\|_2 + \lambda_1 \|s\|_1 = \min_s \|x - Ds\|_2^2 + \lambda_2 \|s\|_1 \tag{2.9}
\]

for some \( \lambda_1 \) and \( \lambda_2 \). Overall, (2.8) establishes that the sparse coding formulation is effective in dealing with noise in the data.
2.3 Diffusion Maps

Diffusion map is a graph-based nonlinear dimensionality reduction technique [83] that has found many successful applications in data processing, such as clustering [84] and image inpainting [85]. For a high dimensional dataset, diffusion map seeks to find a lower dimensional manifold that captures major structures in the data. For instance, consider the 2D rendering of a 3D object rotated along two axes, where each data sample is an image of the object viewed at a distinct orientation. Given each image can have thousands of pixels, the task is to find a low dimensional parameterization that can effectively organize/visualize the relationship between the images. It is demonstrated in [83] that the diffusion map approach is able to find a two dimensional parametrization that captures the rotational motion – each dimension in the new representation corresponds to an axis of rotation.

The following summarizes the main steps in computing a diffusion map, and provides a sketch of proof for why does the mapping (new parametrization) induce a good similarity measure. Let \( G = \{ N, E \} \) be a bidirectional graph defined by a set of \( N \) nodes and up to \( N^2/2 \) edges, where each edge is also associated with a non-negative scalar weight \( A_{ij} \). An edge with a larger weight indicates that the two connected nodes are more similar to each other. The goal is to find a representation which maps each node to a new coordinate frame, \( N \mapsto \mathcal{Y} \), such that the distance in the new coordination frame \( ||y_i - y_j||_2 \) is a good similarity measure between nodes \( n_i \) and \( n_j \). Consider a random walk on the graph with a state transition matrix \( M \), where \( M_{ij} \triangleq P(s_{t+1} = n_j | s_t = n_i) = \frac{A_{ij}}{\sum_j A_{ij}} \). This can be written in a matrix form as \( M = D^{-1}A \), where \( D \) is a diagonal matrix with entries \( D_{ii} = \sum_j A_{ij} \). This work assumes that the graph \( G \) is connected and \( D_{ii} > 0 \) for all \( i \). Practically, this implies given sufficient time, that there is a non-zero probability to transition from any node to any other node.

Let \( S = D^{-\frac{1}{2}} A D^{-\frac{1}{2}} \) and note \( S \) is a symmetric matrix since \( A \) is assumed to be
symmetric. Thus, there exists an eigen-decomposition such that

$$ S = VAV^T, \quad (2.10) $$

where $V = [v_1, \ldots, v_N]$ forms an orthonormal basis of $\mathbb{R}^N$, and $\Lambda$ is a diagonal matrix with eigenvalues ordered from largest to smallest, that is $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_N$. Given this decomposition, the state transition matrix can be written as

$$ M = D^{-\frac{1}{2}}SD^{\frac{1}{2}} \quad (2.11) $$

$$ = (D^{-\frac{1}{2}}V)\Lambda(D^{\frac{1}{2}}V)^T \quad (2.12) $$

$$ \triangleq \Phi \Lambda \Psi^T, \quad (2.13) $$

where $\Phi = [\phi_1, \ldots, \phi_N]$ and $\Psi = [\psi_1, \ldots, \psi_N]$ are the left and right eigenvectors of $M$. Note that $\phi_i^T\psi_j = \delta_{ij}$ and therefore $M^t = \Phi \Lambda^t \Psi^T$, which implies the transition probabilities can be easily calculated for any $t$ steps into the future given the eigen-decomposition. Diffusion map associates the $i$-th node in the original graph with the $i$-th entry of each eigenvector; more precisely, the $K$-truncated diffusion map is defined as

$$ n_i \mapsto y_i = [\lambda_2^i \phi_2(i), \lambda_3^i \phi_3(i), \ldots, \lambda_K^i \phi_{K+1}(i)] \quad (2.14) $$

for $2 \leq K \leq N - 1$. Note that the eigenvector $\phi_1$ is omitted from the construction. This is because the largest eigenvalue in $\Lambda$ is 1, and the corresponding eigenvector is constant. The following shows that (i) 1 is an eigenvalue and (ii) all eigenvalues are less or equal to 1 in magnitude. First, $M$ is a state transition matrix and each row sums to one, that is $M1 = 1$, which implies that 1 is an eigenvalue with a corresponding constant eigenvector. Second, let $\xi_k$ be any right eigenvector and let the $j$th entry $|\xi_k|^j$ has the largest magnitude, then $|\lambda_k||\xi_k|^j = |M\xi_k|^j \leq |\xi_k|^j$. The inequality holds because $|M\xi_k|^j$ is a convex combination of $\xi_k$, which must be less than its largest component. It then follows that $|\lambda_k| \leq 1$ for any $k$.

The following theorem [84] states that diffusion distance – the Euclidean distance
in the new coordinate – is a meaningful measure of pairwise similarity.

**Theorem 1.** Consider a Markov random walk on the graph. Let \( s_t \in \{n_1, \ldots, n_N\} \) denote the state of the walk after \( t \) iterations, and let \( P_{hi}^t \) denote \( P(s_t = h|s_0 = i) \). When keeping the full set of eigenvectors by setting \( K = N - 1 \) in (2.14), the diffusion distance between two nodes is equal to a weighted distance between two probability clouds after \( t \) steps. Specifically, for any nodes \( n_i \) and \( n_j \),

\[
||y_i - y_j||_2^2 = \sum_h \frac{1}{D_{hh}} \left[ P_{hi}^t - P_{hj}^t \right]^2.
\]

(2.15)

**Proof.**

\[
RHS = \sum_h \frac{1}{D_{hh}} \left[ M_{hi}^t - M_{hj}^t \right]^2
\]

\[
= \sum_h \frac{1}{D_{hh}} \left[ \sum_k \lambda_k^t \phi_k(i) \psi(h) - \sum_k \lambda_k^t \phi_k(j) \psi(h) \right]^2
\]

\[
= \sum_h \frac{1}{D_{hh}} \left[ \sum_k \lambda_k^t (\phi_k(i) - \phi_k(j)) \psi(h) \right]^2
\]

\[
= \left\| \sum_k \lambda_k^t (\phi_k(i) - \phi_k(j)) D^{-\frac{1}{2}} \phi_k \right\|^2
\]

\[
= \left\| \sum_k \lambda_k^t (\phi_k(i) - \phi_k(j)) v_k \right\|^2
\]

\[
= \sum_k \left( \lambda_k^t (\phi_k(i) - \phi_k(j)) \right)^2 \quad b/c \ V \ is \ orthonormal
\]

\[
= ||y_i - y_j||_2^2 = LHS
\]

\( \square \)

The proof is reproduced here (from [86]) to show that the eigenvalue structure plays a significant role in the construction of diffusion maps. More importantly, this motivates only keeping \( K \ll N \) eigenvectors, as the remaining ones quickly decays to
zero given the $\lambda_k$ term in (2.14). More formally, it has been shown in [86] that the $K$-truncated diffusion map is the best $K$-dimensional approximation of the full diffusion map, in the sense of the average approximation error of diffusion distance (2.14). This permits an efficient approximation that leads to significant computational savings, because computing all the eigenvectors has $O(N^3)$ complexity, whereas finding the first $K$ eigenvectors has $O(KN^2)$ complexity. More computational details, such as how to choose $K$ in relation to motion planning, will be provided in Section 4.1.3.

2.4 Reinforcement Learning

Reinforcement learning (RL) [87] is a class of machine learning methods for solving sequential decision making problems with unknown state-transition dynamics. Typically, a sequential decision making problem can be formulated as a Markov decision process (MDP), which is defined by a tuple $M = (S, A, P, R, \gamma)$, where $S$ is the state space, $A$ is the action space, $P$ is the state-transition model, $R$ is the reward function, and $\gamma$ is a discount factor. The reward function is often assumed to be bounded, such that $R(s, a) \in [0, 1]$; and the state-transition model prescribes the probability of reaching a next state given the current state and action, such that $P(s', s, a) = \text{Pr}(s' | s, a)$.

Solving a RL problem amounts to finding a policy $\pi : S \mapsto A$ that maximizes a discounted cumulative reward known as the value function,

$$V^\pi(s) = \mathbb{E} \left[ \sum_{t=0}^{T} \gamma^t R(s, \pi(s)) \right].$$

The optimal value function can be solved by enforcing the Bellman Optimality condition [87],

$$V^*(s) = \max_a \mathbb{E} \left[ R(s, a) + \gamma V^*(s') \right],$$

where the expectation is over the set of possible next states. This relation reveals
the recursive structure in the problem, which inspires dynamic programming-based solution approaches such as value iteration [87]. Furthermore, the optimal policy can be retrieved from the optimal value function,

\[ \pi^*(s) = \arg\max_a \mathbb{E}[R(s,a) + \gamma V^*(s')] \]  

(2.18)

2.4.1 Deep Reinforcement Learning

Solving RL problems requires parametrizing the policy and the value function. The most common approach is to discretize each dimension and adopt a tabular representation of the state space. For instance, the 2D position of a robot can be represented as the Cartesian product of the two physical dimensions, \( S = X \times Y \). A major challenge to this approach is the curse of dimensionality, for which the size of the state space grows exponentially in the number of dimensions. More importantly, real-world decision making problems often reside in high (tens to hundreds) dimensional spaces, making it impractical to even enumerate, let alone computing the value, for each distinct state. For this reason, application of RL has been generally limited to simple, low dimensional domains.

Recent breakthroughs in deep learning [80, 88] have shown that deep neural networks (DNNs) can be trained to represent complex functions in a memory-efficient manner. Specifically, a DNN is parametrized by a large set (thousands to millions) of hidden units \( \theta \), which can be tuned through back-propagation to fit any function. Recent works have demonstrated that integrating RL with DNNs – thus named deep reinforcement learning (DRL) – can achieve human-level performance on video games [89], and even defeat the best human player in Go [90]. More details on DRL, particularly in relation to motion planning, will be provided in Chapter 5.

2.5 Hardware Platforms

The followings describe the hardware setup and the perception capabilities of two robotic vehicles developed in the Aerospace Controls Lab. The vehicles' design is
the joint effort between many members in the lab, particularly Justin Miller, Michael Everett, Andres Hasfura, Shih-yuan Liu, and myself. These vehicles are used to test the planning algorithms developed in later chapters.

2.5.1 Hardware Setup

A first generation ground vehicle was developed based on the Pioneer 3-AT platform as illustrated in Fig. 2-1(a). The vehicle is equipped with a 2D SICK Lidar for mapping and localization, a 3D Velodyne Lidar for obstacle detection, two Kinect RGB-D cameras for monitoring the Lidar’s blind spots, and three cameras for object recognition. A Gigabyte Brix computer is used for onboard processing of sensor inputs. Note this robot is standalone and does not rely on any external motion capture system.

While the first generation vehicle is capable of navigating autonomously (see Section 4.3), it is a bit too slow relative to people’s average walking pace. In particular, given its approximately 1.4m of height and 50kg of weight, the vehicle can travel...
up to 0.7m/s before starting to wobble. This motivates the development of a second generation vehicle shown in Fig. 2-1(b), which is based on a smaller, more agile Clearpath Jackal robot. Notice the footprint of the sensor suite is reduced substantially on the Jackal. For instance, the SICK Lidar is removed and the Kinect sensors are replaced with Realsense cameras that are about ten times smaller. Overall, the second generation vehicle is about half the size of the first generation, and can operate at a top speed of 2.0m/s. In comparison, most pedestrians walk around 1.2m/s [91]. This allows for testing human-robot interaction, as will be explored in Chapter 5. Interested readers are referred to [92] for a more detailed description of the robot’s hardware construction.

To provide some insights into the sensor selection and arrangement process, a visualization of the Jackal robot’s sensor inputs is provided in Fig. 2-2. In particular, for the environment illustrated in Fig. 2-2(a), the Lidar and camera inputs are shown in Figs. 2-2(b) and 2-2(c), respectively. The Lidar has a 50m range and a 360° field of view, and it is used as a baseline for calibrating the other sensors. The cameras have a 90° field of view, and they are positioned strategically for object detection. The precise pose of each camera is determined by manually aligning (superposition) objects (e.g., chairs) in the image with the Lidar pointcloud. Figure 2-2(a) illustrates a blind spot of the Lidar, where it could not detect the cardboard box in front of the vehicle. Thus, depth cameras are used to supplement the Lidar. In general, sensor arrangement is an iterative process that requires painstakingly identifying the system’s perception limitations, such as finding blind spots.

### 2.5.2 Perception System

The system’s software stack is built on the Robot Operating System (ROS) [93], which is a convenient framework for managing real-time communication (message-passing) between multiple processes. In particular, each software module can be updated independently as long as the input/output formats stay the same. Figure 2-3 provides an overview of the perception system’s organization, where the sensor inputs are shown as the solid black boxes on the left, and the processed outputs are
Figure 2-2: Calibrating the onboard sensors. (a) shows the picture of a scene taken by a person from behind. (b-c) show the sensor outputs. The colored pointcloud is generated by a Velodyne, the white pointcloud is generated by three depth cameras, and the images are from four onboard cameras. The sensors are calibrated by aligning the objects’ positions in each sensor’s frame. (d) shows an enlarged Lidar’s view of the cardboard box and the pole. Notice that the Velodyne has a 1m minimum sensing distance and thus could not see the cardboard box. The depth cameras are used supplement the Velodyne to detect objects close to the vehicle.

shown as the solid colored boxes on the right. The followings describe the methods used for computing each of the perception outputs.

**Mapping**  The ROS SLAM Karto package is used for constructing an occupancy grid map from a 2D Lidar scan, which is obtained by taking a slice of the 3D Velodyne pointcloud. Shown as the gray block in Fig. 2-3, the map is built by driving the vehicle manually through the environment. Once a map is constructed, it is used as a static input for other perception and planning modules.

**Localization**  The vehicle can localize itself by matching laser scans with the known map, as shown in brown in Fig. 2-3. Specifically, the vehicle updates its pose using a particle filter implemented in the ROS AMCL package.
Figure 2-3: Block diagram for the perception system. Black corresponds to the sensor inputs and gray is a static map generated beforehand. The blocks with a white background are the ROS nodes that process data, and the blocks with a solid color are the outputs of the perception system.

**Pedestrian tracking** The pedestrian tracking pipeline is shown in green in Fig. 2-3, which uses a combination of camera and Lidar. The raw Lidar pointcloud is first processed to remove the points that correspond to static objects in the map. Then, the dynamic-means algorithm [94] is used to cluster and track the remaining points. Independently, the cameras are used for identifying pedestrians with the VeryFast [95] detector. Since the cameras are calibrated (aligned) with respect to the Lidar, the camera detections can be used to label the pointcloud clusters. This allows for extracting the pedestrians' paths within the vehicle's perception range. Interested readers are referred to [96] for more details.

**Detecting free space** In dynamic environments, objects can shift in position over time. Thus, for collision-free navigation, it is important to detect the free space in vicinity of the vehicle, as shown in blue in Fig. 2-3. This work uses the ROS costmap_2d package for aggregating Lidar and depth camera pointclouds to populate an occupancy grid map in a 10 × 10m region centered at the vehicle. A visualization of the costmap is shown in Fig. 5-20.
2.6 Summary

This chapter provided some background information on the main algorithmic tools used in this thesis. In particular, details regarding Gaussian processes, dictionary learning and sparse coding, diffusion maps, and reinforcement learning were covered. Also, this chapter presented the hardware capabilities of two robotic vehicles. These tools will be used in the following chapters to tackle various challenges for autonomous navigation in dynamic, uncertain environments.
Chapter 3

Predictive Motion Modeling

Understanding the mobility patterns of dynamic agents is important for many robotics applications. For example, it can improve the safety of autonomous driving [97], reduce congestion in traffic [98], help detect anomaly in video surveillance [31], and increase the success rate in aerial interception missions [17]. As described in Section 1.2.1, Markovian-based and clustering-based methods are the two main types of approaches for modeling trajectories. The Markovian-based methods learn a state transition model from the training trajectories, and make predictions based on an agent’s current state and inferred hidden intent (e.g., goal). The clustering-based methods group the training trajectories into a few clusters, and make predictions by fitting a predictive motion model, such as a Gaussian Process [12], to each cluster. Using only the current state, Markovian-based methods can be more susceptible to measurement noise. In comparison, clustering-based methods have been shown to generally produce better prediction accuracies, but they can be slow to detect changes in an agent’s behaviors [11]. More importantly, clustering-based methods implicitly assume that the training dataset contains an underlying clustering structure, but this assumption might not hold. In particular, as illustrated in Fig. 3-1, there often exist long, overlapping trajectories in real datasets over large domains. Since such trajectories can be similar in some parts while different in others, it is ambiguous whether they should be assigned to the same cluster. Also, while existing work often assumes the ability to obtain complete trajectories, such as by using fixed wall-mount camera
Figure 3-1: Finding motion patterns from a trajectory dataset. (a) shows a synthetic trajectory dataset. (b) shows a clustering-based approach, where trajectories of the same color are grouped together. Notice the two green trajectories are similar in the upper left but different on the right, which illustrates that there can be inherent ambiguity when clustering long, overlapping trajectories. (c) shows a feature-based approach, where each trajectory is represented as a concatenation of local features. This work proposes a method for finding a feature representation of trajectory datasets.

with an unobstructed view of the scene [12], this work considers tracking dynamic agents using sensors onboard autonomous vehicles. Due to limited field of view and possible occlusion of the onboard sensors, there often exist many short, incomplete trajectory segments in such datasets. This creates a similar challenge as for overlapping trajectories, where it is conceptually unclear whether a good clustering structure exists in such datasets.

To address the shortcomings of the clustering-based methods, this work develops a part-based trajectory representation using sparse dictionary learning. In particular, this chapter introduces the augmented semi-nonnegative sparse coding (ASNSC) formulation, a flexible framework that allows for specifying different constraints on distinct parts of the dictionary. By using a novel parametrization of multivariate trajectories, the proposed method is able to learn dictionary atoms (features) that correspond to visually intuitive local motion patterns. It is interesting to note that finding a part-based representation can be seen as a generalization of the clustering-based approach. Specifically, a trajectory can exhibit multiple features (i.e., segments) in a part-based representation, whereas it is assumed/constrained to only exhibit one feature (i.e., belong to one cluster) in the clustering-based approach. If there is an oracle who can specify a good segmentation pattern for the trajectories, then the clustering-based approach can be used on the segments to find the similar local motion patterns as in the part-based representation. However, finding a good segmentation pattern is
difficult because it requires learning across all trajectories in the dataset to identify the similar parts (i.e., overlapping region). This work addresses this challenge by developing a dictionary learning method that simultaneously performs segmentation and clustering.

Noticeably, this work combines the merits of Markovian-based methods and clustering-based methods by finding the local clusters characterized by partial trajectory segments, and also the global Markovian transition dynamics between the clusters. Both the local motion models and the global Markovian transition dynamics are used for making motion predictions. While previous works have explored finding local trajectory clusters [99, 100], they were limited to modeling local motion patterns as short straight line segments; in comparison, this work is more flexible as the shape of local motion patterns is not constrained.

3.1 Problem Formulation

3.1.1 Augmented Semi-nonnegative Sparse Coding

As reviewed in Section 2.2, sparse coding is a class of algorithms that find a succinct representation of data by learning a set of overcomplete basis vectors. In this framework, each data sample $x_i \in \mathbb{R}^p$ is represented as a linear combination of a few basis vectors $d_k \in \mathbb{R}^p$, such that $x_i \approx \sum_k s_{ki} d_k$, where $s_{ki}$ is a set of sparse coefficients. Since many types of natural data (e.g., image pixel intensity) are non-negative, and motivated by the need to find interpretable part-based representations, researcher has developed non-negative sparse coding (NSC) [101] and non-negative matrix factorization (NMF) [102, 103] algorithms by imposing non-negativity constraints on the coefficient $s_{ki}$ and dictionary atoms $d_k$. However, few work has been focused on semi-nonnegative sparse coding, which allows $d_k$ to be unconstrained while maintaining $s_{ki}$ non-negative. We believe that this is not because of a lack of algorithms [104] to solve the problem, but an inherent issue with the problem formulation. In particular, the non-negativity of $s_{ki}$ is often imposed to achieve interpretability, such that a
non-zero coefficient would correspond to the activation of a basis feature. Yet, this problem formulation would allow dictionary atoms with opposite signs to cancel each other, thus making $s_{ki}$ difficult to interpret semantically. For instance, as illustrated in Fig. 3-2(b), it is ambiguous whether a trajectory exhibits two local motion patterns that add to cancel each other. It is plausible that an offset can be added to shift the entire dataset into the positive range, which then allows the application of NSC and NMF algorithms. However, this trick cannot be applied for some types of data, such as modeling a motion pattern as a velocity flow field. This is because the sign and magnitude of a velocity variable has different semantic meanings – heading directions and speeds – that can change after a constant shift. A detailed example will be presented in Section 3.1.2.

To address the aforementioned modeling problem, the following introduces the augmented semi-nonnegative sparse coding framework. For a set of $I$ data samples, $X = [x_1, \ldots, x_I]$, where $x_i$ is a column vector of length $p$, the goal is to learn a set of $K$ dictionary atoms, $D = [d_1, \ldots, d_K]$, and the corresponding non-negative sparse coefficients $S = [s_1, \ldots, s_I]$. More precisely, the objective is to solve

$$\arg\min_{D, S} \|X - DS\|_F^2 + \lambda \sum_i \|s_i\|_1$$

(3.1)

subject to

$$d_k \in Q, \quad s_{ki} \geq 0 \quad \forall k, i,$$

(3.2)

where $\lambda$ is a regularization parameter, and $Q$ is the feasible set in which $d_k$ resides.

This problem formulation generalizes prior work in several important ways. In particular, the non-negative sparse coding (NSC) problem [101] is obtained if $Q = \mathbb{R}^p_+$, and the non-negative matrix factorization (NMF) problem [102] is obtained by further setting $\lambda$ to zero, and the semi-nonnegative matrix factorization (semi-NMF) problems [104] is obtained by having $d_k$ unconstrained. The key difference from prior work is that (3.2) provides the flexibility to characterize different constraints for different parts of the dictionary. This can be very useful, for example, in cases where each data sample consists of two related measurement readings from different sensors. Consider a scenario in which a data sample can be constructed by stacking
up two column vectors – each corresponding to a measurement from a different sensor – such that \( x_i^T = [x_{i1}^T, x_{i2}^T] \). The corresponding dictionary atom can be partitioned similarly, such that \( d_k^T = [d_{ak}^T, d_{bk}^T] \), where \( d_{ak} \) and \( d_{bk} \) can be subjected to different constraints. More precisely, we can define \( Q \) in (3.2) such that

\[
\begin{align*}
d_{ak} & \in D_a, & d_{bk} & \in D_b, & g(d_{bk}, d_{ak}) & \leq 0 & \forall k,
\end{align*}
\]

where \( D_a \) and \( D_b \) are the feasible sets in which \( d_{ak} \) and \( d_{bk} \) reside, and \( g(\cdot, \cdot) \) is a joint constraint on \( d_{ak} \) and \( d_{bk} \). In real applications, \( D_a \) and \( D_b \) can be specified to reflect the properties, such as range and discretization level, of the corresponding sensor measurements.

### 3.1.2 Trajectory Modeling

A ground vehicle is assumed to be able to localize itself within a known map, as well as detect and track other dynamic agents. In practice, this can be achieved by using simultaneous localization and mapping algorithms (SLAM) [1] and object detection techniques [4] on camera and Lidar inputs. Thereby, the vehicle can track other agents' trajectories, which for the \( i \)th agent, \( t^i = [p_{i1}, p_{i2}, \ldots, p_{in_i}] \) is a sequence of position measurements \( p_{ih} \in \mathbb{R}^2 \) taken at a fixed time interval \( \Delta s \). Depending on the time when an agent enters and leaves the field of view of the vehicle's sensors, the length (e.g., number of points \( m_i \)) of each trajectory can vary. The vehicle is assumed to have collected a dataset of \( n_t \) trajectories, \( \mathcal{T} = \{t^1, t^2, \ldots, t^{n_t}\} \), which captures the other agents’ typical movement patterns. An example of a trajectory dataset is shown in Fig. 3-4.

For predictive motion modeling, an agent's velocity is calculated using finite difference approximation, that is, \( v_{ih} = (v_{x,ih}, v_{y,ih}) \approx (\frac{p_{x,ih+1} - p_{x,ih}}{\Delta s}, \frac{p_{y,ih+1} - p_{y,ih}}{\Delta s}) \). Since other agents can move at different speeds depending on environmental factors (e.g., traffic conditions and signals) and due to inherent variability in agility, it is often more important to model the heading direction which determines the shape of the possible paths. Thereby, this work focuses on modeling the shape of the trajectories.
by normalizing the magnitude of the velocities to one, that is, require \( v_x^2 + v_y^2 = 1 \).

Adapting to the dictionary learning framework, each trajectory \( t^i \) can be represented as a column vector \( x_i \) in the data matrix \( X \). In a domain discretized into \( M \times N \) blocks of width \( w \), this work computes the average x-y velocities \( (v_x^{mn}, v_y^{mn}) \) of each trajectory \( t^i \) going through each block \( mn \). Specifically, let \( g : \mathbb{R}^2 \rightarrow \{1, \ldots, MN\} \) be a mapping from 2D positions to grid location indices, then \( v_i^{mn} = \frac{\sum_h v_{ih} I_{mn}(g(v_{ih}))}{\sum_h I_{mn}(g(v_{ih}))} \), where \( I(\cdot) \) is the indicator function. If a trajectory does not go through a grid position, the corresponding velocities are default to zero. Further, a set of binary activeness variables \( a_i^{mn} \) is computed, such that \( a_i^{mn} = 1 \) if trajectory \( t^i \) goes through a grid position \( mn \), and \( a_i^{mn} = 0 \) otherwise.

A problem with this discretization scheme is that any point on the trajectory can be arbitrarily close to a grid line (boundary) and thus only count on one side but not the other. To address this issue, each point on a trajectory is replicated four times and shifted by \( w/2 \) in all directions. Figure 3-2(a) shows the effect of this shifting procedure, such that the activeness (non-zero) variables are more spread out. In terms of the ASNSC formulation in (3.3), let

\[
\begin{align*}
x_{ai} &= [v_{x1}^{11}, \ldots, v_{x1}^{MN}, v_{y1}^{11}, \ldots, v_{y1}^{MN}]^T \in \mathbb{R}^{2MN}, \\
x_{bi} &= [a_i^{11}, \ldots, a_i^{MN}]^T \in \{0, 1\}^{MN}.
\end{align*}
\]

An example of the discretized representation of a simple trajectory is illustrated in Fig. 3-2(a). Recall velocity is modeled to distinguish trajectories traveling in the opposite directions; and the non-negativity of sparse coefficients \( (s_{ki}) \) is enforced to achieve better interpretability, which would be important for the prediction step. A problem with using velocity in the dictionary learning framework is that dictionary atoms containing opposite velocity can cancel each other by addition, as illustrated in Fig. 3-2(b). This is undesirable because it can lead to learning noisy dictionary atoms and making the sparse coefficients less interpretable. In particular, the black data sample has a coefficient 1 corresponding to the green dictionary atom, but it clearly does not exhibit the green motion pattern. We address this issue by aug-
The vector representation of a trajectory

\[ \mathbf{x}_i = [v_x, v_y, a_i] \]

(a) The vector representation of a trajectory

(b) Effects of the activeness variables

Figure 3-2: Augmenting the activeness variables. (a) shows that in a domain discretized into \(M \times N\) blocks of width \(w\), a trajectory \(t_i\) is represented by a vector \(\mathbf{x}_i = [v_{x_i}, v_{y_i}, a_i]\), where \(v_{x_i}, v_{y_i}, a_i\) are the normalized x, y velocities, and the activeness variables, respectively. (b) shows that a data sample (black) on the left can be represented as a linear combination of the two dictionary atoms (green and red) on the right. Top row shows summing dictionary atoms (features) with opposite velocities can numerically fit the data. This is undesirable as it suggests the black data sample exhibits the green feature, which makes the sparse codes less interpretable. Bottom row shows that augmenting the positive activeness variables addresses this problem, because when velocities of opposite direction add to reduce error, the activeness variables would add to increase error.

The activeness variables, \(a_{im}\), which specifies whether a trajectory goes through a particular grid. When relaxed to be in the range \([0, 1]\), the activeness variables can be interpreted as the degree of confidence that a motion primitive (dictionary atom) goes through a particular grid. Further, the magnitude of the velocity components is constrained to be upper bounded by the corresponding activeness variable, that is, \(|v_{x_i}^{mn}| \leq a_{im}^{mn}\), \(|v_{y_i}^{mn}| \leq a_{im}^{mn}\). This formulation addresses the cancellation problem because when velocities of opposite signs add to reduce error, the positive activeness variables would add to increase error, thereby discouraging combinations of dictionary atoms with opposite signs. Rewrite these conditions in terms of (3.3) of the ASNSC formulation, such that

\[ \mathbf{d}_{ak} \in \mathbb{R}^{2MN}, \quad \mathbf{d}_{bk} \in \mathbb{R}^{MN}, \]

\[ |d_{ak}[j]| \leq d_{bk}[j], \quad |d_{ak}[2j]| \leq d_{bk}[j], \]  

where \([j]\) denotes the \(j\)th element of a vector, and (3.7) is the coupling constraint.
Algorithm 1: Augmented Semi-Nonnegative Sparse Coding (ASNSC)

1. D ← 0, S ← 0
2. while not converged do
   3. \[ r_{\text{max}} \leftarrow \max_{x_i} ||x_i - Ds||_2 / ||x_i||_2 \] // add a new dictionary atom if needed
   4. if \( r_{\text{max}} > \text{thres} \) then
   5. \[ d_{\text{new}} \leftarrow \arg \max_{x_i} ||x_i - Ds|| \]
   6. \[ D \leftarrow [D, d_{\text{new}}] \]
   7. S ← constrQP(X, D)
   8. \[ D' \leftarrow \text{gradientDescent}(X, S) \]
   9. \[ D \leftarrow \text{projection}(D') \]
10. return D, S

between the velocity variables and the corresponding activeness variables.

3.2 Dictionary Learning

This section presents an algorithm to solve for the dictionary and the sparse coefficients, D and S, in the ASNSC framework as defined in (3.1). The next section will describe how D and S are used for motion prediction. The proposed algorithm is outlined in Algorithm 1, which solves the constrained optimization problem by iterating between three major steps - (i) learning the non-negative sparse coefficients given the current dictionary in line 7, (ii) updating the dictionary given the current sparse codes in line 8, and (iii) modifying the current dictionary to enforce the constraints in line 9. Furthermore, adapting to data complexity, the size of the dictionary (number of columns) is expanded incrementally. In particular, if a data column is poorly represented as indicated by a large residual, the data column is added to the current dictionary (lines 3-6). The addition of a new dictionary atom happens once in a few iterations to allow for the current set of dictionary atoms to converge, as indicated by changes in the objective value. The following details each of the three major steps.
3.2.1 Learning the Non-negative Sparse Coefficients

The sparse coefficients $S$ can be solved for when holding the dictionary $D$ fixed. In particular, note that each column $s_i$ is independent from each other, and can be optimized individually, that is,

$$\arg\min_{s_i} \| x_i - Ds_i \|_2^2 + \lambda \| s_i \|_1$$

subject to $s_{ki} \geq 0 \forall k, i.$

Given the non-negativity constraint, the $l_1$-norm can be replaced with a summation term, $\sum_{ki} s_{ki}$. This reduces (3.8) to a constrained quadratic program (QP), which can be solved using many off-the-shelf optimization packages. Lee et al. [79] developed the feature-sign (FS) algorithm for finding sparse coefficients without the non-negativity constraints. This work develops the non-negative feature-sign (NFS) algorithm, an adaptation of FS to accommodate for the non-negativity constraint. We derive a NFS update step and show that it strictly decreases the objective value, which allows for finding the optimum in a finite number of steps. NFS’s pseudocode and detailed convergence proof are provided in Section 3.2.4. This adaptation was compared with Matlab’s quadprog solver [105] using various optimization setting, and the results showed similar accuracy and better run time for solving (3.8).

3.2.2 Updating the Dictionary

The dictionary $D'$ can be solved for when holding sparse coefficients $S$ fixed. Since the dictionary is shared across all data samples, the dictionary atoms are solved jointly by minimizing the following,

$$f(D') = ||X - D'S||_F^2 + \eta \|D'\|_F^2,$$  

(3.9)

where $\eta \|D'\|_F$ is a regularization term. Let $D_{old}$ be the previous iterate, the dictionary can be updated by gradient descent, $D' = D_{old} - \alpha \nabla_D f$, where $\nabla_D f$ =
\[-2(X - D'S)S^T + 2\eta D' \] is the gradient direction, and \( \alpha \) is a scalar step size.

### 3.2.3 Enforcing Constraints on the Dictionary Atoms

The previous step updates the dictionary \( D \) without considering the constraint in (3.2). Here, we enforce the constraints by projecting each column of \( D' \) back onto the feasible space. Let \( d'_k \) be the \( k \)th column of \( D' \), update each dictionary column,

\[
d_k = \arg\min_{d_k} ||d_k - d'_k||_2
\]

\[
s.t. \quad d_k \in \mathcal{Q}.
\]

For trajectory modeling, (3.7) specifies that only the variables at same grid position, \((a_{mn}, v_{mn}^{xk}, v_{mn}^{yk})\), are jointly constrained; whereas variables at different grid positions are independent from each other since there is no joint constraint on \( a_{mn}^{h}, a_{h}^{hl} \) for \( m \neq h \) or \( n \neq l \). This observation allows for a simple projection operation by solving for the variables at each grid position independently.

### 3.2.4 The Non-negative Feature-Sign Algorithm

Recall from Section 3.2.1 that the each column of the coefficient matrix \( S \) can be solved individually. This section presents the non-negative feature-sign algorithm for finding each column in \( S \). For clarity of exposition, equation (3.8) is rewritten below in a slightly different form,

\[
\arg\min_s f(s) = ||x - Ds||_F^2 + \lambda 1^T s
\]

\[
s.t. \quad s_k \geq 0 \quad \forall k,
\]

where \( 1^T s = ||s||_1 \) since \( s_k \geq 0 \). Lee et al. introduced the feature-sign (FS) algorithm \cite{Lee}, whose insight is that if the sign of each entry at the optimal value \( s^* \) is known, then the optimal solution can be found easily. Specifically, knowing the active set \( U = \{k | s_k^* > 0\} \) leads to forming an unconstrained quadratic program (QP),
which permits an analytical solution. The FS algorithm repeats the execution of a feature-sign update step in a nested loop, for which each iteration is shown to strictly reduce the objective value while updating the active set. It is further established that the algorithm will converge to the optimum in a finite number of steps. Here presents the non-negative feature-sign algorithm (Algorithm 2) and show that it inherits the convergence property.

Let \( s^* \) be the optimal solution to (3.12), then the gradient vector needs to satisfy \( \nabla_s f(s^*)^T(s - s^*) \geq 0 \) for all nonnegative \( s \). Equivalently, the \( k \)th component of the gradient vector needs to satisfy \( (\nabla_s f(s^*))_k = 0 \) for \( s^*_k > 0 \), and \( (\nabla_s f(s^*))_k \geq 0 \) for \( s^*_k = 0 \). By taking the derivative of (3.12), we obtain the optimality conditions (a) and (b) shown in line 18-19 of Algorithm 2. Following the nomenclature in [79], a coefficient vector \( s \) is consistent with an active set \( U \) if \( s_k > 0 \) for \( k \in U \) and \( s_i = 0 \) otherwise. Further, let \( \hat{s} \) and \( \hat{D} \) denote a subvector and a submatrix of \( s \) and \( D \), which correspond to those entries belonging to the active set; and define \( \tilde{f}(\hat{s}) = ||x - \hat{D}\hat{s}||^2_F + \lambda^T\hat{s} \) to be the sub-objective. Note that \( \tilde{f}(\hat{s}) = f(s) \) for a consistent \( s \).

**Lemma 1.** If the current set of coefficients \( s \) is consistent with the active set \( U \), but the corresponding \( \hat{s} \) is not optimal with respect to the sub-objective \( \tilde{f}(\cdot) \), then the feature-sign update (line 10-13) is guaranteed to strictly reduce the objective \( f(\cdot) \).

**Proof.** Since \( \hat{s} \) is not optimal with respect to the sub-objective \( \tilde{f}(\cdot) \), solving the unconstrained QP (obtaining \( \hat{s}^{\text{new}} \)) in line 10 will strictly reduce the sub-objective value, that is, \( \tilde{f}(\hat{s}^{\text{new}}) < \tilde{f}(\hat{s}) \). If any entry in \( \hat{s}^{\text{new}} \) is less than zero, a line search is performed from \( \hat{s} \) to \( \hat{s}^{\text{new}} \) to obtain the first zero-crossing point \( \hat{s}^d \), and note \( \tilde{f}(\hat{s}^d) < \tilde{f}(\hat{s}) \) by convexity of the objective function \( f(\cdot) \). Denote the updated coefficient in either case \( \hat{s}^{\text{next}} \) (line 13 or 16). Since \( s \) is assumed be consistent with \( U \), and the feature-sign update maintains consistency (line 14), thereby establishes \( f(\hat{s}^{\text{next}}) < f(s) \).

**Lemma 2.** If the current set of coefficients \( s \) is consistent with the active set \( U \), and the corresponding \( \hat{s} \) is optimal with respect to the sub-objective \( \tilde{f}(\cdot) \), after expanding the active set in line 4, \( \hat{s} \) is no longer optimal with respect to \( \tilde{f}(\cdot) \), and the feature-sign
Algorithm 2: Non-negative feature-sign algorithm

1. coefficient vector $s \leftarrow 0$, active set $U \leftarrow \emptyset$

2. while condition (b) not satisfied do

3. From zero coefficients of $s$, select $i \leftarrow \arg \min_i \frac{\partial ||x - Ds||^2}{\partial s_i}$

4. Add $i$ to the active set $U \leftarrow U \cup \{i\}$

5. while condition (a) not satisfied do

6. // feature-sign step

7. Let $\hat{D}$ be a submatrix of $D$ containing only the columns corresponding to the active set.

8. Let $\hat{s}$ be a subvector of $s$ corresponding to the active set.

9. Find the analytical solution to the unconstrained QP for the active set.

10. $\hat{s}_{\text{new}} = (\hat{D}^T\hat{D})^{-1}(\hat{D}^T\hat{x} - \frac{1}{2}1)$ *use pseudoinverse if needed.

11. if $\hat{s}_{\text{new}}^k \leq 0$ for any $k$ then

12. line search to find $\hat{s}^d$, the first zero-crossing point from $\hat{s}$ to $\hat{s}_{\text{new}}$

13. Update coefficients $\hat{s} \leftarrow \hat{s}^d$

14. Remove zero elements from the active set $U \leftarrow U \setminus \{k|\hat{s}_k^d = 0\}$

else

16. $\hat{s} \leftarrow \hat{s}_{\text{new}}$

17. return $s$

18. Optimality condition (a): for nonzero coefficients ($i \notin U$): $\frac{\partial ||x - Ds||^2}{\partial s_i} = -\lambda$

19. Optimality condition (b): for zero coefficients ($i \in U$): $\frac{\partial ||x - Ds||^2}{\partial s_i} \geq -\lambda$

update (line 10-13) strictly reduces the objective $f(\cdot)$.

Proof. Let $i$ be the newly added index to the active set. After adding index $i$, condition (b) (line 19) is no longer satisfied with respect to the sub-objective $\tilde{f}(\cdot)$, thus establishing the first part of the lemma. To establish the second part of the lemma, it remains to show that if there is a zero-crossing between $\hat{s}$ and $\hat{s}_{\text{new}}$ (line 12), then the zero-crossing is not the current iterate, that is $\hat{s}^d \neq \hat{s}$. This would imply $\tilde{f}(\hat{s}^d) < \tilde{f}(\hat{s})$ similarly as lemma 1. Since $\hat{s}_k$ is strictly greater than zero for all $k \neq i$ as these indices are in the active set, showing $\hat{s}^d \neq \hat{s}$ is equivalent to showing $\hat{s}_{\text{new}}^i > \hat{s}_i$. By condition (a) in line 18, the $k$th component of the gradient vector is zero, that is $(\nabla_{s\tilde{f}})_k = 0$ for $k \neq i$. Since $\hat{s}$ is no longer optimal after expanding the active set by adding index $i$, the condition (b) in line 19 is violated by the $i$th component of the gradient vector, hence $(\nabla_s\tilde{f})_k = c < 0$ for $k = i$. Let $d = \hat{s}_{\text{new}} - \hat{s}$, and note it is a descent direction by convexity of $\tilde{f}$. Since $d$ is a descent direction, its dot product
with the gradient vector satisfies $\nabla_s \tilde{f}(\mathbf{d}) = cd_i < 0$. Since $c < 0$, we must have $d_i > 0$, which implies $s_i^{\text{new}} > \tilde{s}_i$.

**Theorem 2.** The non-negative feature-sign algorithm finds the optimum of (3.12) in a finite number of steps.

**Proof.** Lemma 1 and 2 show that the non-negative feature-sign algorithm strictly reduces the objective value for every feature-sign update (line 10-16). Every time condition (a) is satisfied, the solution $s_U$ corresponds to the global optimum of a particular active set $U$. Since function value strictly decreases, $s_U$ will not repeat. Since number of columns in $\mathbf{D}$ is finite, there is infinite number of the possible active sets and so the outer loop (line 2-16) repeats a finite number of times. The inner loop (line 5-16) would also repeat a finite number of times because the active set, which has a finite number of elements, always decreases in size (line 14).

### 3.2.5 Analysis

**Convergence Properties of ASNSC**

Let $C(\mathbf{D}, S)$ denote the objective function $\| \mathbf{X} - \mathbf{DS} \|_F^2 + \lambda \sum_i \| s_i \|_1 + \eta \| \mathbf{D} \|_F^2$. The following shows that Algorithm 1 will converge to a local minimum of the objective function.

**Lemma 3.** Given that $\mathcal{Q}$ is convex, for any $(\mathbf{D}_{\text{old}} \in \mathcal{Q}, S)$, the gradient-projection step (lines 8-9 in Algorithm 1) reduces the objective value, that is, $C(\mathbf{D}, S) \leq C(\mathbf{D}_{\text{old}}, S)$, where $\mathbf{D}$ is obtained by solving (3.9) and (3.11). Further, if $C(\mathbf{D}, S) = C(\mathbf{D}_{\text{old}}, S)$, then $\mathbf{D} = \mathbf{D}_{\text{old}}$.

**Proof.** Let $\mathbf{D}'$ be the solution to (3.9). Since $\| \mathbf{D}' - \mathbf{D} \|_F^2 = \sum_k \| d'_k - d_k \|_2^2$, solving for (3.11) is equivalent to finding the $l_2$ projection of $\mathbf{D}'$ onto the feasible space. Further, since $\mathcal{Q}$ is convex, lines 8-9 of Algorithm 1 is a gradient-projection step that is guaranteed to reduce the objective value [81]. Further, the gradient-projection update step returns the same $\mathbf{D} = \mathbf{D}_{\text{old}}$ if and only if $\mathbf{D}_{\text{old}}$ is a stationary point [81].

57
Theorem 3. Algorithm 1 converges to a local minimum of the optimization problem (3.1) if $Q$ is convex.

Proof. Given a data matrix with a finite number of columns, the algorithm would stop introducing new dictionary columns after a finite number of iterations (lines 3-6), which allows us to only consider lines 7-9 for showing convergence. Convergence can be established using the monotonic convergence theorem by showing that the feature-sign step (line 7) and the gradient-projection step (lines 8-9) both decrease the objective function $C(D, S)$. Let $(D_{old}, S_{old})$ be values from the previous iteration. The feature-sign algorithm computes $S$ given $(S_{old}, D_{old})$. Theorem 2 establishes $C(D_{old}, S) \leq C(D_{old}, S_{old})$, and Lemma 1 establishes $C(D, S) \leq C(D_{old}, S)$. If either equality holds, then the algorithm has converged; otherwise, the objective value strictly decreases, that is, $C(D_{old}, S_{old}) < C(D, S)$. Further, when the algorithm has converged to $(D, S)$, Lemma 3 implies that this solution is a stationary point.

We claim that the feasible space $Q$ as defined in (3.7) is convex. In particular, it is straightforward to verify algebraically that for any $d_1, d_2 \in Q$, a convex combination of $d_1, d_2$ also belongs to $Q$. The algebraic details of the proof are omitted for brevity. Thus, we can establish the following corollary by invoking Theorem 1.

Corollary 4. Algorithm 1 converges to a local minimum of the trajectory modeling problem characterized by (3.7).

Computational Complexity

For each of the three major steps of ASNSC (line 7-9) in Algorithm 1, the per iteration time complexity is provided below. Assume there are $n_t$ samples, each of which is a vector of length $p = 3 \times M \times N$, and the number of dictionary elements is $k_d$. For the problem considered in our paper, $p > n_t \gg k_d$.

In the nonnegative feature-sign (line 7 in Algorithm 1) update, the major computation loads are the matrix operations. For a dictionary matrix $D \in \mathbb{R}^{p \times n_t}$, it
requires calculating $D^T D$, $(D^T D)^{-1}$ and $D^T X$, which has complexity $O(k_d^2 p)$, $O(k_d^3)$ and $O(pk_d n_t)$, respectively. Overall, the time complexity for updating sparse codes is $O(pk_d n_t)$.

The gradient descent update (line 8 in Algorithm 1) requires computing the gradient information, in which the major computation loads are for calculating $XS^T$ and $DS$, which has complexity $O(pk_d n_t)$ and $O(pk_d n_t)$, respectively. The projection step requires solving for a $l_2$ minimization problem for each dictionary column and depends on the complexity of the convex set. For the trajectory modeling problem, the complexity of the projection step is $O(pk_d)$. Overall, the time complexity for updating the dictionary is $O(pk_d n_t)$.

Combining the three steps, ASNSC has computational complexity $O(pk_d n_t)$.

### 3.2.6 Implementation Details

**A Faster Dictionary Update Step** While the proposed dictionary learning step has good theoretical guarantee, the gradient-projection update can be a bit slow because it takes a sequence of small steps to find the local optimum. Since the algorithm alternates between solving for $D$ and $S$, we find empirically that it is often better to find a good dictionary quickly and move on to the next iteration, compared with finding the locally optimal dictionary $D$ for the current iterate of $S$. The following describes a Lagrange-dual-projection step that yields better computational speed at the expense of theoretical rigor.

Recall that the data matrix $X$ consists of $p = 3 \times M \times N$ rows and $I$ columns, which correspond to the dimension and the number of data samples, respectively. Due to domain discretization as described in Section 3.1.2, this parametrization of a trajectory typically has more dimensions than samples, that is $p \gg I$. This observation motivates using of the Lagrange dual approach [79] to solve (3.9), because the number of dual variables is much less than the number of primal variables. After solving for the diagonal dual variable matrix $\Lambda$, the dictionary can be reconstructed.
by computing

\[ D' = (SS^T + \Lambda + \eta I)^{-1} (SX^T). \]  \hspace{1cm} (3.13)

Compared with [79], the formulation in (3.9) contains an extra regularization term, \( \eta ||D'||_F \), whose effect can be understood by inspection of (3.13). In particular, since \( p \gg I \), the matrix \( SS^T + \Lambda \) can sometimes be singular. Compared with the gradient descent step, the Lagrange-dual update takes a large step and finds the unconstrained global optimum \( D^* \) of the objective function (3.9). Then, line search is performed from the previous dictionary \( D_{old} \) to the global optimum \( D^* \) to ensure that the update step would still be feasible. In practice, the two update steps can be interwoven to achieve faster convergence and still attain theoretical properties. Empirically, we found that this update rule can reduce time to convergence by approximately three times.

**Reducing data dimension** Recall in the ASNSC framework (Section 3.1.2), a trajectory \( t^i \) is represented as a vector \( x_i \) by discretizing the domain into \( M \times N \) grids. Thus, when considering the x-y velocity and the activeness variables, the data dimension is \( p = 3 \times M \times N \). Yet, in real-world domains, a large proportion of the grid cells often corresponds to static environmental features, such as walls, pillars, car lanes. Correspondingly, there would be no trajectories in the dataset that go through these grid cells. Hence, a preprocessing step is run to omit such extraneous grid cells when creating \( x_i \)'s, for which depending on the domain, can reduce the data dimension by 2-10 times.

### 3.3 Motion Prediction

The preceding section described a method for solving a dictionary \( D \) and a set of coefficients \( S \). The following explains how to make use of \( D \) and \( S \) to generate predictions for new observations.
3.3.1 Segmenting the Trajectories

Recall each data column is represented as a linear combination of the dictionary atoms, that is $x_i \approx \sum_k d_k s_{ih}$. For each data column, a membership vector, $q_i \in \{0,1,\ldots,K\}^{MN}$, can be computed to find the dictionary atom that best explains the data column in each of the grid positions. More specifically,

$$q_i[j] = \arg \min_k \|x_i^j - d_k^j s_{ikh}\|_2,$$

where $q_i[j]$ is the $j$th entry (grid position) of the membership vector, $x_i^j$ is the part (tuple of velocity and activeness variables) of the data column that corresponds to the $j$th grid position, and similarly for $d_k^j$. Intuitively, the membership variables partition the data column using the sparse coefficients.

Subsequently, the membership vector is used to segment the original trajectories. Recall from Section 3.1.2, each trajectory $t^i$ is a sequence of 2D positions, and $g(\cdot)$ is the mapping from positions to grid indices. For each point $p_{ih} \in t^i$, a cluster assignment is read off from membership vector,

$$c_i[h] = q_i[g(p_{ih})].$$

The cluster assignment vector $c_i$ effectively segments $t^i$ by assigning each point to one of the learned dictionaries. A visualization of a trajectory’s segmentation pattern is shown in Fig. 3-5(b).

3.3.2 Building a Transition Matrix

Astute readers might have noticed that the vector representation from dictionary learning loses the temporal information, which is the ordering of the points in a trajectory. Yet, this information is critical for finding the transition between the dictionary atoms. Thus, when computing the cluster assignment vector, the algorithm returns to the original trajectory representation – finding a $c_i$ for each $t^i$.

A transition matrix, $T \in \mathbb{Z}^{K \times K}$, can be built by counting the number of switches
in each cluster assignment vector \( c_i \). In particular, \( T_{kl} \) corresponds to the number of trajectories that transitioned from the \( k \)th dictionary atom to the \( l \)th dictionary atom. Subsequently, two types of motion patterns – unitary and transitional, which correspond to the diagonal and off-diagonal elements in \( T \) – can be constructed using the transition matrix. Specifically, each motion pattern is built from a triple, \( \langle p_{mp}, v_{mp}, d_{mp} \rangle \), which are the dictionary atom and the trajectory segments (positions and velocities) belonging to corresponding clusters. Section 3.3.3 will detail the construction of a motion pattern from these quantities.

For clarity, a small example is presented below. Consider the a dataset with three trajectories, for which Algorithm 1 learned three dictionary atoms, \( \{d_1, d_2, d_3\} \). Following the descriptions in Section 3.3.1, the corresponding cluster assignment vector is found as follows,

\[
\begin{align*}
t^1 & = [p_{1,1}, \ldots, p_{1,8}], & c_1 & = [1, 1, 1, 1, 2, 2, 2, 2], \\
t^2 & = [p_{2,1}, \ldots, p_{2,10}], & c_2 & = [1, 1, 1, 1, 1, 3, 3, 3, 3], \\
t^3 & = [p_{3,1}, \ldots, p_{3,4}], & c_3 & = [3, 3, 3, 3].
\end{align*}
\]

By inspection, there is one transition from \( d_1 \) to \( d_2 \), another transition from \( d_1 \) to \( d_3 \). Hence, a transition matrix can be written as,

\[
T = \begin{bmatrix}
0 & 1 & 1 \\
0 & 1 & 0 \\
0 & 0 & 2
\end{bmatrix},
\]

where the diagonal entries count the number of trajectories that end at the particular dictionary atom. From the transition matrix, it can be seen that there are three unitary and two transitional motion patterns. In particular, the unitary motion patterns can be built from

\[
\begin{align*}
MP^{uni}_1 & : \langle \{ p_{1,1:4}, p_{2,1:6} \}, \{ v_{1,1:4}, v_{2,1:6} \}, \{ d_1 \} \rangle > \\
MP^{uni}_2 & : \langle \{ p_{1,5:8} \}, \{ v_{1,5:8} \}, \{ d_2 \} \rangle >
\end{align*}
\]
Figure 3-3: Visualization of a motion pattern, which is composed of (a) a Gaussian Process for velocity prediction, and (b) the active region over which the training trajectories passed through. Subfigure (a) shows that GP regression fits the blue velocity field from the red training data. Given a new observation, shown as the black segment up to the green circle, the GP model generates a green predicted path that matches well with the actual path.

\begin{align*}
MP_{3}^{\text{uni}} : & \{p_{2,7:10}, p_{3,1:4}, v_{2,7:10}, v_{3,1:4}, d_{3}\}, \\
\text{and the transitional motion patterns can be built from} & \\
MP_{12}^{\text{tran}} : & \{p_{1,1:8}, v_{1,1:8}, d_{1}, d_{2}\}, \\
MP_{13}^{\text{tran}} : & \{p_{2,1:10}, v_{2,1:10}, d_{1}, d_{3}\}. \tag{3.16}
\end{align*}

3.3.3 Gaussian Process Motion Model

A motion pattern can be modeled as a two dimensional flow field. This work uses two independent Gaussian Processes (GPs) [12,17] to fit the x-y velocities. In particular, for x-velocity, a set of data pairs \(D_{x} = \{(p, v_{x})\}\) can be assembled from (3.16). Then, using the regression technique introduced in Section 2.1, a mapping from 2D positions to the scalar x-velocity can be found, that is, \(GP_{x}(p) \rightarrow v_{x}\). The process is repeated similarly for the y-velocity. Thereby, a GP motion pattern is characterized by a pair \((GP_{x}, GP_{y})\), an example of which is illustrated in Fig. 3-3(a). Notice that although there is measurement noise in the training data (red points), the predicted output (blue arrows) is quite smooth.
A GP motion pattern can be used to generate a predicted path from an arbitrary 2D starting position $p'$. This involves invoking (2.6) repeatedly to find the expected velocity, $v \leftarrow GP(p')$, and then update the position $p' \leftarrow p' + v \Delta t$. An example of a predicted path is shown in green in Fig. 3-3(a). Furthermore, recall from Section 2.1, GP regression not only predicts a mean, but also a variance, and thereby characterizes a distribution. Thus, a GP motion pattern can also be used as a likelihood model for finding how well it does agree with a new observation (e.g., trajectory segment). In particular, for a new observation $t'$ of length $l$ converted to a sequence of position and velocity pairs $\{(p'_1, v'_1), \ldots, (p'_{l}, v'_{l})\}$, its likelihood of being generated by a GP motion pattern can be computed as

$$P(t'|GP_x, GP_y) = \prod_{p' \in t'} \mathcal{N}(v_x; \mu_{GP_x}(p'), \sigma_{GP_x}^2(p')) \mathcal{N}(v_y; \mu_{GP_y}(p'), \sigma_{GP_y}^2(p')),$$

where $\mathcal{N}$ denotes the normal distribution. This formulation treats each point on the new trajectory as independent from each other. By inspection of (2.4), it is also possible, and perhaps will generate more accurate results, to compute the joint probability of the every point on the new trajectory (without the independence assumption). However, finding the joint probability is more computationally expensive, and so this work uses (3.17). Interested readers are referred to [106] for further details, such as for tuning the hyper-parameters in a GP motion pattern.

A known issue [106] with the GP motion patterns described above is that GPs do not explicitly model the distribution of the training data $\{p \in D\}$ over $\mathbb{R}^2$. For example, as shown in Fig. 3-3(a), at a point (7,2) which is far away from any training data, the GP motion pattern still generates a confident (low variance) prediction. This can lead to being overconfident, such as using a motion pattern formed by trajectories on one side of a street to predict the motion of a pedestrian on the other side. There are also other important, yet more subtle factors that negatively affect prediction accuracy [106]. To address this issue, the model would ideally have an additional component that specifies the region over which the motion pattern is active.
Conveniently, the dictionary atoms found by ASNSC provides such information (Section 3.1.2). Specifically, the active region can be found by taking the union of the activeness variables from each dictionary atom in (3.16). A visualization of the active region (AR) is shown in Fig. 3-3(b). To this end, a motion pattern is defined by a triple $MP = < GP_x, GP_y, AR >$, and the likelihood model (3.17) is appended by an active region term,

$$P(t'|MP) = P(t'|GP_x, GP_y) \prod_{p' \in t'} b(I_{AR}(p'); \beta), \quad (3.18)$$

where $b(\cdot)$ is the Bernoulli distribution, $I_{AR}(p')$ is an indicator variable of whether a point $p'$ is inside the active region, and $\beta \in [0,1]$ is the coin-flip parameter.

In short, using the segmented trajectories and the learned dictionary atoms (3.16), this work constructs motion patterns as a pair of GPs and a corresponding active region. A motion pattern can be used to (i) predict a path by repeated forward propagation, (ii) find the likelihood of a new observation with (3.18), and (iii) find the likelihood of a predicted path (i.e., likelihood decreases when a predicted path goes outside the active region).

### 3.3.4 Generating Predictions

Following the steps above, a transition matrix $T$ and a set of motion patterns are constructed. Here presents a method for generating a set of predicted paths using these quantities. Consider a new observation $t'$, which is a path traced out by a pedestrian in the past five seconds. The algorithm would first determine the unitary motion pattern that most likely generated this observation as described in (3.18),

$$\hat{k} = \arg\max_k P(t'|MP_k^{uni}). \quad (3.19)$$

Then, a set of probable future actions (dictionary atoms) can be read off from the transition matrix, that is, $N = \{ j | T_{kj} > 0 \}$. Each of the transitional motion patterns $MP_{kj}^{tran}$ for $j \in N$ is used to generate a predicted path $s_j$, and the predicted paths are
sorted based on the likelihood $P(s_j \mid MP_{k_{ij}}^{\text{tran}})$. In addition, the likelihood information indicates degree of confidence in the predictions, which is useful for anomaly detection, such as when a pedestrian does not follow any motion patterns from the training set. An example of path prediction is shown in Fig. 3-8(a) and will be explained in more detail in Section 3.4.2.

3.4 Results

The proposed approach is evaluated on two trajectory datasets of different length scales. The first dataset consists of 147 taxi trajectories adapted from [107], each of which was converted to a vector representation of length 2484. Dataset I shows the importance of augmenting the activeness variables for dictionary learning. The second dataset consists of 72 pedestrian trajectories collected using a Velodyne Lidar, each of which was converted to a vector representation of length 375. Dataset II is used to show the improvement in prediction accuracy compared with existing methods. The algorithms were run on a computer with an Intel i7-4510U CPU and 16GB of memory.

3.4.1 Learning Traffic Patterns

The taxi dataset is used to demonstrate the importance of augmenting the activeness variables for dictionary learning. On this dataset, Algorithm 1 converged in 168 iterations within 33.1 seconds, and learned a 26 column dictionary as well as the associated sparse coefficients.

For brevity, only the first six dictionary atoms are illustrated in Fig. 3-5. Visualization of the dictionary atoms and the segmentation pattern of the training trajectories are shown in the top and middle subfigures, respectively. The cluster assignment vector (segmentation pattern), $c_i$, of one trajectory is shown in the bottom row of Fig. 3-5. Subfigures (a) and (b) correspond to solving (3.1) with classical semi-nonnegative sparse coding without augmenting the activeness variables, and with ASNSC, respectively. By augmenting the activeness variables, we addressed the
3.4.2 Making Predictions at an Intersection

On the pedestrian dataset, the ASNSC algorithm converged in 59 iterations within 4.98 seconds, and learned a 9 column dictionary as well as the associated sparse codes. The learned dictionary corresponds well to human intuition – each dictionary atom specifies a motion pattern that either enters or exits the intersection. This is quite remarkable because the algorithm works with generic 2D position data and
Figure 3-5: Effects of the activeness variables (AVs) in the ASNSC framework. (a) and (b) show without and with augmenting the AVs, respectively. A visualization of six dictionary atoms $D$, and the corresponding segments of the training trajectories, are shown in the top and middle subfigures, respectively. The bottom subfigure shows the segmentation of one training trajectory, with each color corresponding to a different dictionary atom (color is uncorrelated to the top and middle subfigures). Every trajectory in the training set is plotted in gray in the background.
Figure 3-6: Dictionary atoms found by solving ASNSC on dataset II. (a) shows that Algorithm 1 found 9 dictionary atoms, which are shown in groups of three. (b) shows the segmentation of training trajectories corresponding to the dictionary atoms.

has no semantic understanding of an environment, such as an intersection. All nine dictionary atoms and the corresponding projection are shown in Fig. 3-6.

Prediction accuracy is evaluated on a separate test set of 24 trajectories by comparing with a set of hand-labeled predictions, as illustrated in Fig. 3-7(a). In particular, given the observed path of a pedestrian going into an intersection (shown in black), the algorithm finds a set of possible future paths. Each predicted path is categorized as either correct (red), incorrect (green), missing or repetitive (orange).

Recall in clustering-based trajectory modeling approaches Section 1.2.1, given an observed path, predictions are made by using (i.e., forward propagation) the n clusters that agree the most with the observation. A problem of this approach is that in real datasets, there often exist many fragmented trajectories due to occlusion in the data collection process. Thus, the fragmented trajectories are often classified into their own clusters. Consequently, making predictions using these clusters can lead to poor results, as illustrated in Fig. 3-7(b). In contrast, this work models each trajectory as a concatenation of a few dictionary atoms, and makes predictions as described in Section 3.3.

ASNSC is compared against using Gibbs sampling for the Dirichlet Process mix-
Figure 3-7: Metrics for evaluating prediction quality. As a pedestrian enters an intersection, an algorithm makes predictions about the set of possible future paths the person may take. The predictions are evaluated against a hand-labeled set as ground truth. In (a), red shows the correct set of predictions, green shows an incorrect prediction that extends into a wall, and orange shows a repetitive prediction. (b) shows that clustering-based methods can generate poor predictions due to the presence of incomplete trajectory segments in the dataset. In particular, if each of the three trajectories forms its own cluster, predictions based on the corresponding motion patterns can be repetitive (orange) or incorrect (green).

Figure 3-8: Comparison between ASNSC and DPGP. Algorithm 1 is trained on the pedestrian dataset, and used to make predictions for a separate test set of 24 trajectories. For brevity, three of the test trajectories are shown here. Given the black observation history of a pedestrian, the goal is to find the set of possible future paths. (b) shows that DPGP made some repetitive and incorrect predictions. In contrast, (a) shows that ASNSC correctly predicted the pedestrian's future paths in all three cases (consistent with the gray training set).
Table 3.1: Prediction performance comparison between ASNSC and DPGP. DPGP’s prediction results depend on two parameter settings, the likelihood threshold $\xi$, and the number of chosen clusters $n$. In contrast, ANSC predictions do not require parameter tuning. Results show that ASNSC generates more accurate predictions than DPGP.

<table>
<thead>
<tr>
<th></th>
<th>DPGP ($\xi, n$)</th>
<th>ASNSC</th>
</tr>
</thead>
<tbody>
<tr>
<td>missed (%)</td>
<td>27.3 36.4 31.8 18.2</td>
<td>4.5</td>
</tr>
<tr>
<td>correct (%)</td>
<td>72.7 63.6 68.2 81.8</td>
<td>95.5</td>
</tr>
<tr>
<td>repetitive (%)</td>
<td>38.6 31.8 38.6 31.8</td>
<td>0</td>
</tr>
<tr>
<td>incorrect (%)</td>
<td>9.1 4.5 13.6 50.0</td>
<td>4.5</td>
</tr>
</tbody>
</table>

The prediction results on a few representative trajectories from the test set are shown in Fig. 3-8. ASNSC produced correct predictions in nearly all test cases, whereas DPGP made some repetitive and erroneous predictions. To find the set of possible paths using DPGP, we identify the set of clusters that agree with observation (above a likelihood value $\xi$) and pick the best $n$ clusters. The parameters $\xi$ and $n$ are tuned through a grid search procedure. In comparison, the ASNSC does not require parameter tuning for prediction because the transition matrix $T$ specifies the number of possible future behaviors.

Table 3.1 compares the prediction performance of DPGP at 4 different parameter settings with the proposed prediction scheme. Moreover, the per iteration complexity of the proposed method is an order of magnitude lower than using Gibbs sampling for the DPGP model. In particular, learning the DPGP model via Gibbs sampling (200 steps) took 377.3 seconds to complete, while the proposed algorithm took 4.98 seconds to learn the set of dictionary atoms.

3.4.3 Sensitivity Analysis

ASNSC does not assume a fixed size dictionary. Instead, the proposed algorithm incrementally expands the size of the dictionary as dictated by the parameter $\text{thres} \in \mathbb{R}$.
Figure 3-9: Sensitivity analysis of the \( \text{thres} \) parameter (line 4 Algorithm 1) for dataset II. (a) shows the number of dictionary atoms learned from data as a function of the \( \text{thres} \) parameter. (b) shows the objective value per data sample as a function of the \( \text{thres} \) parameter.

(0, 1) in line 4 of Algorithm 1. When the relative representational error of a data sample exceeds a certain threshold, the data sample is added to the dictionary (with some random noise). Thus, the \( \text{thres} \) parameter lends a very natural interpretation—the largest accepted representational error. Consequently, with a larger value of the \( \text{thres} \) parameter, the learned dictionary is expected to have with fewer columns and higher representational error. This is corroborated with empirical evaluation on both trajectory datasets; for brevity, only result from dataset II is shown in Fig. 3-9. Further, the segmentation pattern does not change drastically in the neighborhood of the selected \( \text{thres} \) value. For instance, Fig. 3-9 shows that ASNSC finds 9 dictionary atoms on dataset II for \( \text{thres} \in [0.65, 0.8] \).

### 3.5 Summary

This chapter presented the augmented semi-nonnegative sparse coding framework (ASNSC) that allows for specifying coupling constraints on different parts of the dictionary. An algorithm with convergence guarantees was developed for solving ASNSC. Furthermore, the motion prediction problem was formulated in the ASNSC framework by modeling trajectories as concatenations of local motion patterns. The
learned motion patterns showed good correspondence with human intuition (e.g.,
identifying an intersection), and were be used for generating path predictions. Results
on real-world datasets showed significant improvement in both prediction accuracy
and computational time over DPGP, a state-of-art clustering-based method.
Chapter 4

Motion Planning in Uncertain Environments

Many robotics applications require repeated, on-demand motion planning in mapped environments. For instance, it is important for mobility-on-demand shuttle vehicles [47, 96] and indoor service robots [10] to find collision-free paths quickly at customer’s requests. Although a map is typically available in such applications, the presence of other dynamic agents, such as people, can induce frequent, dynamic changes in the environment. In particular, as shown in Fig. 4-1, static objects can emerge or shift in position over time, and other dynamic agents often operate in close proximity of the vehicle. The challenge is how to use prior knowledge of the map to speed up motion planning while also deal with dynamic changes efficiently. As reviewed in Section 1.2.2, one solution approach is to pre-compute a potential function that encodes pairwise cost-to-go for improving the computational speed of finding feasible paths, and for guiding local searches around dynamic obstacles. However, since storing pairwise potential can be impractical given the $O(|V|^2)$ memory requirement, existing work often needs to compute a potential function for each query to a new goal, which would require substantial online computation.

This chapter addresses the problem by using diffusion map, a machine learning algorithm, to learn the map’s geometry and develop a memory-efficient parametrization (i.e., $O(|V|)$) of pairwise potentials. Specially, each state in the map is transformed to
Figure 4-1: Dynamic elements in a mapped environment. (a) shows new static obstacles (e.g., poster stands) can emerge and position of existing static objects (e.g., chairs) can be shifted over time. (b) shows the presence of dynamic agents, such as pedestrians. A vehicle need to be able to re-plan its path efficiently given perception updates (changes) in such environments.

4.1 The Diffusion Space

4.1.1 Coordinate Transformation

As explained in Section 2.3, diffusion map is a graph-based algorithm commonly used for dimensionality reduction. The following reviews the construction of a diffusion map and its key properties in relation to motion planning. For clarity, the following notations are used in the exposition below. For a given graph $G$, each node $n_i$ is associated with a coordinate $x_i$ in the original space $\mathcal{X}$. Diffusion map finds an alternative parametrization $x_i \mapsto y_i$ in a diffusion space $\mathcal{Y}$. Let $d_\mathcal{X}, d_\mathcal{Y}, d_G$ denote Euclidean distance in the original space, Euclidean distance in the diffusion space, and geodesic distance on the graph, respectively.
### Algorithm 3: Diffusion maps

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. <strong>Input:</strong></td>
<td>distance matrix $W$, time step $t$, truncation $k$</td>
</tr>
<tr>
<td>2. <strong>Output:</strong></td>
<td>diffusion matrix $Y$</td>
</tr>
<tr>
<td>3.</td>
<td>similarity matrix $A \leftarrow \text{ker}(W)$</td>
</tr>
<tr>
<td>4.</td>
<td>diagonal degree matrix $D \leftarrow \text{rowSum}(A)$</td>
</tr>
<tr>
<td>5.</td>
<td>$A_1 \leftarrow \frac{1}{2}A + \frac{1}{2}D$ // lazy Markov chain</td>
</tr>
<tr>
<td>6.</td>
<td>$A_2 \leftarrow D^{-1}A_1D^{-1}$ // anisotropic scaling</td>
</tr>
<tr>
<td>7.</td>
<td>$D_2 \leftarrow \text{rowSum}(A_2), \quad M_2 \leftarrow D_2^{-1}A_2$</td>
</tr>
<tr>
<td>8.</td>
<td>$(V, \Lambda) \leftarrow \text{eigs}(D_2^{-\frac{1}{2}}A_2D_2^{-\frac{1}{2}}, k + 1)$ // eigen-decomposition</td>
</tr>
<tr>
<td>9.</td>
<td>$\Phi \leftarrow D_2^{-\frac{1}{2}}V$</td>
</tr>
<tr>
<td>10.</td>
<td>$Y \leftarrow [\lambda_2\phi_2, \lambda_3\phi_3, \ldots, \lambda_{k+1}\phi_{k+1}]$</td>
</tr>
<tr>
<td>11.</td>
<td><strong>return</strong> $Y$</td>
</tr>
</tbody>
</table>

The key idea of diffusion maps is that local distances $(d_X)$ captured by edge weights in the graph are reliable, but large distances in the original coordinates $(d_X)$ can be misleading as a similarity metric. For instance, little can be inferred about the relationship between two images – either the same object viewed at different angles or two distinct objects – if every pair of pixels differs substantially. This work is motivated by the insight that, for motion planning, local distances (i.e., $d_X$ between adjacent nodes) are reliable and easy to compute, but large distances ($d_X$) between non-neighboring nodes are often uninformative. For example, a pair of non-neighboring nodes with a small $d_X$ could be separated by a wall, thus having a large geodesic distance $d_g$. The diffusion map algorithm addresses this problem by solving for a diffusion process to learn the geometry of the graph in the original space $X$ [108].

The construction of a diffusion map for motion planning is outlined in Algorithm 3. Given an occupancy grid map represented as a graph $G$, the first step is to construct a sparse distance matrix $W$ (line 1), where $W_{ij}$ is the Euclidean distance between node $i$ and $j$ if they are adjacent, and infinity otherwise. Then, a similarity matrix $A$ is computed by applying a kernel (line 3) that maps each entry in the distance matrix to a similarity score. This work uses the Gaussian kernel $g(x) = \exp(-\frac{x^2}{2w^2})$, where $w$ is the width of a grid. The similarity matrix can be interpreted as the transition probabilities of a Markov chain. In particular, $A_{ij}/D_{ii}$ is the transition probability from node $i$ to $j$, where $D_{ii} = \sum_j A_{ij}$ is called the degree of node $i$ (line 4). To
avoid potential issues with aperiodicity, the Markov chain is made lazy in line 5. The last step computes a transition matrix $M_2$ (line 7), its $k+1$ largest eigenvalues $\{\lambda_1, \ldots, \lambda_{k+1}\}$, and the associated right eigenvectors $\{\phi_1, \ldots, \phi_{k+1}\}$ (lines 8-9). Recall from Section 2.3, diffusion map associates the $i$th node in the original graph with the $i$th entry of each eigenvector (line 10),

$$x_i \mapsto y_i = [\lambda_2^t \phi_2(i), \lambda_3^t \phi_3(i), \ldots, \lambda_{k+1}^t \phi_{k+1}(i)]. \quad (4.1)$$

The new space $\mathcal{Y}$ is called the $k$-truncated diffusion space, and the distance $||y_i - y_j||_2$ is known as the $k$-truncated diffusion distance [84]. The choice of parameter $t$ in (4.1) will be explored in greater details in Section 4.1.3.

We make a few remarks regarding the process of computing this mapping. First, the coordinate $x_i$ in the original space is not needed in computing the diffusion map, because Algorithm 3 finds the eigenvectors only using the distance matrix $W$, which captures local distances and connectivities. Second, for motion planning problems, $W$ is typically sparse since each node in a grid map has only a few neighbors. The sparse structure is maintained throughout Algorithm 3, thus allows the procedure to be memory efficient. Third, a disconnected graph with $r$ components would have precisely $r$ eigenvalues equal to one, and the corresponding eigenvectors would reveal the $r$ components. This can be seen intuitively by noting that a transition matrix always has an eigenvalue of one since each row sums to one ($M1 = 1$), and so a disconnected graph would have one such eigenvector for each of its $r$ components. For simplicity, the graph (domain) is assumed to be connected in this work.

Recall from Section 2.3, diffusion map can be analyzed by simulating a Markov random walk on the graph. Theorem 1 states that the diffusion distance with a time scale parameter $t$ equals to a weighted distance between two probability clouds after $t$ steps. Intuitively, it is expected that nodes with smaller geodesic distance on the graph to have more overlap in densities after a $t$-step diffusion process. This suggests that diffusion distance is a meaningful pairwise similarity measure.

In light of the algebraic form of $\lambda_i^t$ in (2.15) and the fact that $\lambda_i \in [0, 1]$, the
diffusion distance can be approximated by keeping the first $k$ eigenvectors, where $k \ll n$. Further, there often exists a spectral gap for real-world problems [86], so a small value of $k$ is typically sufficient to achieve a good approximation of (2.15). It is empirically demonstrated that $k \approx 10$ is sufficient for the complex planning domains considered in Section 4.3. An interesting perspective is that finding the full diffusion coordinate (i.e., $k = n - 1$) is as expensive as solving the all-pairs-shortest-path problem, because finding all $n$ eigenvectors requires $O(|V|^3)$ computation and $O(|V|^2)$ storage. Yet, a diffusion map’s explicit eigenvalue formulation permits an efficient approximation that leads to substantial savings on both computational time and memory usage, because finding the first $k$ eigenvectors requires $O(k|V|^2)$ computation and $O(k|V|)$ storage.

Finally, it can be shown that the eigenvectors are related to geometric structures in the original domain. Without the anisotropic scaling step (line 6 Algorithm 3), $\{v_i\}$’s would also be the eigenvectors of the graph Laplacian matrix [83], which can be seen as a discrete approximation of Laplace-Beltrami operator plus a density term [83,109]. The density term is undesirable because the diffusion process is biased toward regions of high connectivity (i.e., nodes with a large degree). The anisotropic scaling step removes the effect of the density term [83]. Hence, finding a diffusion map is related to solving the heat equation. For instance, the $\{v_i\}$’s of a line graph resemble sinusoidal functions, which are the eigenfunctions of the 1D heat equation. Thus, the first few eigenvectors capture the low frequency components, which can be interpreted as long range structures in the original domain [108]. Researchers have investigated using the eigenvectors $\{v_i\}$’s as basis functions for solving reinforcement learning problems [110], but have not used pairwise relationship as revealed in diffusion maps for motion planning.

In short, diffusion coordinates reveal geometric structures of the original domain, and diffusion distance $d_Y$ is a meaningful similarity metric that relates to geodesic distance $d_G$. Furthermore, diffusion coordinates can be computed and stored efficiently.
A running example  The occupancy grid map of an office space is created using the ROS *gmapping* package, as shown in Fig. 4-2(a). The width of each grid is set to be 0.1m, and grids within 0.25m are considered neighbors if a line connecting their centers does not intersect any obstacle. When converted to a graph representation, this domain contains 20,666 nodes, and an average of 18.7 connections per node. This domain is used as a running example to illustrate various aspects of the proposed algorithms.

The first three dimensions of the diffusion coordinates are visualized in Fig. 4-2(b). For numerical reasons, this work scales each eigenvector (diffusion coordinate) in (4.1) such that $||\phi_k||_2 = N$, where $N$ the number of vertices in the graph. Color shows the correspondence between the original 2D space and the diffusion space. For example, the lime color shows that the top right region in the 2D space is mapped to the top middle region in the diffusion space. Evidently, this diffusion map captures long range structures in the map – the corner rooms are being mapped to tips of a roughly tetrahedron-shaped object in the diffusion space. Hence, pairs of points with large geodesic distances, such as those in different rooms, would also have large diffusion distances.
4.1.2 Finding the Truncation Parameter

Recall a diffusion map is memory efficient by only keeping the first $k$ eigenvectors (4.1). Figure 4-3(a) shows the first 15 eigenvalues for the domain illustrated in Fig. 4-2. Notice that there is a sharp drop between the 7th and 8th eigenvalues. When computing the diffusion distance, since each diffusion coordinate is defined as $\lambda_k^t \phi_k$, the influence of the eigenvectors corresponding to smaller eigenvalues decays very quickly. Figure 4-3(b) shows the value of $\lambda_k^t$ when the time scale parameter $t$ is set to 10,000. Recall this work uses $k = 10$ (Section 4.1.1), which can be explained by the observation that $\lambda_k^t \approx 0$ for $k > 8$.

4.1.3 Finding the Time Scale Parameter

Recall from (2.15), the time scale parameter $t$ is the number of steps in a random walk process. Due the algebraic form of scaling term $\lambda_k^t$ in (4.1), when $t$ is set too small, the approximation of diffusion distance would be poor because the truncated coordinates have non-negligible weights; when $t$ is set too large, a lot of information would be lost because the diffusion space would collapse to the first dimension. To empirically determine a good value of $t$, we find the optimal paths for several randomly
Figure 4-4: Effect of a time scale parameter \( t \) on diffusion distance. The subfigures plot the geodesic distance \( d_g \) against the diffusion distance to goal \( d_y \) on a set of optimal paths at the specified settings of \( t \). (a), (b) and (c) show when the time scale parameter \( t \) is set too small, too big, or well tuned, respectively. In particular, a well tuned \( t \) would induce a monotonic relationship between \( d_g \) and \( d_y \).

Figure 4-5: Visualization of the diffusion distance with \( t = 10,000 \), which is well tuned according to Fig. 4-4. Each subfigure shows the diffusion distance \( d_y \) from every node in the graph to the goal marked with a red dot. These plots show that diffusion distance forms an intuitive pairwise similarity measure, such that points further apart tend to have larger diffusion distances.

generated test cases (e.g., use A*), and plot the geodesic distance versus the diffusion distance to goal. Specifically, each path is plotted using a different color as shown in Fig. 4-4. Ideally, there would be a monotonic linear relationship, which would allow diffusion distance to be a perfect surrogate for (completely encodes) geodesic distances. Figures 4-4(a) and 4-4(b) show that when \( t \) is set too small or too big, respectively, there can be local minima in diffusion distance along certain directions. It is found empirically that setting \( t \approx 50w_G \) as shown in Fig. 4-4(c), where \( w_G \) is the width of a graph, leads to good performance. Figure 4-5 visualizes the diffusion distances from everywhere in the domain to the goal nodes marked with the red dots.
4.2 Motion Planning with Diffusion Maps

4.2.1 Static Environments

Diffusion search is developed in Algorithm 4, which finds paths by steepest descent on diffusion distance surface (e.g., Fig. 4-5). Using terminology common to graph-based search algorithms, a priority queue OPEN is used for choosing the next state to be expanded, and a list CLOSED is used for keeping track of the expanded states (line 3). The OPEN queue is sorted by the diffusion distance to goal (line 4). The main loop is shown in lines 5-10, which proceeds by repeatedly expanding the node with minimum diffusion distance to goal (line 6) and adding its unexplored neighbors to the queue (lines 7-9). As explained in Section 4.1.3, a problem with keeping only the first $k$ coordinates in computing diffusion distance is that short scale structures would be lost. It is found that the diffusion distance surface (e.g., Fig. 4-5) is typically very flat in the vicinity of the goal, leading the diffusion search algorithm to visit many more states than necessary near the goal state. This behavior can be observed in Fig. 4-4(c), in which the lines becomes more flat when diffusion distance is below 20. In practice, this problem can be handled by noting that the vehicle is typically very close to the goal (e.g., less than 2m) when diffusion distance falls below a certain threshold. At this point, a simple analytical solution, such as computing a Dubins curve, is usually sufficient to reach the goal. For a fair comparison with other graph-based search algorithms, the diffusion search algorithm switches to A* after the diffusion distance falls below a threshold $\eta$, as shown in lines 11-13.

Recall this work assumes that the domain can be represented as a graph with a finite number of nodes. Given such a representation, diffusion search is complete. In particular, if there exists a path between the start node and the goal node, diffusion search will find a feasible path in finite time. This can be seen by noticing that (i) the algorithm would terminate in finite time because each state can be expanded no more than once, and (ii) either a feasible path would be found or all nodes reachable from the start node would be expanded (no solution exists).

An example of diffusion search is shown in Fig. 4-6(b). Compared with the A*
Algorithm 4: Diffusion search (DS)

1. **Input:** diffusion matrix $Y$, diffusion threshold $\eta$
2. **Output:** path $p$ from $s_{\text{start}}$ to $s_{\text{goal}}$
3. $\text{OPEN} \leftarrow \emptyset$, $\text{CLOSED} \leftarrow \emptyset$
4. insert $s_{\text{start}}$ into $\text{OPEN}$ with value $||y_{\text{start}} - y_{\text{goal}}||_2$
5. while $s_{\text{goal}}$ not expanded do
   6. $s \leftarrow \text{pop}(\text{OPEN})$
   7. for each neighbor $s'$ of $s$ do
      8. if $s'$ not in $\text{CLOSED}$ then
         9. insert $s'$ into $\text{OPEN}$ with value $||y_{s'} - y_{\text{goal}}||_2$
      10. insert $s$ into $\text{CLOSED}$
     11. if $||y_s - y_{\text{goal}}||_2 < \eta$ then
         12. $p' \leftarrow \text{retrievePath}(s_{\text{start}}, s)$
         13. return $p \leftarrow p' \cup \text{findPath}(s, s_{\text{goal}})$

In Fig. 4-6(a), diffusion search finds a path that is 10.2% longer than optimal but expands many fewer states ($\leq 2\%$ the number in this case). Since computational time is linear in the number of expanded states, diffusion search is more than an order of magnitude faster than A*. A more detailed performance comparison will be presented in Section 4.3.3. A major source of sub-optimality is choosing the next state greedily (line 9 Algorithm 4), which can lead to some oscillations along the path (e.g., near $(17, 10)$ in Fig. 4-6(b)). This is because as explained in Section 4.1.1, the $k$ truncated diffusion map is an approximation to the full diffusion map, which throws away high frequency components (short range structures) for computation and storage reasons. It is possible to improve the solution quality by using post-processing methods [111,112]. Alternatively, the local search area for the next state (line 7) can be expanded to beyond just the set of neighbors. A more detailed description of this idea is presented in Section 4.2.2.

Furthermore, diffusion distance can be used as a heuristic to prune states that are unlikely to be on the optimal path. Thereby, diffusion search can be easily integrated into an existing graph-based motion planner to improve computational speed. In particular, Weighted A* [39] is considered. Using an inadmissible heuristic $h(x_1, x_2) = c \cdot ||x_1 - x_2||_2$ for $c > 1$, Weighted A* achieves better computational speed.
Figure 4-6: Path planning in a static environment. Four algorithms are used to find a path from (20, 22) to (20, 12). Red curve shows the planned path, the red dot marks the goal state, and blue shows the set of expanded states. Diffusion distance can be either used by itself for finding a feasible path as shown in (b), or used as a similarity metric and integrated with an existing algorithm as shown in (d). Note that diffusion search finds a path of comparable quality (within 10% of optimal) at an order of magnitude smaller computational cost.

at the expense of losing optimality. An example of using Weighted A* for path planning is shown in Fig. 4-6(c). The problem with Weighted A* is that similar to A*, a substantial amount of computation can be spent on exploring dead-ends. This problem is addressed by pruning states whose diffusion distance to goal are higher than that of their parents. Specifically, nodes being pushed onto the OPEN queue are penalized (i.e., adding a large constant to their cost) if their diffusion distance to goal is larger than that of their parent. An example of Weighted A* with diffusion metric is shown in Fig. 4-6(d). Note that Weighted A* with diffusion metric attains

1Weighted A* reduces to A* with c = 1. This work uses c = 3.
Algorithm 5: Diffusion search with motion primitives

1. motion primitives MP ← computeMP()
2. while $x_{cur}$ not sufficiently close to goal do
3.   $(x_{cur}, Obs) ←$ sensorUpdate()
4.   MP$' ←$ pruneWithObs($x_{cur} + MP, Obs$)
5.   // find motion primitive segments that reduce diffusion distance to goal
6.   MP$'' ←$ pruneWithDiffusionDist(MP$'$, $Y$)
7.   $p'' ←$ chooseBestLocalPath(MP$''$)
8.   move along $p''$

a slightly better path while expands only half as many nodes.

4.2.2 Dynamic Environments with Limited Sensing Range

In many real-world applications, there can be frequent small changes in the environment. For instance, for indoor navigation, obstacles' position (e.g., chairs) might be shifted in relation to the map and there might be other dynamic agents in the environment. Assuming no major changes to the underlying geometry (e.g., blocking an entire corridor), the pre-computed diffusion map can be used to guide a local motion planner. This enables the planning system to focus the computational resources within the vehicle's perception range, without needing to compute a complete feasible path to goal.

Algorithm 5 proposes a motion planner that combines diffusion search with motion primitives. A set of motion primitives, shown in Fig. 4-7(a), is pre-computed based on the Dubins car model [46] (line 1), which accounts for kinematics constraints such as the minimum turning radius limitation. The main loop is repeated until the vehicle reaches the goal (line 2). At every time step, the vehicle receives an update of its pose and the surrounding obstacles within its perception range (line 3). Then, the set of motion primitives is translated and rotated with respect to the vehicle's pose to form a set of local paths. Each local path is pruned up to the first point that collides with any obstacle (line 4). Each path is further pruned up to the point with the minimum diffusion distance to goal (line 5-6). At this stage, what remains is a set of traversable paths that lead to positions closer to the goal than the vehicle's
Figure 4-7: Path planning in a dynamic environment. (a) shows a set of motion primitives based on the Dubins car model. (b-d) show path planning using Algorithm 5. Grey shows the map, purple shows the newly added obstacles that are only visible within vehicle's perception range. Red shows the path traveled, and green shows the chosen local path as described in line 7 Algorithm 5.

The last step is to choose the best local path within this set (line 7), which is the one that leads to the lowest diffusion distance to goal. The vehicle is presumed to move along this chosen path until the next sensor update (line 8).

An example of applying Algorithm 5 for motion planning is shown in Figs. 4-7(b) to 4-7(d). New obstacles shown in purple are generated at random to simulate dynamic changes in the environment. The vehicle is assumed to have a perception radius of three meters, within which the newly added obstacles can be seen. Following diffusion distance pre-computed from the static map, and planning locally using the pre-computed motion primitives, the vehicle finds a path that is within 2% of the
Algorithm 6: Diffusion search for multiple routes

1. **Input:** diffusion matrix $Y$, number of raw paths $n_p$, sampling threshold $\eta_1$, diffusion threshold $\eta_2$, filter threshold $\eta_3$, number of partitions $K$

2. **Output:** a set of paths $\mathcal{P}$ from $s_{start}$ to $s_{goal}$

3. $C \leftarrow k\text{-}means(Y, K)$ // partition the graph

4. $\mathcal{P} \leftarrow \emptyset$, $\mathcal{L} \leftarrow \emptyset$

5. $S_{mid} \leftarrow \text{sampleMidPts}(Y, s_{start}, s_{end}, n_p, \eta_1)$

6. for each point $s_{mid}'$ in $S_{mid}$ do

7. // plan a path from $s_{start}$ to $s_{end}$ through each sampled point $s_{mid}'$

8. $p_{start}, s' \leftarrow \text{diffusion_search_near}(Y, s_{start}, s_{mid}', \eta_2)$

9. $p_{end}, s'' \leftarrow \text{diffusion_search_near}(Y, s_{end}, s_{mid}', \eta_2)$

10. $p_{mid} \leftarrow \text{diffusion_search}(Y, s', s'', \eta_2)$

11. $p^i \leftarrow [p_{start}, p_{mid}, p_{end}]$

12. $\mathcal{P} \leftarrow \mathcal{P} \cup p^i$

13. for each path $p_i$ in $\mathcal{P}$ do

14. $l^i = [l^i_1, \ldots, l^i_K] \leftarrow \text{findSegLen}(p^i, C)$ // segment each planned path

15. $\mathcal{L} \leftarrow \mathcal{L} \cup l^i$

16. return $\mathcal{P} \leftarrow \text{sort_and_filter}(\mathcal{P}, \mathcal{L}, \eta_3)$ // return distinct paths

optimal. Moreover, comparing this approach with Algorithm 4 sheds light on the aforementioned oscillation problem. In particular, choosing the best motion primitive can be seen as having a larger look-ahead distance than choosing the best neighbor. As a result, the oscillation problem due to throwing away high frequency components is mitigated by using the motion primitives. Despite of having more obstacles, the path here (Fig. 4-7(d)) is about 8% shorter than the one found by diffusion search (Fig. 4-6(b)).

### 4.2.3 Multiple Routes

The preceding paragraphs described methods for finding one feasible (near optimal) path and handling local changes in an environment. The following presents a method for finding multiple distinct paths, which would be useful for handling topological (global) changes. In particular, there are often multiple ways to go between two locations, and the shortest route might not always be the most desirable. For example, consider an office building with multiple elevators and staircases. When the closest elevator is out of order, or during peak hours when the perceived traffic (congestion)
Figure 4-8: Finding paths from different homotopic classes. (a) shows partitioning the original space by using k-means in the diffusion coordinates. (b) shows using Algorithm 6 to find three paths, each shown in a distinct color, that connect (20, 22.5) and (20, 12.5). This enables the vehicle to re-plan at the route level in response to local congestion.

would drastically increase the amount of time to traverse the shortest path, a person may choose to take a different elevator on a longer but faster route. In such cases, it is important to be able to find multiple routes, and then pick the best one under the current traffic conditions.

This work distinguishes between finding multiple routes and finding the k-shortest paths. The former considers topological diversity (e.g., non-homotopic) while the latter does not. When using grid maps, the k-shortest paths might be very similar to each other and thus not useful for re-routing. This work finds multiple routes by leveraging on the diffusion map’s ability to (i) plan path quickly between any pair of nodes as shown in Section 4.2.1, and (ii) partition the domain into intuitive regions, such as along walls and other boundaries. Specifically, recall the diffusion coordinates (eigenvectors of the transition matrix) are related to geometric structures in the graph and have deep connections to spectral clustering [86]. An example of partitioning a graph by running k-means in the diffusion space is shown in Fig. 4-8(a).

Given a partitioning of the graph, it is conceivable that distinct paths can be found via a two-stage process. First, find the k-shortest sequences of partitions (e.g., region $8 \rightarrow 5 \rightarrow 7 \rightarrow 2 \rightarrow 4$), and second, plan a set of paths each constrained to a corresponding
sequence of partitions. Yet, this work develops a different approach that is empirically faster. In particular, the proposed algorithm first finds a set of feasible paths constrained to go through different middle points, and then filters out the redundant paths using the partition information.

Diffusion search for multiple routes is outlined in Algorithm 6. The graph is first partitioned into $K$ regions using the k-means algorithm in line 3. Then, a set of $n_p$ nodes are sampled randomly from the graph in line 5. To obtain points more likely on distinct routes, the samples are constrained to be at least $\eta_1$ apart from each other in the diffusion space, and yet less than $\|y_{\text{start}} - y_{\text{end}}\|_2$ away from either the start or the end node. For each of the sampled points $s_{\text{mid}}$, the algorithm (lines 6-12) finds a corresponding feasible path that goes through it. Specifically, diffusion search (Algorithm 4) is used to find three path segments (lines 7-10) that, when joined, connects the start and the goal while comes to within a diffusion distance of $\eta_2$ from the point $s_{\text{mid}}$. At this stage, the algorithm has found a set of feasible paths $\mathcal{P}$, and what remains is to select a subset of paths that are distinct from each other. Given the partition information, each path can be associated with a feature vector $\mathbf{V}$ (lines 13-15), which computes the length of the path through each partition. Intuitively, similar paths (e.g., slightly shifted) are expected to have similar feature vectors. Last, the algorithm sorts the set of paths by total length $\sum_k l_k$, and picks the ones that are sufficiently different, such that $\|\mathbf{l}^i - \mathbf{l}^j\|_2 > \eta_3$ for each $i, j$ pair in the selected subset. An example of three routes found by Algorithm 6 is shown in Fig. 4-8(b). The algorithm spent approximately 75% the time to find the three routes (automatically selected three out of $n_p = 10$ planned paths) as using A* to find one path (Fig. 4-6(a)). This computational efficiency is due to diffusion map's on the ability to find individual path quickly (lines 6-12) and to distinguish between them using the partition information (lines 13-16).

For simplicity, this work manually tunes the number of partitions, $K$. Future work will consider learning $K$ from data, such as applying DP-means [113].
Figure 4-9: Path planning for a nonholonomic vehicle. The shaded colors (yellow to blue) show the number of expanded states at each 2D position. Recall this is a 3D domain, so the heading angle dimension is collapsed for plotting the number of expanded states.

4.3 Results

The following presents simulation and hardware results on motion planning with diffusion maps in two additional domains. The performance of the diffusion-based algorithms is assessed on randomly generated test cases.

4.3.1 Path Planning for a Nonholonomic Vehicle

Consider motion planning for a nonholonomic vehicle in the environment shown in Fig. 4-2(a). Following [111], a heading angle dimension discretized at $5^\circ$ increments is included to account for vehicle dynamics. When converted to a graph representation, this domain contains $\approx 1.4$ million nodes. The connectivity (number of neighbors per node) of this domain is lower than that of the 2D map because the vehicle is constrained by a $0.5$m minimum turning radius. An example is shown in Fig. 4-9, in which diffusion search found a feasible path within 15% of the optimal path but explored three orders of magnitude fewer nodes than A* that uses the Euclidean distance heuristic.
Table 4.1: Computational time for finding diffusion maps in domains of varying complexity. This work keeps ten eigenvectors in the diffusion space, that is \( k = 10 \) (line 10 Algorithm 3).

<table>
<thead>
<tr>
<th></th>
<th>2D</th>
<th>3D-nh</th>
<th>mlt-flr</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of states</td>
<td>20,666</td>
<td>1,485,912</td>
<td>311,890</td>
</tr>
<tr>
<td>avg. number neighbors per node</td>
<td>18.7</td>
<td>11.9</td>
<td>19.1</td>
</tr>
<tr>
<td>computational time (sec)</td>
<td>1.7</td>
<td>181</td>
<td>381</td>
</tr>
</tbody>
</table>

4.3.2 Path Planning on Multiple Floor Levels

Next consider motion planning on three floors connected by multiple elevators, as shown in Fig. 4-10. Each floor is mapped using the ROS gmapping package and discretized to build a grid map with the same resolution as Fig. 4-2(a). When converted to a graph representation, the three floors combined have more than 300,000 nodes. In addition to having ten times more nodes than Fig. 4-2(a), this domain is challenging because there are often multiple routes connecting nodes on different floors. An example is shown in Fig. 4-11, in which A* with the Euclidean distance heuristic explores half of the nodes before finding the optimal path. In comparison, shown in Fig. 4-10(d), diffusion search finds a feasible path within 12% of the optimal path but explores two orders of magnitude fewer nodes than A*. Connections between floor levels and the diffusion distances with respect to the goal are shown in Fig. 4-10.

4.3.3 Performance Comparison

The domains illustrated in Fig. 4-2(a), Fig. 4-9, and Fig. 4-11 are denoted as 2D grid map (2D), 3D grid map with nonholonomic constraints (3D-nh), and multiple connected floor levels (mlt-flr), respectively. The complexity of each domain in graph representation and the corresponding time required for computing diffusion maps are summarized in Table 4.1. The algorithms are run on a computer with an Intel i7-2600K CPU and 16GB of memory.

Diffusion-based motion planning algorithms are being evaluated on 100 randomly generated test cases in each domain. In particular, Fig. 4-12 shows a comparison of solution quality (path length) and the number of expanded states. Run time is also computed and shown in Table 4.2. To account for variability among different
test cases, we normalized the performance statistics by that of A*. The performance statistics are summarized in Table 4.2.

Diffusion search finds paths that are about 20% longer than the optimal paths. A major cause is that greedy search leads to some oscillations along the path. In practice, the use of a larger look-ahead distance, as in Algorithm 5, mitigates this issue to a large extent. Diffusion search often finds a feasible path two orders of magnitude faster than A* and one order of magnitude faster than Weighted A*. Moreover, diffusion distance can be integrated into A*-based algorithms for pruning states that are unlikely to be on the optimal path. In particular, a factor of two speedup and
Figure 4-11: Path planning on multiple floors using the A* algorithm. Grey, pink and lime green show three floor levels, and gray vertical lines are the connecting elevators. Blue denotes expanded states, and red shows planned path. A comparison with Fig. 4-10(d) shows that diffusion search can find a feasible path for this problem while exploring much fewer states.

Also a small improvement in solution quality were achieved by augmenting Weighted A* with diffusion distance, as shown in the bottom two rows of Table 4.2. Moreover, in addition to significantly reducing the run time, the availability of a pre-computed diffusion map also allows the vehicle to move towards its goal without needing to first find a feasible path, such as described in Algorithm 5.

4.3.4 Hardware Experiment

An autonomous vehicle is developed using the Pioneer 3-AT rover platform, as shown in Fig. 2-1(a). The vehicle is equipped with a SICK Lidar for mapping and localization, and a Velodyne sensor for obstacle detection. The vehicle navigated fully autonomously in a dynamic environment (Fig. 4-10) using Algorithm 5. A hardware demonstration video is available at https://youtu.be/3s9gswbhWSU.

Figure 4-13(a) illustrates the vehicle navigating at a nominal speed of 0.7m/s through a busy indoor environment with many pedestrians – an average of 11.3 people.

---

3In this work, grids within 0.25m are considered neighbors if the line connecting their centers does not intersect any obstacle. Thus, a feasible path can have fewer states on its path than that of the optimal path.
Figure 4-12: Performance statistics on random test cases. $A^*$ always finds the optimal solution but needs to explore a large number of states. Diffusion search finds feasible paths that are about 20% longer than the optimal paths, but explores two orders of magnitude fewer states than $A^*$. 

(a) Performance on the 2D domain

(b) Performance on the 3D-nh domain

(c) Performance on the mlt-flr domain
Figure 4-13: Autonomous navigation in a dynamic environment using Algorithm 5. (a) shows the vehicle comes in frequent interaction with pedestrians, averaging to 11.3 people within 2.5m of the vehicle per minute. (b) illustrates the vehicle detecting obstacles (black regions) within its perception range (white patch). A set of motion primitives (blue) is pruned by the detected obstacles. The feasible motion primitive (green) that leads to the smallest diffusion distance to goal is chosen, which in this case, navigates around a person.

came within 2m of the vehicle per minute. The ROS costmap_2d package is used to process Velodyne point-cloud data for detecting obstacles within a 10m x 10m region around the vehicle. The size of the region is chosen to achieve a 10Hz update rate for enabling fast local replanning in Algorithm 5. Figure 4-13(b) shows the vehicle quickly finds a safe local path (green) from its motion primitives library (line 4-7 Algorithm 5) to navigate around a pedestrian. In short, our experiment shows that pre-computing a diffusion map is an effective method for reducing computational time. Recall in Algorithm 5, a vehicle does not need to wait for a feasible path to be found when assigned a new goal.

4.4 Summary

This chapter presented a multi-query motion planning algorithm for dynamic environments. In particular, given a previously generated map, the proposed algorithm would learn the map’s geometry and find a parametrization that encodes pairwise similarity. This similarity metric, called diffusion distance, was used as a potential function to guide local searches. The proposed algorithms were applied to three real-world do-
Table 4.2: Average performance on the random test cases shown in Fig. 4-12. Path quality is measured by the ratio of (path length)/(optimal path length). Number of visited states is normalized by the number of states on the optimal path. Run time is normalized by that of A*. Second row shows that diffusion search finds feasible paths of comparable quality to other approaches while expanding many fewer nodes. Third and fourth rows show augmenting diffusion distance to a graph-based method leads to more than two fold reduction in the number of expanded states and run time. DS, A*_w, and A*_wd stand for diffusion search, weighted A* and weighted A* with diffusion metric, respectively.

<table>
<thead>
<tr>
<th></th>
<th>Path Length Ratio</th>
<th>Expanded States Ratio</th>
<th>Run Time Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2D</td>
<td>3D-nh</td>
<td>mlt-flr</td>
</tr>
<tr>
<td>A*</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>DS</td>
<td>1.14</td>
<td>1.26</td>
<td>1.25</td>
</tr>
<tr>
<td>A*_w</td>
<td>1.11</td>
<td>1.33</td>
<td>1.14</td>
</tr>
<tr>
<td>A*_wd</td>
<td>1.07</td>
<td>1.27</td>
<td>1.12</td>
</tr>
</tbody>
</table>

mains, and achieved orders of magnitude improvement in computational speed compared with single-query methods. Moreover, a hardware experiment demonstrated autonomous navigation in a dynamic indoor environment.
Chapter 5
Decentralized Collision Avoidance and Socially Aware Navigation in Dynamic Environments

The previous chapter presented a strategy for finding/re-planning feasible paths in environments with frequent static changes, such as moved furnitures. The strategy was also used to enable autonomous navigation around dynamic agents by treating them as static obstacles while re-planning at a high rate. As shown from the experiments in Section 4.3.4, this approach works well when a vehicle is traveling slowly (e.g., half the speed) relative to the other dynamic agents. However, at speeds comparable to that of the surrounding agents, this approach can sometimes lead to unsafe behaviors and even collisions as shown in Fig. 5-1. This chapter develops a computationally efficient algorithm for collision avoidance, which considers the other agents’ motion to overcome such limitations.

Collision avoidance is central to many robotics applications, such as multiagent coordination [51], autonomous navigation through human crowds [59], and crowd simulation [24]. To resolve path conflicts, many of existing works [50–52,114] assume a communication network for agents to broadcast their intents, such as their goals or planned paths. Yet, building a reliable communication network can be expensive or even impossible due to hardware limitations, such as when considering human-
Figure 5-1: An unsafe scenario for interacting with a dynamic agent. (a) illustrates a red vehicle plans its action by treating the blue dynamic agent a static obstacle. In this scenario, if the blue agent continues with its current velocity, the red agent would collide with blue agent regardless of how fast the re-planning loop is running onboard. (b) shows of a hardware experiment of this scenario, where a robot using a such motion planning strategy gets dangerously close to an inattentive pedestrian who is occupied with his phone.

As reviewed in Section 1.2.3, existing work on decentralized collision avoidance can be classified into cooperative and non-cooperative methods. Non-cooperative methods first predict the other agents' motion, and then plan a collision-free path for the vehicle with respect to the other agents' predicted motion. However, this can leads to the freezing robot problem [55], where the vehicle fails to find any feasible path because the other agents' predicted paths would mark a large portion of the space untraversable. The key to resolve this issue is to model interaction, such that the vehicle's action can influence the other agent's motion, thereby having all agents share the responsibility for avoiding collision.

Cooperative methods consider the interaction between dynamic agents, and can be further classified into reaction-based and trajectory-based methods. Reaction-based methods [54, 57, 58] specify one time-step interaction rules for the current geometric configuration, such as modeling dynamic agents as repulsive electric charges. In comparison, trajectory-based methods [15, 16, 60] first infer the other agents' intents, then plan a set of jointly feasible paths for all neighboring agents in the environment. The important distinction here is the look-ahead time, which is one step for
reaction-based methods and multiple steps (until reaching the goal) for trajectory-based methods. As explained in Section 1.2.3, the look-ahead time determines the tradeoff between computational speed and solution quality (e.g., smoothness). In particular, reaction-based methods are fast computationally but can generate oscillatory, unnatural trajectories as in Fig. 5-4, whereas trajectory-based methods tend to generate smooth paths but are computationally prohibitive in crowded environments.

This work seeks to combine the best of both worlds – the speed of the reaction-based methods and the smooth, cooperative motion of the trajectory-based methods. To this end, this chapter presents the collision avoidance with deep reinforcement learning (CADRL) algorithm, which tackles the aforementioned look-ahead time tradeoff by using RL to offload the expensive online computation to an offline learning procedure. Specifically, a computationally efficient (i.e., real-time implementable) interaction rule is developed by learning a value function that implicitly encodes cooperative behaviors.

Furthermore, for robotic vehicles to navigate safely and efficiently in pedestrian-rich environments, it is important to model subtle human behaviors and navigation rules, such as passing on the right and overtaking on the left. However, while instinctive to humans, socially compliant navigation is still difficult to quantify due to the stochasticity in people’s behaviors. As explained in Section 1.2.4, existing works are mostly derived from collision avoidance methods, whose parameters are often learned/tuned with feature-matching techniques to imitate human behaviors. However, these approaches do not generalize well since the feature values can vary from person to person, and even run to run. This work notes that while it is challenging to directly specify the details of what to do (precise mechanisms of human navigation), it is straightforward to specify what not to do (violations of social norms). Specifically, by characterizing undesirable behaviors, CADRL is extended to develop a navigation policy that respects common social norms. The proposed method is shown to enable fully autonomous navigation of a robotic vehicle moving at the average human walking pace (1.2 m/s) in an environment with many pedestrians.
5.1 Problem Formulation

5.1.1 Sequential Decision Making

A non-communicating multiagent collision avoidance problem can be formulated as a partially-observable sequential decision making problem. Let \( s_t, u_t \) denote an agent's state and action at time \( t \). The agent's state vector can be divided into two parts, that is \( s_t = [s^o_t, s^h_t] \), where \( s^o_t \) denotes the observable part that can be measured by all other agents, and \( s^h_t \) denotes the hidden part that is only known to the agent itself. In this work, let \( p \) and \( v \) denote the position and velocity vectors in 2D, respectively; let action be the agent's velocity, \( u = v \); let the observable states be the agent's position, velocity, and radius (size), \( s^o = [p_x, p_y, v_x, v_y, r] \in \mathbb{R}^5 \); and let the hidden states be the agent's intended goal position, preferred speed, and heading angle, \( s^h = [p_{gx}, p_{gy}, v_{pref}, \psi] \in \mathbb{R}^4 \).

The following presents a two-agent collision avoidance problem formulation, where an agent's own state and the other agent's state are denoted by \( s \) and \( \bar{s} \), respectively. The objective is to minimize the expected time, \( \mathbb{E}[t_g] \), of an agent to reach its goal by developing a policy, \( \pi : (s_{0:t}, \bar{s}^o_{0:t}) \mapsto u_t \), which selects an action given the observed state trajectories,

\[
\arg\min_{\pi(s_{0:t}, \bar{s}^o_{0:t})} \mathbb{E}[t_g | s_{0:t}, \bar{s}^o_{0:t}, \pi, \bar{\pi}] 
\]

\[
\text{s.t. } \|P_t - \bar{P}_t\|_2 \geq r + \bar{r} \quad \forall t \tag{5.2}
\]

\[
P_{t+1} = P_t + \Delta t \cdot \pi(s_{0:t-1}, \bar{s}^o_{0:t-1}) \tag{5.3}
\]

\[
\dot{P}_t = \dot{P}_{t+1} + \Delta t \cdot \bar{\pi}(\bar{s}^o_{0:t-1}, s^o_{0:t-1}) \tag{5.4}
\]

where (5.2) is the collision avoidance constraint, (5.3) is the goal constraint, (5.4) is the agents' kinematics, and the expectation in (5.1) is with respect to the other agent's policy and hidden states (intents). Note that static obstacles can be modeled as stationary agents, which will be discussed in more details in Section 5.5.2.
Although it is difficult to solve for the optimal solution of (5.1)-(5.4), this problem formulation can be useful for understanding the limitations of the existing methods. In particular, it provides insights into the approximations/assumptions made by existing works. A common assumption is reciprocity, $\pi = \tilde{\pi}$, such that each agent would follow the same policy [16, 57]. Thereby, the main difficulty is in handling the uncertainty in the other agent’s hidden intents (e.g., goals).

Reaction-based methods [54, 57] often specify a Markovian policy, $\pi(s_{0:t}, \tilde{s}_{0:t}^o) = \pi(s_t, \tilde{s}_t^o)$, that optimizes a one-step cost while satisfying collision avoidance constraints. For instance, in the velocity obstacle approach [57], an agent chooses a collision-free velocity that is closest to its preferred velocity (i.e., directed toward its goal). Given this one-step nature, reaction-based methods do not anticipate the other agent’s hidden intent, but rather rely on a fast update rate to react quickly to the other agent’s motion. Although computationally efficient given these simplifications, reaction-based methods are myopic in time, which can sometimes lead to generating unnatural trajectories [15] (e.g., Fig. 5-5(b)).

Trajectory-based methods [15, 16, 60] solve (5.1)-(5.4) in two steps. First, the other agent’s hidden state is inferred from its observed trajectory, $\hat{s}_t^h = f(\tilde{s}_{0:t}^o)$, where $f(\cdot)$ is an inference function. Second, a centralized path planning algorithm, $\pi(s_{0:t}, \tilde{s}_{0:t}^o) = \pi_{central}(s_t, \tilde{s}_t^o, \hat{s}_t^h)$, is employed to find jointly feasible paths. By planning/anticipating complete paths, trajectory-based methods are no longer myopic. However, both the inference and the planning steps are computationally expensive, and need to be carried out online at each new observation (sensor update $\tilde{s}_t^o$).

Our approach uses a reinforcement learning framework to solve (5.1)-(5.4) by pre-computing a value function $V(s, \tilde{s}^o)$ that estimates the expected time to the goal. As a result, the proposed method offloads computation from the online planning step (as in trajectory-based methods) to an offline learning procedure. The learned value function enables the use of a computationally efficient one-step lookahead operation, which will be defined in (5.17) and explained in Section 5.2.
5.1.2 Reinforcement Learning

Reinforcement learning (RL) [87] is a class of machine learning methods for solving sequential decision making problems with unknown state-transition dynamics. Typically, a sequential decision making problem can be formulated as a Markov decision process (MDP), which is defined by a tuple \( M = (S, A, P, R, \gamma) \), where \( S \) is the state space, \( A \) is the action space, \( P \) is the state-transition model, \( R \) is the reward function, and \( \gamma \) is a discount factor. By detailing each of these elements and relating to (5.1)-(5.4), the following provides a RL formulation of the two-agent collision avoidance problem.

**State space** The system’s state is constructed by concatenating an agent’s full state and the observable part of the other agent’s state, \( s^{jn} = [s, \bar{s}^o] \in \mathbb{R}^{14} \).

**Action space** The action space is the set of permissible velocity vectors. Here, it is assumed that an agent can travel in any direction at any time, that is \( u(s) = v \) for \( \|v\|_2 < v_{pref} \). It is also straightforward to impose kinematic constraints, which will be explored in Section 5.2.4.
**Reward function**  A reward function is specified to award the agent for reaching its goal (5.3), and penalize the agent for getting too close or colliding with the other agent (5.2),

\[
R_{\text{col}}(s_t^n, u) = \begin{cases} 
-0.25 & \text{if } d_{\text{min}} < 0 \\
-0.1 - d_{\text{min}}/2 & \text{else if } d_{\text{min}} < 0.2 \\
1 & \text{else if } p = p_g \\
0 & \text{o.w.}
\end{cases}
\]  

(5.5)

where \(d_{\text{min}}\) is the minimum separation distance between the two agents within a duration of \(\Delta t\), assuming the agent travels at velocity \(v = u\), and the other agent continues to travel at its observed velocity \(\bar{v}\). Note that the separation \(d_{\text{min}}\) can be calculated analytically through simple geometry.

**State transition model**  A probabilistic state transition model, \(P(s_{t+1}^n, s_t^n | u_t)\), is determined by the agents’ kinematics as defined in (5.4). Since the other agent’s choice of action also depends on its policy and hidden state, the system’s state transition model is unknown. As in existing work [57], this work also assumes reciprocity \(\pi = \bar{\pi}\), which leads to the interesting observation that the state transition model depends on the agent’s learned policy.

**Value function**  The objective is to find the optimal value function

\[
V^*(s_0^n) = \mathbb{E} \left[ \sum_{t=0}^{T} \gamma^{t-v_{\text{pref}}} R(s_t^n, \pi^*(s_t^n)) \right],
\]

(5.6)

where \(\gamma \in [0, 1)\) is a discount factor. Recall \(v_{\text{pref}}\) is an agent’s preferred speed and is typically time invariant. It is used here as a normalization factor for numerical reasons, because otherwise the value function of a slow moving agent could be very
small. The optimal policy can be retrieved from the value function, that is
\[
\pi^*(s_{t+1}^n) = \arg\max_u R(s_t, u) + \gamma^{t+1} \int P(s_{t+1}^n|s_t^n, u) V^*(s_{t+1}^n) ds_{t+1}^n.
\] (5.7)

This work chooses to optimize \( V(s^n) \) rather than the more common choice \( Q(s^n, u) \), because unlike previous works that focus on discrete, finite action spaces [89,90], the action space here is continuous and the set of permissible velocity vectors depends on the agent's state (preferred speed).

5.2 Collision Avoidance with Deep RL

The following presents an algorithm for solving the two-agent RL problem formulated in Section 5.1.2, and then generalizes its solution (policy) to multiagent collision avoidance. Applications of RL are typically limited to discrete, low-dimensional domains, because classical dynamic programming-based algorithms (e.g., value iteration [87]) often rely on a tabular representation of the state space. However, for this domain, the joined state vector \( s^n \) is in a continuous 14 dimensional space, which would be too large to discrete and enumerate in a tabular form. Hence, this work leverages on the recent advances in Deep RL [89,90,115–117], which by using deep neural networks to parameter the value function, have demonstrated human-level performance in complex, high-dimensional spaces. This work employs a fully connected deep neural network with ReLU nonlinearities to parametrize the value function, as shown in Fig. 5-3(a). The value network is denoted by \( V(\cdot; \theta) \), where \( \theta \) is the set of weights in the neural network.

5.2.1 Parametrization

From a geometric perspective, there is some redundancy in the parameterization of the system's joint state \( s^n \), because the optimal policy should be invariant to any coordinate transformation (rotation and translation). To remove this ambiguity, an agent-centric frame is defined, with the origin at the agent's position, and the x-axis
Algorithm 7: CADRL (Coll. Avoidance with Deep RL)

1. **Input:** value network \( V() ; \theta \)
2. **Output:** trajectory \( s_{0:t_f} \)
3. **while not reached goal** do
   4. update \( t \), receive new measurements \( s_t, \dot{s}_t \)
   5. \( \hat{v}_t \leftarrow \text{filter}(\dot{v}_{0:t}) \)
   6. \( \hat{s}_{t+1} \leftarrow \text{propg}(\hat{s}_t, \Delta t \cdot \hat{v}_t) \)
   7. \( U \leftarrow \text{sampleActions()} \)
   8. \( u_t \leftarrow \argmax_{u_t \in U} R(s_t^{in}, u_t) + \gamma V(s_{t+1}, \hat{s}_{t+1}) \)
      where \( \gamma \leftarrow \gamma \Delta t \cdot \psi_{pref} \), \( \hat{s}_{t+1} \leftarrow \text{propg}(s_t, \Delta t \cdot u_t) \)
9. return \( s_{0:t_f} \)

pointing toward the agent's goal, that is,

\[
\begin{align*}
\mathbf{s}^{in'} & \triangleq [s', \tilde{s}'_t] = \text{rotate}(\mathbf{s}^{in}) \quad (5.8) \\
\text{s.t.} & \\
\mathbf{s}' & = [d_g, v_{pref}, v'_x, v'_y, \psi', r] \quad (5.9) \\
\tilde{s}'_t & = [\tilde{p}'_x, \tilde{p}'_y, \tilde{v}'_x, \tilde{v}'_y, \tilde{r}, \tilde{d}_a, \tilde{\alpha}', \tilde{b}_{on}] \quad (5.10)
\end{align*}
\]

where \( d_g = ||p_g - p||_2 \) is the agent's distance to goal, \( \tilde{d}_a = ||p - \tilde{p}||_2 \) is the distance to the other agent, \( \tilde{\alpha}' = \tan^{-1}(\tilde{v}'_y/\tilde{v}'_x) \) is the other agent's moving direction, and \( \tilde{b}_{on} \) is a binary flag indicating whether the other agent is real or virtual (details will be provided in Section 5.4.2). An illustration of this parametrization is shown in Fig. 5-2(a). Note that this agent-centric parametrization is only used when querying the neural network.

### 5.2.2 Generating Paths Using a Value Network

Given a value network \( V \), an RL agent can generate a path to its goal by repeatedly maximizing an one-step lookahead value (5.7), as outlined in Algorithm 7. This corresponds to choosing the action that on average, leads to the joint state with the highest value. However, the integral in (5.7) is difficult to evaluate, because the other agent's next state \( \hat{s}_{t+1} \) has an unknown distribution that depends on its unobservable intent. We approximate this integral by assuming that the other agent would be traveling at
Figure 5-3: Convergence of a Deep RL policy. (a) shows a neural network used to parameterize the value function. (b) shows the values of three distinct test cases converge as a function of RL training episodes. For example, the blue line corresponds to the initial state of the test case shown in the top row of Fig. 5-5.

A filtered velocity for a short duration $\Delta t$ (line 5-6). This work calculates the average velocity of the past 0.5 seconds and sets $\Delta t$ to 1.0 second. The use of a filtered velocity addresses a subtle oscillation problem as discussed in [56]. This propagation step amounts to predicting the other agent’s motion with a simple linear model, which has been shown to produce good accuracy over small time scales [118]. It is important to point out that this approximation is not assuming a linear motion model for $t > \Delta t$; uncertainty in the other agent’s future motion is captured in the projected next state’s value, $V(s_{t+1}, \hat{s}_{t+1}; \theta)$. Furthermore, the best action is chosen from a set of permissible velocity vectors (line 8). This work uses 30 pre-computed actions (e.g., directed toward an agent’s goal or current heading) and 5 randomly sampled actions.

An example of this one-step lookahead operation is visualized in Fig. 5-2(a), in which the red agent chooses the green velocity vector to cut behind the blue agent, because this action maximizes the value of the projected state shown in Fig. 5-2(b).

### 5.2.3 Training a Value Network

The training procedure, outlined in Algorithm 8, consists of two major steps. First, the value network is initialized by supervised training on a set of trajectories generated by a baseline policy (line 3). Specifically, each training trajectory is processed to
Algorithm 8: Deep V-learning for CADRL

1. **Input:** trajectory training set $D$
2. **Output:** value network $V(\cdot; \theta)$
3. $V(\cdot; \theta) \leftarrow \text{train\_nn}(D)$ \hspace{1cm} //step 1: initialization
4. duplicate value net $V' \leftarrow V$ \hspace{1cm} //step 2: RL
5. initialize experience set $E \leftarrow D$
6. for $episode=1, \ldots, N_{cps}$ do
   7. for $m$ times do
      8. $s_0, \tilde{s}_0 \leftarrow \text{randomTestcase}()$
      9. $s_{0:t_f} \leftarrow \text{CADRL}(V), \tilde{s}_{0:t_f} \leftarrow \text{CADRL}(V)$
     10. $y_{0:T}, \tilde{y}_{0:T} \leftarrow \text{findValues}(V', s_{0:t_f}, \tilde{s}_{0:t_f})$
     11. $E \leftarrow \text{assimilate}(E, (y, s^{in})_{0:t_f}, (\tilde{y}, \tilde{s}^{in})_{0:t_f})$
   12. $e \leftarrow \text{randSubset}(E)$
   13. $\theta \leftarrow \text{RMSprop}(e)$
   14. for every $C$ episodes do
      15. \hspace{1cm} $\text{Evaluate}(V), V' \leftarrow V$
16. return $V$

generate a set of state-value pairs, $\{(s^{in}, y)_{k}\}_{k=1}^{N}$, where $y = \gamma^t s^{v_{ref}}$ and $t_g$ is the time to reach goal. The value network is trained by back-propagation to minimize a quadratic regression error, $\sum_{k=1}^{N} (y_k - V(s_k^{in}; \theta))^2$. This work uses optimal reciprocal collision avoidance (ORCA) \cite{57} to generate a training set of 500 trajectories, which contains approximately 20,000 state-value pairs.

We make a few remarks about this initialization step. First, the training trajectories do not have to be optimal. For instance, two of the training trajectories generated by ORCA \cite{57} are shown in Fig. 5-4. The red agent was pinched by the blue agent and followed a large arc before reaching its goal. Second, the initialization training step is not simply emulating the ORCA policy. Rather, it learns a value function that encodes a time to goal estimate, which can then be used to generate new trajectories following Algorithm 7. Evidently, this learned value function sometimes generates better (i.e., shorter time to goal) trajectories than ORCA, as shown in Fig. 5-5(a). Third, this learned value function is likely to be suboptimal. For instance, the minimum separation $d_{min}$ between the two agents should be around 0.2m by (5.5), but there is too much slack as $d_{min}$ is greater than 0.4m in Fig. 5-5(a).
The second training step refines the policy through reinforcement learning. Specifically, a few random test cases are generated in each episode (line 8), and two agents are simulated to navigate around each other using an \( \epsilon \)-greedy policy, which selects a random action with probability \( \epsilon \) and follows the value network greedily otherwise (line 9). The simulated trajectories are then processed to generate a set of state-value pairs (line 10). For convergence reasons, as explained in [89], rather than being used to update the value network immediately, the newly generated state-value pairs are assimilated (replacing older entries) into a large experience set \( E \) (line 11).

Also, two value networks, \( V \) and \( V' \), are maintained during the training process. For each episode, a set of training points is randomly sampled from the experience set (line 12), which is used to update the parameters \( \theta \) in the value network \( V \) through RMSprop [119]. The duplicate value network \( V' \), which is used to generate the state-value pairs for training (line 10), is updated once in every \( C \) episodes (line 15).

To monitor convergence of the training process, the value network is tested regularly on a few pre-defined evaluation test cases (line 15), two of which are shown in Fig. 5-5. Note that the generated paths become tighter as a function of the number of training episodes, such that \( d_{\text{min}} \) reduced from 0.4m to 0.2m in the top row. Also,
the amount of jerky motion is reduced, as the trajectories in the bottom row became smoother over time. A plot of the test cases' values $V(s_0^n; \theta)$ shows that the value network has converged in approximately 800 episodes (Fig. 5-3(b)). It is important to point out that training/learning is performed on randomly generated test cases (line 8), but not on the evaluation test cases. The evaluation set is only used for debugging and monitoring convergence of the training process.

5.2.4 Incorporating Kinematic Constraints

Kinematics constraints need to be considered for operating physical robots. Yet, in many existing works, such constraints can be difficult to encode and might lead to a substantial increase in computational complexity [50,56]. In contrast, it is straightforward to incorporate kinematic constraints in the RL framework. For a skid-steering
Figure 5-6: A rotational kinematic constraint. Top left shows the same test case as in Fig. 5-2(a), but here the red agent chooses to slow down due to a rotational kinematic constraint (5.11). Bottom left shows the set of permissible velocity vectors for the red agent. Right shows a pair of sample trajectories generated by CADRL with rotational constraints, where the thin lines show the agents' heading angles. To minimize the time to goal, the red agent initially turns on the spot, and then starts moving (while continuing to turn) before its heading angle is aligned with its goal.

vehicle, we impose rotational constraints,

\[
\mathbf{u}(s) = [v_s, \phi] \quad \text{for} \quad v_s < v_{\text{pref}}, \quad |\phi - \psi| < \pi/6 \quad (5.11)
\]

\[
|\psi_{t+1} - \psi_t| < \Delta t \cdot v_{\text{pref}} \quad (5.12)
\]

where \(\phi\) and \(\psi\) denote the steering and heading angles, respectively. Notice that (5.11) limits the direction that an agent can travel, and (5.12) specifies a maximum turning rate that corresponds to a minimum turning radius of 1.0m. Figure 5-6(a) illustrates the application of these rotational constraints to the test case in Fig. 5-2(a). Here, the red agent chooses to slow down given the set of more constrained actions. Notice the agent is allowed to spin on the spot, which leads to an interesting optimal control problem when an agent's current heading angle is not perfectly aligned with its goal.
5.3 Social Norms

5.3.1 Characterizing Social Norms

It has been widely observed that humans tend to follow simple navigation norms to avoid colliding with each other, such as passing on the right and overtaking on the left [120]. Albeit intuitive, it remains difficult to quantify the precise mechanisms of social norms, such as when to turn and how much to turn when passing another pedestrian. As explained in Section 1.2.4, existing work often use featuring matching techniques such as max entropy [16] to characterize human behaviors from demonstration. However, such statistics can vary significantly between person to person, or even run to run, and the problem further exacerbates as the number of nearby pedestrians increases. The main challenge is how to characterize social norms when there is a lot of stochasticity in people’s motion.
Figure 5-8: Indications of a navigation convention emerging from the CADRL policy. Circles show each agent’s position at the labeled time, and stars mark the goals. (a) CADRL shows a preference to the right as two agents pass each other in a symmetrical test case (swapping positions). (b) This passing direction is robust to a small offset in the initial condition. (c) The passing direction changes at a larger offset (balancing with time to reach the goal). (d) The convention is not consistent with human social norms, as two agents of different sizes and velocities exhibit a preference to the left.

Rather than trying to quantify human behaviors directly, this work notes that the complex normative motion patterns can be a consequence of simple local interactions. For instance, an intuitive pairwise collision avoidance rule [61] can cause simulated agents moving in the same direction to form lanes in long corridors. Thus, we conjecture that rather than a set of precisely defined procedural rules, social norms are the emergent behaviors from a time-efficient, reciprocal collision avoidance mechanism. Reciprocity implicitly encodes a model of the other agents’ behavior, which is the key for enabling cooperation without explicit communication. Also note that reciprocity does not require a unique set navigation rules, since both the left-handed and right-handed rules can resolve path conflicts as shown in Fig. 5-7. Evidently, human navigation conventions are not unique, as the strength (separation distance)
and passing direction vary in different countries [121].

**Emergence of cooperative behaviors** Existing works have reported that human navigation (or teleoperation of a robot) tend to be cooperative and time-efficient [16, 68]. This work notes that these two properties are encoded in the CADRL formulation through using the min-time reward function and the reciprocity assumption ($\pi = \pi$). Furthermore, it was interesting to observe that while no behavior rules (e.g., function forms) were imposed in the problem formulation, CADRL policy exhibits certain navigation conventions, as illustrated in Fig. 5-8. In particular, Fig. 5-8(a) illustrates two CADRL agents passing on the right of each other, showing signs of conforming to mutually agreed rules. More importantly, this preference in passing direction is robust to small deviations in the initial condition, as shown in Fig. 5-8(b). As the offset increases, the CADRL agents eventually change passing direction in favor of shorter, smoother paths (Fig. 5-8(c)). Recall no communication took place and each agent’s intent is not known to the other.

However, the cooperative behaviors emerging from a CADRL solution are not consistent with the human interpretation of social norms. For instance, two CADRL agents with different sizes and preferred speeds show a preference to pass on the left of each other (Fig. 5-8(d)). This is because an agent’s state $s$ is defined to be the concatenation of its position, velocity, size and goal, so a tie breaking navigation convention should not be solely dependent on relative position (as for human social norms). Moreover, the cooperative behaviors of CADRL cannot be controlled – they are largely dependent on the initialization of the value network and the set of randomly generated training test cases. The following will address this issue to induce behaviors that respect human social norms.

### 5.3.2 Inducing Social Norms

Here presents socially aware collision avoidance with deep reinforcement learning (SA-CADRL), an extension to CADRL for capturing social norms. This work notes that social norms are one of the many ways to resolve a symmetrical collision avoidance
scenario, as illustrated in Fig. 5-7. To induce a particular norm, a small bias can be introduced in the RL training process in favor of one set of behaviors over others. For instance, to encourage passing on the right, states (configurations) with another agent approaching from the undesirable side can be penalized, such as for the green region in Fig. 5-9. The advantage of this approach is that violations of a particular social norm are easy to specify, and this specification need not be precise. This is because the addition of a penalty breaks the symmetry in the collision avoidance problem, thereby favoring behaviors respecting the desired social norm. This work specifies of a norm-inducing reward function,

\[
R_{\text{norm}}(s^{in}, u) = q_n I(s^{in} \in S_{\text{norm}})
\]

s.t. \( S_{\text{norm}} = S_{\text{pass}} \cup S_{\text{outk}} \cup S_{\text{cross}} \)

\[
S_{\text{pass}} = \{ s^{in} | d_g > 3, \ 1 < \tilde{p}_x' < 4,
-2 < \tilde{p}_y' < 0, \ |\tilde{\phi}' - \psi'| > 3\pi/4 \} \quad (5.14)
\]

\[
S_{\text{outk}} = \{ s^{in} | d_g > 3, \ 0 < \tilde{p}_x' < 3, \ |\nu| > |\tilde{\nu}|,
0 < \tilde{p}_y' < 1, \ |\tilde{\phi}' - \psi'| < \pi/4 \} \quad (5.15)
\]

\[
S_{\text{cross}} = \{ s^{in} | d_g > 3, \ \tilde{d}_a < 2, \ \tilde{\phi}_{\text{rot}} > 0,
-3\pi/4 < \tilde{\phi}' - \psi' < -\pi/4 \} , \quad (5.16)
\]
where $q_n$ is a scalar penalty, $I(\cdot)$ is the indicator function, $\bar{\phi}_{rot} = \tan^{-1}((\bar{v}_y' - \bar{v}_y)/\bar{v}_x')$ is the relative rotation angle between the two agents, and the heading angle $\tilde{\phi}' - \psi'$ is wrapped between $[-\pi, \pi]$. An illustration of these three penalty sets is provided in Fig. 5-9, where a penalty is issued when a neighboring agent (black) resides in any of the shaded regions. The current specification corresponds to the right-handed rules, and it could modified (e.g., reverse the signs) to specify the left-handed rules. Note that the overall reward function is the sum of the original CADRL reward and the norm-inducing reward, that is, $R(\cdot) = R_{cad}(\cdot) + R_{\text{norm}}(\cdot)$.

The parameters defining the penalty set $S_{\text{norm}}$ affect the rate of convergence. With the parameters setting in (5.14)-(5.16), the SA-CADRL policy converged within 700 episodes. In particular, from this episode and onwards, the trajectories for all validation test cases exhibited the desired behaviors such as passing on the right. With a 30% smaller penalty set by shrinking the shaded regions in Fig. 5-9, convergence occurred after 1250 episodes. Larger penalty sets, however, could lead to instability or divergence. Also, as long as training converges, the size of the penalty sets does not have a major impact on the learned policy. This is expected because the desired behaviors are not in the penalty set. Starting from the same initialization, we trained two SA-CADRL policies corresponding to left-handed and right-handed norm respectively, and the results are shown in Fig. 5-10. The learned policies exhibited similar qualitative behaviors as shown in Fig. 5-7. Also note that training is performed on randomly generated test cases, and not on the validation test cases.

5.4 Scaling Up to Multiagent Systems

The following describes two approaches for generalizing CADRL to multiagent systems with more than two agents. The first approach uses a minimax step to adapt the two-agent value network to multiagent scenarios. The second approach develops a novel multiagent network structure, which is used to train/parametrize a multiagent policy directly. For nomenclature, since Approach I uses the two-agent network and requires no additional training, it inherits the name CADRL. Conversely, Approach II
Figure 5-10: SA-CADRL policies exhibiting socially aware behaviors. Circles show each agent's position at the labeled time, and stars mark the goals. (a) and (b) show the trajectories generated by SA-CADRL policies trained to learn left-handed and right-handed rules, respectively. These behaviors are time-efficient and agree with the qualitative characterization of social norms shown in Fig. 5-7.
inherits the name SA-CADRL. A comparison of the two approaches will be provided in Section 5.5.3.

5.4.1 Approach I: Using a Two-agent Value Network

Let \( \tilde{s}_i^o \) denote the observable part of the \( i \)th neighbor's state, and \( s^{in,i} = [s, \tilde{s}^{o,i}] \) denote a vehicle's joint state with the \( i \)th neighbor. CADRL (Algorithm 7) can be extended to \( n > 2 \) agents by propagating every neighbor's state one step forward (line 5-6), and then selecting the action that has the highest value with respect to any neighbor's projected state \( \hat{s}_{t+1}^{o,i} \); that is, replace line 8 with

\[
\arg\max_{u_t \in U} \min_i \ R(s^{in,i}_t, u_t) + \gamma^{\Delta t} v_{pref} V(\hat{s}_{t+1}, \hat{s}_{t+1}^{o,i}).
\]  

(5.17)

Although using a two-agent value network, CADRL can produce multiagent trajectories that exhibit complex interaction patterns. Figure 5-11(a) shows three pairs of agents swapping positions on a circle. The agents veer more than the two-agent case (top row of Fig. 5-5), which makes more room in the center to allow everyone to pass smoothly. Figure 5-11(b) shows three pairs of agents swapping positions horizontally, where the pair in the center slows down near the origin so the outer agents can pass first. Both cases demonstrate that CADRL can produce smooth, natural looking trajectories for multiagent systems.

5.4.2 Approach II: Training a Multiagent Network

Since Approach I used a value network trained on a two-agent system, it was difficult to encode/induce higher order behaviors, such as considering the relations between nearby agents. Approach II addresses this problem by developing a method that allows for training on multiagent scenarios directly.

To capture the multiagent system's symmetrical structure, a neural network with weight-sharing and max-pooling layers is employed, as shown in Fig. 5-12. In particular, for a four-agent network shown in Fig. 5-12(b), the three nearby agents' observed states can be swapped (blue input blocks) without affecting the output value. This
Figure 5-11: Multiagent trajectories produced by CADRL. Circles show each agent’s position at the labeled time, and stars mark the goals. Although CADRL uses a two-agent value network for multiagent scenarios (5.17), more complex interaction patterns have emerged. In particular, although both test cases involve three pairs of agents swapping position, each agent follows a path much different from the two agent case shown in the top row of Fig. 5-5.

condition is enforced through weight-sharing, as shown in Fig. 5-12(a). Two of such symmetrical layers are used, followed by a max-pooling layer for aggregating features and two fully-connected layers for computing a scalar value. Here also uses the rectified linear unit (ReLU) as the activation function in the hidden layers.

The input to the n-agent network is a generalization of (5.9)-(5.10), that is, \( s^{in} = [s, s^{o,1}, \ldots, s^{o,n-1}] \), where the superscripts enumerate the nearby agents. The norm-inducing reward function is defined similarly as (5.13), where a penalty is given if an agent’s joint configuration with the closest nearby agent belongs to the penalty set \( S_{norm} \).

The procedure for training a multiagent SA-CADRL policy is outlined in Algorithm 8, which follows similarly as Algorithm 8. A value network is first initialized by training on an n-agent trajectory dataset through neural network regression (line 1). Using this value network (5.7) and following an \( \epsilon \)-greedy policy, a set of trajectories can be generated on random test cases (line 5-7). The trajectories are then turned into state-value pairs and assimilated into the experience sets \( E, E_b \) (line 10-11). A subset of state-value pairs is sampled from the experience sets, and subsequently used to update the value network through back-propagation (line 12-13). The process
(a) Weight matrix

(b) Symmetric multiagent net

Figure 5-12: A network structure for multiagent scenarios. (a) illustrates the weight matrix used in the first layer, where block partitions with the same color are constrained to be the same. (b) shows the overall network structure, with two symmetric layers, one max-pooling layer, and two fully connected layers. The colored connections in the first layer correspond to (a), where red are the weights associated with the agent itself, and blue are that of the nearby agents.

repeats for a pre-specified number of episodes (line 3-4).

Compared with Algorithm 8, two important modifications are introduced in the training process. First, two experience sets, $E$, $E_b$, are used to distinguish between trajectories that reached the goals and those that ended in a collision (line 2, 11). This is because a vast majority ($\geq 90\%$) of the random generated test cases were fairly simple, requiring an agent to travel mostly straight towards its goal. The bad experience set $E_b$ improves the rate of learning by focusing on the scenarios that fared poorly for the current policy. Second, during the training process, trajectories generated by SA-CADRL are reflected in the x-axis with probability $\epsilon_f$ (line 8). By inspection of Fig. 5-7, this operation flips the paths’ topology (left-handedness vs right-handedness). Since a trajectory can be a few hundred steps long according to (5.7), it could take a long time for an $\epsilon$-greedy policy to explore the state space and find an alternative topology. In particular, empirical results show that, without this procedure, policies can still exhibit the wrong passing side after 2000 training episodes. This procedure exploits symmetry in the problem to explore different topologies more efficiently.

Furthermore, an $n$-agent network can be used to generate trajectories for scenarios
Algorithm 9: Deep V-learning for SA-CADRL

1 initialize and duplicate a value net with $n$ agents $V(\theta; \tau, n)$, $V' \leftarrow V$
2 initialize experience sets $E \leftarrow \emptyset$, $E_b \leftarrow \emptyset$
3 for episode $= 1, \ldots, N_{\text{ep}}$ do
4     for $m$ times do
5         $p \sim \text{Uniform}(2, n)$
6         $s_0^1, s_0^2, \ldots, s_0^p \leftarrow \text{randomTestcase}(p)$
7         $s_{0:tf}^1, s_{0:tf}^2, \ldots, s_{0:tf}^p \leftarrow \text{SA-CADRL}(V)$
8         with prob $\epsilon_f$, mirror every traj $s_{0:tf}^i$ in the x-axis
9         for every agent $i$ do
10            $y_{0:T}^i \leftarrow \text{findValues}(V', s_{0:tf}^{j_{tn}^i})$
11            $E, E_b \leftarrow \text{assimilate}(E, E_b, (y^i, s_{0:tf}^{j_{tn}^i}))$
12        end
13        $e \leftarrow \text{randSubset}(E) \cup \text{randSubset}(E_b)$
14        $\theta \leftarrow \text{RMSprop}(e)$
15        for every $C$ episodes do
16            Evaluate($V$), $V' \leftarrow V$
17    end
18 return $V$

with fewer agents (line 5). In particular, when there are $p < n$ agents, the inputs in Fig. 5-12(b) corresponding to the non-existent agents can be filled by adding virtual agents – replicating the states of the closest nearby agent and set the binary bit $b_{on}$ to zero (5.10). The use of this parametrization avoids the need for training many different networks. A left-handed and a right-handed four-agent SA-CADRL policies are trained using the network structure shown in Fig. 5.12. Sample trajectories generated by these policies are shown in Fig. 5.13, which demonstrate the preferred behaviors of each respective set of social norms.

5.5 Results

5.5.1 Computational Time

Training a two-agent CADRL network This work uses a neural network with three hidden layers of width $(150, 100, 100)$ (Fig. 5-3(a)), whose size is chosen to balance between computational time and performance. In particular, we also exper-
Figure 5-13: SA-CADRL policies generalized to multiagent scenarios using the network structure in Fig. 5-12. Circles show each agent's position at the labeled time, and stars mark the goals. (a) and (b) show trajectories corresponding to the left-handed and right-handed rules, respectively.

implemented with other network sizes. For instance, a three layer network of double the width (300, 200, 200) produced similar results (paths) but was almost twice as slow. On a computer with an i7-5820K CPU, a Python implementation of CADRL (Algorithm 7), takes an average of 5.7ms per iteration on for a two-agent system. By inspection of (5.17), computational time scales linearly in the number of neighboring agents for a decentralized implementation where each agent runs CADRL individually; and scales quadratically in a centralized implementation where one computer controls all agents. For decentralized control of ten agents, each iteration of CADRL is observed to take 62ms. Moreover, CADRL is parallelizable because it consists of a
large number of independent queries of the value network (5.17).

Offline training (Algorithm 8) took less than three hours and is found to be quite robust. In particular, using mini-batches of size 500, the initialization step (line 3) took 9.6 minutes for 10,000 iterations. The RL step used an \( \epsilon \)-greedy policy, where \( \epsilon \) decays linearly from 0.3 to 0.1 in the first 400 training episodes, and remains 0.1 thereafter. The RL step took approximately 2.5 hours to complete 1,000 training episodes. The entire training process was repeated on three sets of training trajectories generated by ORCA, and the value network converged in all three trials. The paths generated by the value networks from the three trials were very similar, as indicated by a less than 5% difference in time to reach goal on all test cases in the evaluation set.

**Training a four-agent SA-CADRL network**  As above, the size and connections in the multiagent network shown in Fig. 5-12 are tuned to obtain good performance while achieving real-time performance. On a computer with an i7-5820K CPU, a Python implementation of a four-agent SA-CADRL policy takes on average 8.7ms for each query of the value network. Furthermore, offline training (Algorithm 9) took approximately nine hours to complete 3,000 episodes. The four-agent system took much longer to train because its state space is much larger (higher dimensional) than that of the two-agent system. The training process was repeated multiple times and all runs converged to a similar policy – exhibiting the respective desired social norms on all test cases in an evaluation set. Numerical results will be presented in Table 5.2.

### 5.5.2 Performance Evaluation of CADRL

To evaluate the performance of CADRL over a variety of test cases, we compute the average extra time spent to reach the goals, that is

\[
\bar{t}_e = \frac{1}{n} \sum_{i=1}^{n} \left[ t_{i,g} - \frac{||P_{i,0} - P_{i,g}||_2}{v_{i,pref}} \right],
\]  

(5.18)
where \( t_{i,g} \) is the \( i \)th agent's time to reach its goal, and the second term is a lower bound of \( t_{i,g} \) (straight toward goal at the preferred speed). This metric removes the effects due to variability in the number of agents and the nominal distance to goals.

**A crossing scenario** A crossing scenario is shown in Fig. 5-14(a), where two identical agents with goals along collision courses are run into each other at different angles. Thus, cooperation is required for avoiding collision at the origin. Figure 5-14(b) shows that over a wide range of angles (\( \alpha \in [90, 150] \) deg), agents following CADRL reached their goals much faster than that of ORCA. Recall a minimum separation of 0.2m is specified for CADRL in the reward function (5.5). For this reason, similar to the bottom row of Fig. 5-5, CADRL finds paths that are slightly slower than ORCA around \( \alpha = 0 \). It is also interesting to note that CADRL with rotational constraints (CADRL w/ cstr) performs slightly better than the unconstrained. This is because CADRL w/ cstr is more conservative (yielding) early on, which is coincidentally good for the crossing scenario. More specifically, if the other agent has stopped (reached goal) or turned before it reached the origin, unconstrained CADRL would have performed better. In short, as will be shown later in Fig. 5-16, unconstrained CADRL is better on average (randomized test cases), but can be slightly worse than CADRL w/ cstr on particular test cases.

**Random test cases** In addition to showing that CADRL can handle some difficult test cases that fared poorly for ORCA (Figs. 5-5 and 5-14), a more thorough evaluation is performed on randomly generated test cases. In particular, within square shaped domains specified in Table 5.1, agents are generated with randomly sampled speed, radius, initial positions and goal positions. This work chooses \( v_{pref} \in [0.5, 1.5] \text{m/s}, \ r \in [0.3, 0.5] \text{m}, \) which are parameters similar to that of typical pedestrian motion. Also, the agents' goals are projected to the boundary of the room to avoid accidentally creating a trap formed by multiple stationary agents. A sample four-agent test case is illustrated in Fig. 5-15, where agents following CADRL were able to reach their goal much faster than that of ORCA. For each configuration set-
Figure 5-14: Performance comparison on a crossing scenario. (a) shows the crossing configuration, where a red agent travels from left to right, and a blue agent travels at a diagonal angled at $\alpha$. Both agents have a radius of 0.3m and a preferred speed of 1.0m/s, and they would collide at the origin if both travel in straight lines. (b) compares the extra time spent to reach the goal using different collision avoidance strategies. CADRL performs significantly better than ORCA on a wide range of angles $\alpha \in [90, 150]$ deg.

In Table 5.1, one hundred test cases are generated as described above. ORCA, CADRL, and CADRL w/cstr are employed to solve for these test cases. The average extra time to reach goal, $\tilde{t}_e$, is computed for each set of generated trajectories and plotted in Fig. 5-16. Key statistics are computed and listed in Table 5.1, and it can be seen that CADRL performs similarly (slightly better) than ORCA on the easier test cases (median), and more than 26% better on the hard test cases (>75 percentile). Also, performance difference is more clear on test cases with more agents, which could be a result of more frequent interactions.

Navigating around non-cooperative agents Recall CADRL’s value network is trained with both agents following the same collision avoidance strategy, which is the reciprocity assumption common to many existing works [56, 57]. Figure 5-17 shows that the proposed method can also be used to navigate efficiently around non-CADRL agents. Figure 5-17(a) shows a CADRL agent navigating static obstacles modeled as stationary agents ($\bar{v}_i = 0$). We acknowledge that CADRL could get stuck in a dense obstacle field, where traps/dead-ends could form due to positioning of multiple obstacles. Recall CADRL is a collision avoidance (not path planning)
Figure 5-15: A four-agent test case. (a) shows agents following ORCA traversed long arcs before reaching their goals, which reflects the similar two-agent problem shown in Fig. 5-4(b). (b) shows agents following CADRL were able to reach their goals much faster.

algorithm not designed for such scenarios. Figure 5-17(b) shows a CADRL agent navigating around a non-cooperative agent (black), who traveled in a straight line from right to left. In comparison with the cooperative case shown in Fig. 5-5(c), here the red CADRL agent chooses to veer more to its left to avoid collision.

5.5.3 Socially Aware Navigation of SA-CADRL

Three copies of four-agent SA-CADRL policies were trained, one without the norm inducing reward $R_{\text{norm}}$, one with the left-handed $R_{\text{norm}}$, and the other with the right-handed $R_{\text{norm}}$. On perfectly symmetrical test cases, such as those shown in Figs. 5-8 and 5-10, the left and right-handed SA-CADRL policies always select the correct passing side according to the respective norm. To demonstrate SA-CADRL can balance between finding time-efficient paths and respecting social norms, these policies are further evaluated on randomly generated test cases. In particular, we compute the average extra time to reach the goals, the minimum separation distance, and the relative preference between left-handedness and right-handedness. Norm preference is calculated by counting the proportion of trajectories that violate the left-handed or the right-handed version of (5.14)-(5.16) for more than 0.5 second. To ensure the test set is left-right balanced, each random test case is duplicated and reflected in
Figure 5-16: Performance comparison on random test cases. The extra time to goal $t_e$ is computed on one hundred random test cases for each configuration listed in Table 5.1. (a) and (b) show scatter plots of the raw data, and (c) shows a box-whisker plot. CADRL is seen to perform similarly (slightly better) to ORCA on the easier test cases (median), and significantly better on the more difficult test cases (>75 percentile).

Evidently, the optimal reciprocal collision avoidance (ORCA) [57] algorithm — a reactive, rule-based method that computes a velocity vector based on an agent’s joint geometry with its neighbors — attains nearly 50-50 left/right-handedness on these test sets, as shown in the first row of Table 5.2.

The same four-agent SA-CADRL policies are used to generate trajectories for both the two-agent and the four-agent test sets. On the two-agent test-set, all RL-based methods produced more time-efficient paths than ORCA. CADRL exhibited a slight preference to the right (40-60 split). The four-agent SA-CADRL (none) policy, in comparison, exhibited a stronger preference than ORCA and CADRL in each of the passing, crossing, and overtaking scenarios (third row in Table 5.2). This
observation suggests that (i) certain navigation conventions could emerge as a means of resolving symmetrical collision avoidance scenarios, and (ii) the conventions don’t always correspond to human social norms. For instance, SA-CADRL (none) prefers passing on the right but also overtaking on the right, which is a mix between right-handed and left-handed rules. In contrast, the SA-CADRL policies trained with $R_{norm}$ exhibited a strong preference (85-15 split) of the respective social norm. Recall that this ratio is not 1 because there is a tradeoff between time-optimality and social compliance, as illustrated in Fig. 5-8. This tradeoff can be controlled by tuning $q_n$ in (5.13). Evidently, SA-CADRL (lh/rh) achieves better social compliance at a cost of an approximately 20% larger $\bar{t}_n$, because satisfying the norms often requires traveling a longer path.

Similarly, the bottom three rows of Table 5.2 show that in the four-agent test set, all RL-based methods outperformed ORCA, and SA-CADRL (lh/rh) exhibited behaviors that respect the social norms. CADRL produced paths that are closer to time-optimal than the other algorithms, but sometimes came very close (within 0.1m) to other agents. This close proximity occurred because CADRL was trained on a two-agent system, so its action choice is dominated by the single closest neighbor. This possibly causes CADRL to select an action that avoids the closest neighbor but drives
5.5.4 Hardware Experiment

The SA-CADRL policy is implemented on the Jackal robot (Fig. 2-1(b)) to demonstrate autonomous navigation. The robot was run in two indoor environments as shown in Fig. 5-18. The first environment is a controlled lab space for testing various multiagent scenarios, which include both robot-robot and human-robot interactions. The second environment is a large building on campus that is open to the public, which contains many pedestrians as part of the normal traffic. The robot is standalone as everything from localization, pedestrian detection, to motion planning is run onboard of the vehicle.

A controlled lab space  The lab space in Fig. 5-18(a) is a rectangular area of approximately 4m × 7m. Figure 5-19(a) shows two robots swapping positions at a
nominal speed of 1.0 m/s, where both vehicles veered to the side to share the responsibility for collision avoidance. Figure 5-19(b) shows a crossing scenario. Notice the agent starting at the bottom slows down and goes behind the other. A comparison between this scenario and that in Fig. 5-1 demonstrates that SA-CADRL addresses the issue therein. Figure 5-19(c) shows an example of human-robot interaction without explicit communication. The robot demonstrated cooperative behaviors by moving to its left initially, thereby shared the load for collision avoidance. A
Figure 5-20: Visualization of a ground vehicle's sensor data. The vehicle (yellow box) uses the right-handed SA-CADRL policy to navigate autonomously in an indoor environment at a nominal speed of 1.2m/s. (a) shows the vehicle passing a pedestrian on the right, where the person in the front camera image is detected and shown as the blue cylinder in the rviz view. (b) shows the vehicle overtaking a pedestrian on the left, where the person in the right camera image corresponds to the teal cylinder in the rviz view.

A busy campus building  This is the same environment as that in Fig. 4-10(b), which is a public area that spans more than 100 meters. Here, motion planning uses the diffusion map algorithm (Algorithm 5) for finding global paths, and SA-CADRL for local collision avoidance. In particular, the diffusion map algorithm considers static obstacles in the environment to find a subgoal within the vehicle's planning horizon (5m). The subgoal is the location with the lowest diffusion distance to goal as shown at the end of the green line in Fig. 5-20(a). Also, a set of feasible directions (heading and range) is computed, which corresponds to the free space in the white occupancy map. The feasible directions are visualized as the thin blue lines emanating from the vehicle. SA-CADRL takes in the set of detected pedestrians, and chooses an action (velocity vector) from the feasible directions to move the vehicle toward the subgoal. SA-CADRL's decision is shown as the blue arrow in Fig. 5-20(a), which does not necessarily line up with the subgoal. Note that pedestrians can be detected beyond...
the planning horizon, thus allowing socially aware interaction at a longer range. The whole sense-plan-execute cycle is fast enough to operate in real-time at 10Hz on a Gigabyte Brix computer onboard the vehicle.

Using this motion planning strategy, the vehicle was able to navigate fully autonomously in a dynamic indoor environment. In particular, the vehicle is issued randomly generated goals ten times, with an average distance between successive goals of more than 50 meters. During the experiment, an average of 10.2 persons came within 2m of the vehicle each minute. All encountered pedestrians are part of the regular daily traffic, and not testers/personnel associated with this work. For example, there are undergraduate students going between lectures and tourists visiting the campus. At a nominal speed of 1.2m/s, which is approximately the average human walking pace [91], the vehicle maintained safe distance to the pedestrians and generally respected social norms. While a safety driver was walking approximately five meters behind vehicle, he never had to intervene or take over control at any time during the ten runs. Since people in North America follow the right-handed rule, the SA-CADRL policy with right-handed norms is used for the hardware experiment, which causes the vehicle to generally pass pedestrians on the right and overtake on the left. For examples, snippets of the experiment are shown in Fig. 5-20. A hardware demonstration video can be found at https://youtu.be/PMTnUJWeEsA.

5.6 Summary

This chapter developed a decentralized multiagent collision avoidance (CADRL) algorithm based on a novel application of deep reinforcement learning. In particular, a pair of agents were simulated to navigate around each other to learn a value network that encodes the expected time to goal. Furthermore, the algorithm was extended (SA-CADRL) to model and learn human navigation norms, such as passing on the right and overtaking on the left. This approach was further generalized to multiagent (n > 2) scenarios through the use of a symmetrical neural network structure. Simulation results showed more than 26% improvement in paths quality when compared
with ORCA, a state-of-the-art reaction-based collision avoidance approach. Moreover, SA-CADRL was implemented on robotic hardware, which enabled fully autonomous navigation at the average human walking pace in a dynamic environment with many pedestrians.
Table 5.1: CADRL’s extra time to reach goal ($t_e$) statistics on the random test cases shown in Fig. 5-16. CADRL finds paths that on average, reach goals much faster than that of ORCA. The improvement is more clear on hard test cases (>75 percentile) and in multiagent ($n > 2$) scenarios that require more interactions.

<table>
<thead>
<tr>
<th>Test case configuration</th>
<th>Extra time to goal $t_e$ (s)</th>
<th>Average min separation dist. (m)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[Avg / 75th / 90th percentile]</td>
<td>[10th percentile / avg]</td>
</tr>
<tr>
<td></td>
<td>ORCA</td>
<td>CADRL</td>
</tr>
<tr>
<td>2</td>
<td>0.46 / 0.45 / 0.73</td>
<td><strong>0.27</strong> / <strong>0.33</strong> / <strong>0.56</strong></td>
</tr>
<tr>
<td>4</td>
<td>0.69 / 0.85 / 1.85</td>
<td><strong>0.31</strong> / <strong>0.40</strong> / <strong>0.76</strong></td>
</tr>
<tr>
<td>6</td>
<td>0.65 / 0.83 / 1.50</td>
<td><strong>0.44</strong> / <strong>0.56</strong> / <strong>0.87</strong></td>
</tr>
<tr>
<td>8</td>
<td>0.96 / 1.33 / 1.84</td>
<td><strong>0.54</strong> / <strong>0.70</strong> / <strong>1.01</strong></td>
</tr>
</tbody>
</table>

Table 5.2: SA-CADRL’s performance statistics on randomly generated test cases. SA-CADRL policies were trained (i) without the norm inducing reward $R_{norm}$, (ii) with left-handed $R_{norm}$, and (iii) right-handed $R_{norm}$, which are abbreviated as none, lh, rh, respectively. Results show that SA-CADRL policies produced time-efficient paths and exhibited behaviors that respect the corresponding social norm.

<table>
<thead>
<tr>
<th>num of agents in test cases</th>
<th>Method</th>
<th>Extra time to goal $t_e$ (s)</th>
<th>Min separation dist. (m)</th>
<th>Norm preference (%)</th>
<th>[lh / rh]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>[Avg / 75th / 90th percentile]</td>
<td>[10th percentile / avg]</td>
<td>passing</td>
<td>crossing</td>
</tr>
<tr>
<td>2</td>
<td>ORCA [57]</td>
<td>0.46 / 0.49 / 0.82</td>
<td>0.108 / 0.131</td>
<td>45 / 55</td>
<td>51 / 49</td>
</tr>
<tr>
<td></td>
<td>CADRL [64]</td>
<td><strong>0.25</strong> / <strong>0.30</strong> / <strong>0.47</strong></td>
<td>0.153 / 0.189</td>
<td>37 / 63</td>
<td>38 / 62</td>
</tr>
<tr>
<td></td>
<td>SA-CADRL(none)</td>
<td>0.27 / 0.28 / 0.54</td>
<td><strong>0.169</strong> / <strong>0.189</strong></td>
<td>10 / 90</td>
<td>32 / 68</td>
</tr>
<tr>
<td></td>
<td>SA-CADRL(lh)</td>
<td>0.30 / 0.36 / 0.67</td>
<td>0.163 / 0.192</td>
<td>98 / 2</td>
<td>85 / 15</td>
</tr>
<tr>
<td></td>
<td>SA-CADRL(rh)</td>
<td>0.31 / 0.38 / 0.69</td>
<td>0.168 / 0.199</td>
<td>2 / 98</td>
<td>15 / 85</td>
</tr>
<tr>
<td>4</td>
<td>ORCA [57]</td>
<td>0.86 / 1.14 / 1.80</td>
<td>0.106 / 0.125</td>
<td>46 / 54</td>
<td>50 / 50</td>
</tr>
<tr>
<td></td>
<td>CADRL(minimax) [64]</td>
<td><strong>0.41</strong> / <strong>0.54</strong> / <strong>0.76</strong></td>
<td>0.096 / 0.173</td>
<td>31 / 69</td>
<td>41 / 59</td>
</tr>
<tr>
<td></td>
<td>SA-CADRL(none)</td>
<td>0.44 / 0.63 / 0.85</td>
<td><strong>0.162</strong> / <strong>0.183</strong></td>
<td>33 / 67</td>
<td>33 / 67</td>
</tr>
<tr>
<td></td>
<td>SA-CADRL(lh)</td>
<td>0.49 / 0.69 / 1.00</td>
<td>0.155 / 0.178</td>
<td><strong>83</strong> / 17</td>
<td>67 / 33</td>
</tr>
<tr>
<td></td>
<td>SA-CADRL(rh)</td>
<td>0.46 / 1.63 / 1.02</td>
<td>0.155 / 0.180</td>
<td>12 / 88</td>
<td>29 / 71</td>
</tr>
</tbody>
</table>
Chapter 6

Conclusions and Future Work

This thesis presented a strategy for autonomous navigation in dynamic environments cohabited by other mobile agents, such as other vehicles and pedestrians. In particular, this work considered scenarios without explicit communication of each agent’s intent (e.g., goals, paths), which is a constraint common in practice due to hardware limitations. In such environments, finding collision-free paths around other agents is challenging because it may require predicting the others’ motion and anticipating interaction patterns, through a process that needs to be computationally tractable for real-time implementation.

A predictive motion modeling framework named ASNSC was introduced in Chapter 3. Building on dictionary learning, ASNSC seeks to find a succinct representation of each agent’s motion (trajectory) as a concatenation of local motion patterns. Specifically, given a training set of mobile agents’ trajectories, ASNSC formulates an optimization problem to solve for the local motion patterns and the corresponding transition probabilities. It is important to contrast ASNSC with the existing clustering-based motion modeling methods, which find global motion patterns by grouping the training set into a few clusters. This work noted that when working with domains larger than the field of view of fixed sensors (e.g., wall-mount cameras), trajectories are often similar in some parts while different in others, which leads to some inherent ambiguity for finding a global clustering configuration. ASNSC overcomes this issue by finding local motion patterns. Moreover, an algorithm was
developed to solve ASNSC, and demonstrated the ability to generate accurate path predictions in real time.

Chapter 4 and 5 explored motion planning in dynamic environments. The former was focused on efficient re-planning in response to frequent changes in the environment, and latter was focused on navigating around other dynamic agents. Specifically, Chapter 4 presented a path planning algorithm based on a novel application of diffusion map. Assuming knowledge of a map, this work would learn the geometry of the domain by computing a coordinate transformation to a higher dimensional diffusion space. The Euclidean distance in the diffusion space, known as the diffusion distance, is shown to be a meaningful pairwise similarity metric. The proposed motion planning algorithm uses the diffusion distance as a potential function to accelerate path finding, which often leads to an order of magnitude improvement in computation time compared with classical single-query methods. Moreover, drawing insights from the connections between diffusion map and spectral clustering, an algorithm was developed to find multiple routes (topologically distinct paths) in response to perceived local traffic congestion.

Finally, Chapter 5 introduced CADRL, an algorithm for decentralized multiagent collision avoidance. The key challenge is to develop a cooperative coordination rule that yields smooth paths while maintaining computational tractability. This is difficult because generating a smooth path would typically entail predicting all nearby agents' motion (and the interaction pattern), which can be computationally prohibitive. CADRL addresses this issue by learning a value network that minimizes the expected time to goal. This training process effectively offloads the expensive online computation (explicit motion prediction) to an offline learning procedure. Furthermore, CADRL was extended to capture human navigation norms, such as passing on the right and overtaking on the left. Finally, CADRL was combined with diffusion maps from Chapter 4 to form an integrated strategy for autonomous navigation in dynamic environments. This work culminated in a hardware demonstration of a robotic vehicle navigating fully autonomously in a large public building with many pedestrians.
6.1 Future Work

The following describes several directions for extending this work.

**Context-aware motion prediction** Recall the ASNSC framework developed in Chapter 3 would find motion patterns from a set of tracked trajectories. At the time when ASNSC was developed, each trajectory is defined to be a sequence of time-stamped position measurements. As a result, the learned motion patterns are tied to the physical locations where the trajectories were collected. Yet, the past two years have seen significant improvements in object detection due to the advances in deep learning. For instance, modern perception algorithms can segment camera images, extract pedestrians’ pose, and detect environmental features [4] such as sidewalks and traffic lights in real time. The availability of such contextual information opens up exciting new opportunities to expand the ASNSC framework. For example, a discrete feature vector specifying the type of the environmental structure at each point on the trajectory can be appended to the data representation. This could allow ASNSC to associate motion patterns with environmental features, such as learning that pedestrians are more likely to remain on the sidewalks and slow down when approaching an intersection. Moreover, contextual information could be used to explore transfer learning, such that a set of motion patterns learned at one intersection can be transformed (stretched) to form a prior at a different location.

**Dynamic, hierarchical diffusion maps for motion planning** The current implementation of motion planning with diffusion maps requires tuning a time scale parameter, which corresponds to the number of steps in a Markov random walk as revealed in (2.15). By inspection of Fig. 4-4, this time scale parameter should be set small for pairs of nodes that are close together (small geodesic distance), and large for those far apart. Future work could investigate whether the time scale could be varied dynamically (continuously) when planning a path. Not only would this modification save the parameter tuning time, but it might also provide insights into the local oscillation problem described in Section 4.2.1 and thereby improve the quality
of the planned paths.

Furthermore, Section 4.2.3 have started exploring a two-level representation of maps, which is shown to be useful for finding topologically distinct paths. It would be useful to generalize this idea to develop a hierarchical multi-level representation in the diffusion space. This could allow for dynamically updating a portion (or a level) of the map in response to changes in the environment. Since computing a full diffusion map still takes quite some time (Table 4.1), the ability to update sections of the map individually would make large domains much more manageable.

**Lifelong learning with real-world data** Recall the SA-CADRL approach introduced in Chapter 5 would learn a policy by simulating a group of agents navigating around each other. While hardware experiments demonstrated smooth paths and safe interaction with nearby pedestrians, the policy could be further improved by incorporating real-world data. In particular, the observation models used in the simulator are often incorrect, such as assuming perfect pedestrian tracking within the vehicle’s perception range. In reality, there is some noise associated with position tracking and there could be occlusion due to the presence of multiple pedestrians. In addition, as noted in [121], navigation norms vary in direction and strength (e.g., separation distance) in different regions of the world. Also, the environment can be non-stationary, as people’s reaction (trust) to an autonomous robot often change over time. Hence, the discrepancy between the simulator and the real world calls for a more general framework for lifelong learning, such that the algorithm needs to be able to adapt to different, and possibly changing environments.
Bibliography


