7-1-2014

Reinforcement Learning and Planning for Preference Balancing Tasks

Aleksandra Faust

Aleksandra Faust

Candidate

Computer Science

Department

This dissertation is approved, and it is acceptable in quality and form for publication:

Approved by the Dissertation Committee:

Dr. Lydia Tapia, *Chairperson*

Dr. Trilce Estrada

Dr. Rafael Fierro

Dr. Melanie Moses

Dr. Lance Williams

Reinforcement Learning and Planning for Preference Balancing Tasks

by

Aleksandra Faust

B.S., University of Belgrade, Serbia, 1997
M.C.S., University of Illinois, Urbana-Champaign, 2004

DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

July, 2014

Dedication

To Nina, Adrian, and Eric

"He who moves not forward, goes backward." - Goethe

Acknowledgments

I am indebted to many for their help, support, and encouragement along this journey. Above all, I want to thank my advisor, Lydia Tapia. I am grateful for all the insightful advice, guidance, and championship. I have learned a lot from you, and you enabled me to do more than I thought I would be able to. I am also thankful for the members of my thesis committee, Rafael Fierro, Trilce Estrada, Melanie Moses, and Lance Williams, for their feedback, questions, and comments. In particular, thank you, Rafael Fierro, for advising me on control theory. And thank you Patricio Cruz and Rafael Figurea for Figure 5.13. Ivana Palunko and Domagoj Tolic, thank you for introducing me to the UAVs and control theory and for all the hours of very fruitful discussions. I am thankful for Peter Ruymgaart's valuable discussions and ideas, and for David Ackley's robust computing idea and consequent discussions. I am grateful to Marco Morales, Peter Stone, and Shawna Thomas for their very helpful feedback. I would like to thank Patricio Cruz for assisting with experiments.

Also, I am indebted to my lab colleagues, Nick Malone, Kaz Manavi, Torin Adamson, and John Baxter for listening through all the dry-runs and asking probing questions. Additionally I would like to thank Nick Malone for all the productive discussions, and for sharing one of the coldest rooms in the universe; and Torin Adamson for creating virtual environment models.

My family deserves my deepest gratitude. Eric, thank you for the unconditional support through all the long hours. And Nina and Adrian, thank you for all the patience. Mom, Dad, and Srdjan, thank for you being my cheerleading team. To my friends, Lisa Wilkening, Richard Griffith, Tamar Ginossar, and Alena Satpathi, thank you for encouraging me to embark on this journey, and for sharing ups and downs along the way. Last but not least, I am deeply indebted to my colleagues at Sandia National Labs: Bardon Rohrer, Kurt Larson, John Feddema, Marcus Chang, Lorraine Bacca, Joselyne Gallegos, David Gallegos, Steven Gianoulakis, Phil Lewis and Bernadette Montano. You made this journey possible.

Reinforcement Learning and Planning for Preference Balancing Tasks

by

Aleksandra Faust

B.S., University of Belgrade, Serbia, 1997
M.C.S., University of Illinois, Urbana-Champaign, 2004

Ph.D., Computer Science, University of New Mexico, 2014

Abstract

Robots are often highly non-linear dynamical systems with many degrees of freedom, making solving motion problems computationally challenging. One solution has been reinforcement learning (RL), which learns through experimentation to automatically perform the near-optimal motions that complete a task. However, high-dimensional problems and task formulation often prove challenging for RL. We address these problems with PrEference Appraisal Reinforcement Learning (PEARL), which solves Preference Balancing Tasks (PBTs). PBTs define a problem as a set of preferences that the system must balance to achieve a goal. The method is appropriate for acceleration-controlled systems with continuous state-space and either discrete or continuous action spaces with unknown system dynamics. We show that PEARL learns a sub-optimal policy on a subset of states and actions, and transfers the policy to the expanded domain to produce a more refined plan on a class of robotic problems. We establish convergence to task goal conditions, and even when pre-conditions are not verifiable, show that this is a valuable method to use before other more expensive approaches. Evaluation is done on several robotic problems, such as Aerial Cargo Delivery, Multi-Agent Pursuit, Rendezvous, and Inverted Flying Pendulum both in simulation and ex-

perimentally. Additionally, PEARL is leveraged outside of robotics as an array sorting agent. The results demonstrate high accuracy and fast learning times on a large set of practical applications.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Robots improve human lives by performing tasks that humans are unable or unwilling to perform. Search and rescue missions, space exploration, and assistive technology are some examples. In all these cases, robots must act autonomously in unfamiliar and often changing environments, and must be capable of efficient decision making. For successful, autonomous goal-oriented motion, we must ensure safety, communicate what we wish the robot to do (formulate a task), and act in real-time. Yet, robots are dynamical systems that are often highly non-linear with many Degrees of Freedoms (DOFs), making solving motion problems computationally challenging, especially when they need to act in real-time. Three challenges primarly contribute to the difficulty of autonomous motion: high number of DOF s, robot dynamics, and environment complexity.

There are many high-dimensional robotic applications including multi-robot coordination and control of complex kinematic linkages. These complex robotic problems frequently require planning high-dimensional motions that complete the task in a timely manner. Motion Planning (MP) identifies a sequence of actions that moves the robot in accordance to its dynamics (physical constraints)

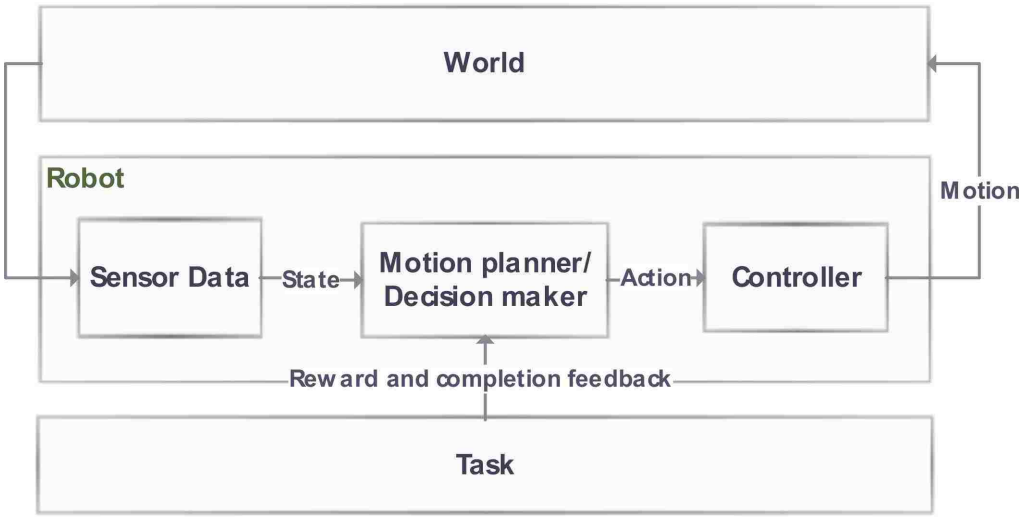


Figure 1.1: Decision making cycle.

and task objectives. Since manually accounting for all possibilities is often infeasible, sampling-based, learning-based, and other intelligent methods are the norm [69].

Robots are mechanical systems, and their movement can be addressed through the control of dynamical systems that describe them. Designing input to perform a task typically requires system dynamics knowledge. Classical optimal control approaches use a combination of open-loop and closed loop controllers to generate and track trajectories [72] requiring knowledge of the system dynamics [59]. On the other hand, the dynamics are frequently either unknown or intractable to solve.

Third, the physical environments where robots work must be considered when solving MP tasks. The resulting path or trajectory must be feasible and collision-free. For example, traveling through water might or might not be allowed depending on the task and the robot. Because the physical environments are continuous spaces and robots are high-dimensional, calculating allowed transitions

is a computational challenge. Sampling-based methods, such as probabilistic roadmaps (PRMs) [56] and Rapidly Exploring Random Trees (RRT) [70, 63], have shown great promise in overcoming computational feasibility challenges.

1.1 Research Objective

This research solves robot MP problems for a specific class of tasks: PBTs with safety, decision-making efficiency, and reuse in mind. The proposed solution overcomes the high-dimensionality, unknown dynamics, and physical environment challenges described above. The research develops conditions and methods for transferring the learned knowledge when the environment, task, or the robot change.

We consider robots as mechanical systems with unknown, non-linear control-affine dynamics controlled with acceleration. A robot's decision-making cycle is a closed feedback loop (see Figure 1.1). Robots receive information about their environment (or world), make the decision with the goal in mind, and actuate their system to perform the motion in order to complete the task. Determination of the external signal on the system, or action, is the responsibility of a motion planning agent (see Figure 1.1). The agent examines the current state and makes the decision that performs best with respect to some task or goal. The high dimensionality of the search space presents challenges for finding a suitable trajectory. The main focus of this research is the decision maker in Figure 1.1, and how it needs to act in order to perform a task.

One of the primary difficulties is how to describe the task. Some approaches for task formulation have been to learn from human demonstration, or experimentation and human feedback [1, 13, 14]. However, certain important tasks cannot

be readily demonstrated because they are either not attainable by humans or they are too costly or risky to demonstrate. Sometimes the task goals and constraints that the robot must obey are unknown or difficult to calculate. For example, consider a simple manipulation task. A robot is required to set a glass on a table without breaking it. We do not know precisely the amount of force that causes the glass to shatter, yet we can describe our preferences: low force and fast task completion. In an another example, it is difficult, even for expert pilots, to demonstrate a UAV flight that does not disturb the UAV's freely suspended load [39]. We consider these types of problems as PBTs. These tasks have one goal state and a set of opposing preferences on the system. Balancing the speed and the quality of the task are often seen as two opposing preferential constraints. For example, the time-sensitive Cargo Delivery Task must deliver the suspended load to origin as soon as possible with minimal load displacement along the trajectory (Figure 4.1b). The Rendezvous Task (Figure 5.7a) requires the cargo-bearing UAV and a ground robot to meet without a predetermined meeting point. In these tasks, the robot must manipulate and interact with its environment while completing the task in timely manner.

Reinforcement learning (RL) solves control of unknown or intractable dynamics by learning from experience and observations. The outcome of the RL is a control policy. RL has been successful for robotic task learning [61] in several problems such as table tennis [92], swing-free UAV delivery [39, 40], and a self-driving car [51]. However, traditional RL methods do not handle continuous and high-dimensional state spaces well [47]. We rely on RL to learn control policy for PBTs without knowing the robot's dynamics.

The objective of this research is to develop, analyze, and evaluate a solution for solving efficiently PBTs on systems with unknown dynamics. The solution will formally define PBT, develop a software framework, and characterize its success

conditions. The evaluation objective of this research to demonstrate the method's applicability on several non-trivial problems both in simulation and experimentally.

1.2 Contributions

This research solves motion-based high-dimensional PBTs using RL in continuous domains, and develops sufficient conditions under which the method is guaranteed to be successful. We also demonstrate that the method works in some cases where the conditions are relaxed. The body of the research is based on the following publications:

- Aleksandra Faust, Peter Ruymgaart, Molly Salman, Rafael Fierro, Lydia Tapia, "Continuous Action Reinforcement Learning for Control-Affine Systems with Unknown Dynamics", Acta Automatica Sinica, in print.

- Aleksandra Faust, Ivana Palunko, Patricio Cruz, Rafael Fierro, Lydia Tapia, "Aerial Suspended Cargo Delivery through Reinforcement Learning", under submission, Technical Report TR13-001, Department of Computer Science, University of New Mexico, August 2013.

- Rafael Figueroa, Aleksandra Faust, Patricio Cruz, Lydia Tapia, Rafael Fierro, "Reinforcement Learning for Balancing a Flying Inverted Pendulum," In Proc. The 11th World Congress on Intelligent Control and Automation, July 2014.

- Aleksandra Faust, Ivana Palunko, Patricio Cruz, Rafael Fierro, Lydia Tapia, "Learning Swing-free Trajectories for UAVs with a Suspended Load,"

IEEE International Conference on Robotics and Automation (ICRA), pages 48874894, Karlsruhe, Germany, May 2013.

- Aleksandra Faust and Nicolas Malone and Lydia Tapia, "Planning Constraint-balancing Motions with Stochastic Disturbances," under submission.

- Aleksandra Faust, Ivana Palunko, Patricio Cruz, Rafael Fierro, Lydia Tapia, "Learning Swing-free Trajectories for UAVs with a Suspended Load in Obstacle-free Environments," Autonomous Learning workshop at IEEE International Conference on Robotics and Automation (ICRA), Karlsruhe, Germany, May 2013.

Combined together, this research gives a problem definition, PEARL, and convergence conditions. We present PEARL for solving high-dimensional PBT (Figure 3.1) on a class of problems. PEARL trains the planning agent on small problems, and transfers the learned policy to be used for planning on high-dimensional problems. The key to PEARL is the feature selection method that constructs task-preference features invariant to the robot's state space dimensionality. Because the method learns and performs the task in the feature space, such transfer is possible.

To that end, this dissertation presents the following specific contributions:

- Feature extractions presented in Chapter 3

- A solution to an aerial transportation PBT (Chapter 4) in environments with static obstacles

- Conditions for learning in small, low-dimensional, spaces and transferring the policy to larger, higher dimensional spaces (Chapter 4)

- Efficient policy approximation for continuous actions (Chapter 4)
- Theoretical analysis and proofs of task goal's asymptotic stability in (Chapter 5)
- Continuous Action Fitted Value Iteration (CAFVI), a RL algorithm, for Markov decision processs (MDPs) with continuous states and actions (Chapter 5)
- Stochastic Continuous Action Fitted Value Iteration (SCAFVI) a stochastic version of CAFVI (Chapter 6)
- Applications of PEARL as local planner for solving PBTs for MDPs with continuous states and actions (Chapter 7)
- Application of PEARL for algorithm execution (Chapter 8)

Evaluation of PEARL and its components results in solving several tasks detailed below

- Coordinated Multi-Agent Pursuit Task (Chapter 3)
- Cargo Delivery Task (Chapter 4)
- Two heterogeneous agent Rendezvous Task (Chapter 5)
- Flying Inverted Pendulum in (Chapter 5).
- Stochastic version of Cargo Delivery Task, Flying Inverted Pendulum, and Rendezvous Tasks (Chapter 6)
- Coffee Delivery Task (Chapter 4)
- Package Delivery Task in Cities (Chapter 7)
- Array Sorting Task (Chapter 8)

Chapter 2

Related Work

PEARL formulates a MP problem as a decision-making problem. It solves it with RL and uses control theory tools for the analysis. This chapter gives necessary background from four diverse fields, and presents the work related to PEARL and its applications. Primers on MP, decision-making, RL and control theory tools are in Sections 2.1, 2.2, 2.3, and 2.4, respectively. We complete the chapter by discussing the application relevant to this research in Section 2.5.

2.1 Motion Planing (MP)

The basic MP problem is finding a valid path between known start and end goal states of some object. MP typically works in configuration space, *C-space*, a set of all the robot's poses in its physical environment (configurations) [69]. The subset in *C-space* that represents the robot's valid configurations is referred to as *C-free* [69]. The size of the *C-space* grows exponentially with the robot's DOFs. Even in the simplest case of environments with static obstacles, the *C-space* is large, and computational feasibility is in the forefront of issues that need to be addressed in

MP algorithm design [68].

To cope with the *C-space* complexities, a number of specialized methods are used for solving MP problems [31]. The approaches are commonly highly sensitive to the topology of the *C-space*, with some approaches working well in relatively free environments, and others in narrow passages or cluttered environments. A less computationally expensive algorithm can afford to search a large area in mostly *C-free* space. On the other hand, a more computationally intensive algorithm can search in more obstacle-laden environments, but is practical only in small areas. In particular, various *C-space* approximation methods, e.g., random sampling and others, have been successful in addressing the *C-space* complexity [31]. Sampling-based methods, as the name suggests, perform the MP by approximating the *C-space* with a countable (and often finite) sample set [31].

Kavraki et al. introduced PRM as one of the first sampling-based algorithms that works well in high-dimensional spaces [56]. The method consists of two phases: learning and querying. The learning phase constructs a roadmap, which is a graph lying fully in *C-free*. The nodes are *C-free* configurations, and the edges are collision-free paths between nodes. To construct a roadmap, *C-space* is sampled. After retaining only *C-free* samples and adding them as nodes in the roadmap, a local planner of choice connects each sample with its k nearest neighbors. The local planner creates a sequence of configurations between two nodes. If all configurations are in *C-free*, the edge is added to the roadmap. Once the roadmap is constructed, it can be queried. A query consists of start and goal points in *C-free*, and a solution to a query is a valid path in *C-free*. To solve a query, PRMs first add the start and goal states to the roadmap using the connection methods described above. Then, a graph search finds the shortest path from start to goal in the roadmap, with respect to some metric, often Euclidean distance. Alternatively, other problem-specific metrics could be used: clearance,

the least number of roadmap edges, etc. Since the original publishing, PRMs are an algorithm of choice for path planning in high-dimensional static environments and have been extended for biased sampling [10, 134] for complex robotic problems, including moving obstacles [52, 109], noisily modeled environments (e.g., with sensor) [76], and localization errors [4, 9]. They are used in a wide variety of applications such as robotic arm motion planning [77, 78], molecular folding [6, 103, 123], and a six-legged lunar rover [48]. Also, PRMs have been previously integrated with RL e.g., a manipulator moving in a dynamic space [103] and for stochastic reachability [80].

In the contrast to PRMs that can be reused over several queries, there are methods that incrementally search *C-space* [70, 63]. RRT chooses the next sample in the locality of the previous sample while conforming to nonholonomic and kinodynamic differential constraints. They have been adapted to work under uncertainty [25]. The ability to handle the system constraints made RRTs popular method for collision detection on physical robots in dynamic environments: arm manipulators [95], UAVs [62], and multi-agent systems [34] among others.

The classical MP problem assumes that the agent knows the goal state prior to the start of planning [69, 31, 68]. In modern applications, this assumption is too strong. For example, an agent might know to recognize the goal state without knowing ahead of time the location of the goal state. In other problems the start and goal states change, and we need to learn from experience and adapt. The dynamics of the system cannot be overlooked either, and in some problems we need to find a feasible trajectory rather than just a path. While both trajectories and paths are sequences of robot's positions, a trajectory is a timed sequence, while a path contains no time information [32].

2.2 Decision-making

A robot interacts with an environment in a closed loop (Figure 1.1): decision-making loop. At each time step, the agent observes the environment and the robot's internal conditions, and gathers that information into what we call a *state*. The decision-making module examines the state and makes a decision about what to do next, e.g., what *action* to perform. The action changes the system and its environment. That change is reflected in the next state that the agent observes. The time between subsequent observations is the time step. Although in general the time step can vary, for the problems considered it will fixed. The environment is external to the robot. The robot contains a decision-making component that we refer to as a decision-making agent, or simply as an *agent*. States of robot's other modules are external to the decision-making agent, and in general the agent has no knowledge of them unless communicated through the state. Nevertheless, we will use the term robot to refer to decision-making agent when it is clear from the context that are not concerned with the entire robotic system, e.g., the robot makes a decision.

The agent observes a reward in addition to observing a state. The reward is a numerical input that communicates to the agent the quality of the immediate state. Note that a reward in some literature is defined as a mapping between state-action pairs rather than only the state [120]. The reward allows us to a formulate a goal for the agent - the agent's goal is to maximize the *return*, the cumulative reward over its lifetime [120]. The action causes a change in the state. The result of an action is another state. We can consider a mapping between pairs of states, and actions that assign a probability of arriving at one state, when the action is applied on the other state. This mapping is called a probability transition function. A decision-making problem can be formalized with a Markov Decision Process (MDP), which we discuss next.

2.2.1 Markov Decision Process (MDP)

Given states, actions, a probability transition function, and a reward, a MDP formalizes a decision-making problem.

Definition 2.2.1. *A first order* Markov Decision Process (MDP) $\mathcal{M}$ *is a tuple* $(S, A, \mathbf{D}, R)$ *where:*

1. *S is a set of states* $\mathbf{s}$,
2. *A is a set of actions* $\mathbf{a}$,
3. *probability transition function* $\mathbf{D} : S \times A \times S \to \mathbb{R}$, *where* $\mathbf{D}(\mathbf{s}_1, \mathbf{a}, \mathbf{s}_2)$ *is a probability of resulting in state* $\mathbf{s}_2$ *when action* $\mathbf{a}$ *is applied to state* $\mathbf{s}_1$,

$$\mathbf{D}(\mathbf{s}_1, \mathbf{a}, \mathbf{s}_2) = Pr\left(\mathbf{s}(t+1) = \mathbf{s}_2 | \mathbf{s}(t) = \mathbf{s}_1, \mathbf{a}(t) = \mathbf{a}\right),$$

   $\mathbf{s}(t)$ *and* $\mathbf{a}(t)$ *are state and action at time* t,

4. *reward signal* $R : S \to \mathbb{R}$ *is an immediate reward associated with state* $\mathbf{s}$, $R(\mathbf{s}) = R_{\mathbf{s}}$, *and*
5. *states S satisfy the Markov property, i.e., the effect of the state depends on only of the current state,* $Pr(\mathbf{s}_{t+1} | \mathbf{s}(t), \mathbf{a}(t)) = Pr(\mathbf{s}(t+1) | \mathbf{s}(t), \mathbf{a}(t), \mathbf{s}(t-1), \ldots, \mathbf{s}(0))$.

Definition 2.2.1 describes a MDP as a stochastic automata with utilities [122]. The Markov property ensures that states capture all information needed for state transitions. It requires MDP to be memoryless, i.e., it does not matter how a particular state was reached to predict future states. From the generalized definition, several variants can be distinguished based on the time step, state and action set sizes, and the type of probability transition function:

- When the state transitions occur at regular intervals $c \in \mathbb{R}$, $c > 0$, i.e., the time step $\Delta t = c$ is fixed, the associated MDP is the *discrete-time MDP*. When $\Delta t \to 0$ MDP is *continuous-time* [122].
- When the state space S is a finite, countable, or continuous set, the MDP is *finite-state, countable-state, or continuous-state MDP*, respectively. We use terms discrete and finite interchangeably [122].
- Similarly, MDPs with actions A that are finite, countable, or continuous sets, are called *finite-action, countable-action, or continuous-action MDPs*, respectively [122].
- When probability transition function is a deterministic process, i.e., an action's effect is uniquely determined by the current state, we refer to the MDP as *deterministic process MDP*. The probability transition function, in this case, can be simplified to a function that maps state-action pairs to their state outcomes, $\mathbf{D} : S \times A \to S$, and we call it a transition function, or simply system dynamics [122].

This research is concerned with discrete-time, deterministic MDPs, and the following introduction is restricted to these types of MDPs unless stated otherwise.

An objective of a decision-making problem defined with a MDP is to move the system through the states in such way as to maximize the accumulated reward starting at an arbitrary state [122]. The objective can be formalized through definition of a state-value function, a mapping between states and returns. The state-value function of a state is a measure of the state's quality, and quantifies the expected return attainable from that state.

Definition 2.2.2. *Function*

$$V(\mathbf{s}(0)) = \sum_{t=0}^{\infty} \gamma^t R(\mathbf{s}(t)). \tag{2.1}$$

is called the state-value function of a discrete-time deterministic MDP $\mathcal{M}$ $(S, A, \mathbf{D}, R)$. $R(\mathbf{s}(t))$ *is an immediate reward observed at time* t *starting at state* $\mathbf{s}$, *and* $0 \leq \gamma \leq 1$ *a discount constant.*

Discount factor γ determines how much the immediate payoff is weighted against the payoff in the future [122]. The bigger the discount factor, the more "long-sighted" the agent is. A discount constant γ of less than one ensures that the state value remains finite. MDPs with undiscounted objectives ($\gamma = 1$) typically use a finite-horizon to maintain finite-valued objective, i.e., the value function is $V(\mathbf{s}) = \sum_{t=0}^{n} R(\mathbf{s}(t))$ for some $n \in \mathbb{N}$ [122].

Among all possible state-value functions, the optimal state-value function is the one that returns the highest value for all states [20]. We define it next.

Definition 2.2.3. *The optimal state-value function,* V^*, *is supremum state-value function, i.e.,*

$$V^*(\mathbf{s}) = \sup_V V(\mathbf{s}). \tag{2.2}$$

Now, we can define a policy as a solution of an MDP problem, and show the connection between policies and state-value functions. Then, we introduce the optimal solution to a discrete-time deterministic MDP.

Definition 2.2.4. *A* solution to a discrete-time, deterministic MDP $\mathcal{M}$ $(S, A, \mathbf{D}, R)$ *is a policy* $\pi : S \to A$, *which assigns an action* $\mathbf{a}$ *to perform in state* $\mathbf{s}$, *i.e.,* $\pi(\mathbf{s}, \mathbf{a}) = \mathbf{s}'$. *A state-value function defined by the policy* π *is* $V^\pi(\mathbf{s}) = \sum_{t=0}^{\infty} \gamma^t \pi^t(\mathbf{s})$.

Similarly to the optimal state-value function, the optimal solution to a discrete-time deterministic MDP is a policy that maximizes the objective, state-value function. Thus the following definition formalizes the optimal solution.

Definition 2.2.5. *A solution to a discrete-time MDP* $\mathcal{M}$ $(S, A, \mathbf{D}, R)$ *is* optimal*, iff the policy* $\pi^* : S \to A$ *is optimal, i.e., the policy maximizes the discounted cumulative state reward* $V^{\pi^*}(\mathbf{s}) = V^*$*. We denote the optimal policy,* π^**.*

Last, we introduce a greedy policy w.r.t. to a state-value function V.

Definition 2.2.6. *A policy* $\pi : S \to A$ *is* greedy *w.r.t. to a state-value function* V*, iff* $\pi^V(\mathbf{s}) = \text{argmax}_{\mathbf{a} \in A} V(\mathbf{s}')$*, where* $\mathbf{s}' = \mathbf{D}(\mathbf{s}, \mathbf{a})$*.*

The greedy policy w.r.t. to optimal state-value function, V^*, is an optimal policy, $\pi^{V^*} = \pi^*$. The proof can be found in [122]. This is important because it allows us to find an optimal policy by calculating the state-value function.

The Bellman equation [20] shows that the state state-value function V^* can be recursively represented as

$$V^*(\mathbf{s}) = R(\mathbf{s}) + \gamma \max_{\mathbf{a} \in A} V^*(\mathbf{D}(\mathbf{s}, \mathbf{a})). \tag{2.3}$$

The state-value function is a fixed point for an operator that adds an immediate state reward to the discounted value of the state that results from the system's transition following greedy policy. The Bellman Equation (2.3) has a unique solution when the γ is less than one [122], and it give us practical means to iteratively calculate the state-value function in order to obtain the MDPsolution [20]. These methods are called dynamic programming.

Indeed, dynamic programming, and in particular, *value iteration* methods iteratively find the state-value function. For example, V-iteration (Algorithm 2.1), a Value Iteration, solves finite-state, finite-action MDPs with a completely known probability transition function [22]. V-iteration starts with an arbitrary state-value function and iteratively calculates state-values traversing all states in each iteration. V-iteration is simple to implement, but is not very efficient. Its running time of $O(n|S||A|)$ and memory requirement of $O(2|S| + |A|)$ make it computationally

impractical to use in large state and action spaces, and infeasible in continuous MDPs. Further, it requires knowledge of the probability transition function **D**.

Algorithm 2.1 Value iteration. Adapted from [22].

Input: MDP $\mathcal{M}$ $(S, A, \mathbf{D}, R)$, discount factor γ, # of iterations n

Output: state-value function V

1: **for** $i = 1, \ldots, n$ **do**
2: **for** $\mathbf{s} \in S$ **do**
3: $V_n(\mathbf{s}) = R(\mathbf{s}) + \gamma \max_{\mathbf{a} \in A} V_{n-1}(\mathbf{D}(\mathbf{s}, \mathbf{a}))$
4: **end for**
5: **end for**
6: return V_n

2.3 Reinforcement Learning (RL)

Reinforcement learning (RL), a field of machine learning, is a family of algorithms that solve MDPs [120]. MDPs can be solved in several ways including dynamic programming and RL. The two methods are similar, but have some major differences. The first difference is that dynamic programming assumes that the rewards and state transitions are known ahead of time, while RL assumes they are not known. RL finds the policy by gradually exploring the state space, and learns the MDP using a technique called bootstrapping that relies on available samples. Bootstrapping means that the state values are estimated using estimates of other states [120]. The second major difference between dynamic programming and RL is that dynamic programming is constrained to a finite state space, which is not the case for all RL algorithms (Section 2.3.5).

The goal of the RL agent is to find an optimal policy $\pi : S \rightarrow A$ that maximizes the accumulated payoff. A policy determines an action that the agent should

take from a state. In addition to state-value function, V, there is another function relevant to the RL algorithm: the action-value function, or Q-function. Recall that state-value function $V : S \rightarrow \mathbb{R}$ is an estimate of the expected accumulated payoff that could be accomplished from the current state. Similarly Q-function, $Q : SxA \rightarrow \mathbb{R}$ represents the expected maximum accumulated payoff associated with transition from state **s**, using action **a**. Both of these functions are referred to as value functions. Either one of the them induces a greedy policy, where the action chosen maximizes the value function. Equation (2.4) shows the relationship between value functions V and Q.

$$Q^{\pi}(\mathbf{s}, \mathbf{a}) = V^{\pi}(\mathbf{D}(\mathbf{s}, \mathbf{a})) \tag{2.4}$$

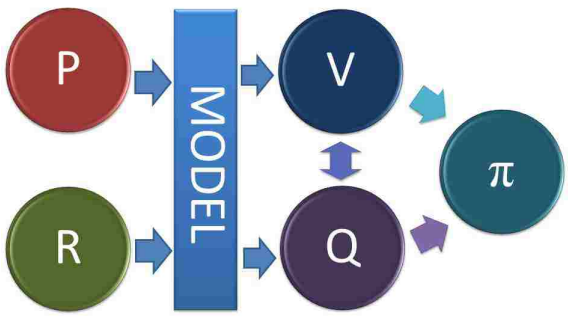
The action-value function, Q , is easier to calculate when the transition function is not known, which is why it is often used in RL instead of the state-value function [27]. For example, when the agent is constructing the value function by interacting with the system, the Q-function stores the cumulative reward associated with the transition, without explicitly learning the transition model as well. Definition 2.3.1 shows that the greedy policy w.r.t. Q does not depend on the probability transition function.

Definition 2.3.1. *A policy $\pi^{Q} : S \rightarrow A$ is greedy w.r.t. to an action-value function Q, iff $\pi^{Q}(\mathbf{s}) = \text{argmax}_{\mathbf{a} \in A} Q(\mathbf{s}, \mathbf{a})$.*

Similarly to the state-value function, we can define an optimal action-value function as the supremum of all Q functions, and it can be shown that greedy policy associated with an optimal action-value function is an optimal solution to the given MDP, i.e., $\pi^{Q^*} = \pi^*$ [130].

The three functions Q, V, and π are closely related. Determining one value function induces the other two (Figure 2.1). Thus, an RL algorithm needs only to

Figure 2.1: Relationship between transitional probabilities, rewards, models, value functions and policies. Transitional probabilities (P) and a reward signal (R) determine the system model. The model determines state value function (V), and state action function (Q). Value functions V and Q induce each other, and each one of them induces a policy (π).



learn one of the three functions, and the other two can be derived [27]. Indeed, different RL algorithms focus on finding one of these three functions.

2.3.1 Exploration vs. Exploitation

Another aspect of RL is the consideration of whether an agent is using and updating the same policy at the time of learning, or if the learning and exploitation phases are separated. Methods that continuously update the current policy are *on-line reinforcement learning*, while the latter cases are *off-line or batch reinforcement learning*[1] [67]. On-line RL interacts with an unknown system while learning the policy and must balance two opposing goals, exploration and exploitation [120]. Exploration is needed to learn, while exploitation uses the already learned knowledge to control the agent's behavior. Because the two are opposing goals, they commonly meet in ϵ-greedy policy, which follows the current learned policy with

[1] Another set of terms commonly used in the literature is *on-policy* and *off-policy*, respectively.

probability $1 - \epsilon$, and performs a random (exploratory) action with ϵ probability, for some $\epsilon > 0$. The problem with ϵ-greedy policy and on-line RL in general is that there are no safety guarantees to the system. This is because the random actions can put the system into an unsafe position, e.g., collision with an obstacle [67]. On the other hand, batch RL has two phases, learning and planning [67]. During learning, which is typically done in simulation, the agent follows another policy that can be 100% exploration [67]. Once the policy is learned, the agent switches to the planning mode, exploiting the learned policy with no exploration. The advantage is that batch RL is safer because it does not perform random actions, but it cannot adapt to the changes in its environment [67, 27].

2.3.2 Task Representation

RL must consider the nature of the task. If the task itself has a defined end state, we call such a task an *episodic* task [120]. The corresponding RL algorithm accomplishes the task in a finite number of steps. This fixed sequence of steps is called an episode. The consequence of having the guarantee of a finite number of steps is the assurance that the value function for any state is finite [120]. In contrast, tasks without a defined end, which continue to run potentially indefinitely are called *non-episodic* [120]. There is a possibility of unbounded value function growth for the non-epsiodic tasks, jeopardizing the algorithm's convergence. For that reason, the accumulated reward is discounted with $\gamma \in (0, 1)$, which guarantees that the value functions remain finite even for non-episodic tasks. However, both episodic and non-episodic tasks can be unified under one architecture, by adding a special absorbing state where the agent gets trapped at the end of an episode [120], which is the view we adopt in this research.

2.3.3 Value Iteration RL

RL algorithms that work in finite domains share the same framework represented in Algorithm 2.2 [27] for an infinite horizon, value and policy iteration algorithm. After initialization, the agent is continuously executes actions, observes the new state, and updates its value function. This generic framework can be easily adapted for episodic tasks, as well as for off-line learning.

Algorithm 2.2 General Value Iteration Learning Framework adapted from [27]

Input: Policy π

1: $InitializeQ(s, a) \forall s, a$
2: Observe current state s
3: **while** not end of the episode **do**
4: select an action according to the policy π and execute it
5: receive the reward r
6: Observe current state s'
7: Value update
8: $s \leftarrow s'$
9: **end while**
10: $pi'(s) = argmax_a V(s)$

Output: Policy π'

Seeking to bring tools of supervised machine learning to system prediction problems, Sutton introduces Temporal Difference (TD) learning [121]. Sutton's goal is not only to be able to perform goal-oriented searches, but rather to learn how the system works, and use the learned knowledge to predict the next state. The key idea is that actual system observations are backtracked to the previously visited states. The backtracks are performed as the difference between the time steps, thus the name: temporal difference. When applied to only the previous

step, the method is called TD(0). TD(0) follows the policy evaluation framework in Algorithm 2.2, and the value update is done with

$$V(\mathbf{s}) = (1 - \alpha)V(\mathbf{s}) + \alpha(R + \gamma V(\mathbf{s}')). \tag{2.5}$$

The update assigns a new estimate to the state-value function in state $\mathbf{s}$ as a weighted average of the existing estimate and the new estimate: the sum of the observed reward and the current estimate of the state-value in $\mathbf{s}'$. The policy is an input to the algorithm, and does not change for the duration of the episode. Learning the constant, α determines how influential is the new information. Larger values of α put more emphasis on the new information. TD(0) converges to the optimal policy for sufficiently small α [120]. To backtrack several steps, the system needs to keep track of the visited states. Then the updates are done to all eligible states, such that they decay by factor λ, as the agent moves further away from the state. This type of memory and updates are called eligibility traces. They are applicable to any method that is based on the idea of the temporal difference.

Watkins made a key observation that two policies can be efficiently evaluated and thus improved upon, using a single-step evaluation by measuring the effect of an action on a policy [129]. The expected return in the current state under a policy π is compared to a policy that differs in a single step: instead of following π another action is chosen. Thus, Watkins introduced the action-value function that measures the "goodness" of state-action pairs. Learning value functions for states and associating them with a greedy policy allows the system to learn to maximize the payoff without having to systematically explore the entire state space, like dynamic programming does. This method is called Q-learning, which updates the learns the policy by updating the Q-function using

$$Q(\mathbf{s}, \mathbf{a}) = (1 - \alpha)Q(\mathbf{s}, \mathbf{a}) + \alpha(R + \gamma \max_{\mathbf{a}' \in A} Q(\mathbf{s}', \mathbf{a}')). \tag{2.6}$$

The updated value of the state-action pair is the weighted average of the current best knowledge and the new estimate. In effect, Watkins' Q-learning is similar

to Sutton's TD methods [121], although the approaches come from different perspectives. The primary difference is that the estimates in TD learning are based on an action, rather then a state. This seemingly subtle difference implies that TD(0) uses ϵ-greedy policy to estimate $V(\mathbf{s})$, while Q-learning is an off-policy method because it is free to use any policy to choose a transition out of state $\mathbf{s}$. Q-learning converges to an optimal Q, as long each state action pair is visited infinitely, and when the learning rate is sufficiently small [130].

Sarsa, introduced by Rummery and Niranjan [112], is an on-policy variant of Q learning. The update step of Sarsa is:

$$Q(\mathbf{s}, \mathbf{a}) = (1 - \alpha)Q(\mathbf{s}, \mathbf{a}) + \alpha(R + \gamma Q(\mathbf{s}', \mathbf{a}')) \tag{2.7}$$

where $\mathbf{a}' = \pi(\mathbf{s}')$. Instead of using $\max_{\mathbf{a}'} Q(\mathbf{s}', \mathbf{a}')$ as an estimate of the value of the next state like Q-learning does, Sarsa estimates the value of state $\mathbf{s}'$ assuming the current policy will be followed, $\mathbf{a}' = \pi(\mathbf{s}')$. It uses an observed ($\mathbf{s}$,$\mathbf{a}$,$R$,$\mathbf{s}'$) tuple in conjunction with $\mathbf{a}'$, hence the name Sarsa. Note that selecting $\mathbf{a}'$ is potentially less computationally expensive than searching for the maximum. To converge to the optimal policy, Sarsa needs to use ϵ-greedy policy in order to continue exploring the environment. It tends to converge faster than Q-learning because, intuitively, its search through the state space is more focused [120].

2.3.4 Policy Search Methods

Another RL approach is policy search, where a policy is represented as a parametrized vector and the policy is directly searched, in contrast to the value iteration methods described in Section 2.3.3 that iteratively improve on value functions [27]. Typically, the policy search is performed as a gradient descent on the parameters. A popular method that falls in this category is the actor-critic method originally introduced by Barto et al. [18]. Inspired by neurons Barto et al. de-

signed a feedback method for neural networks to learn control on the inverted pendulum problem. This work was first in what later became a series of similar approaches all sharing the basic framework, but differing in choice of approximators [27]. The actor-critic method is a gradient-based policy search method, which means that the policy is differentiable by the parameter. The method works in two phases. First, the value function is computed for the current policy. Then the value function is used to make a gradient descent in the direction of the higher return, which obtains the new policy. The policy is the actor, and the value function is the critic [18, 27]. Recently, Peters and Schaal improved on the actor-critic method using natural gradient descent rather than standard gradient descent [105]. Natural policy gradient is the steepest descent w.r.t. *Fisher Information*. Fisher Information is a measure of the information quantity that a random variable contains, and thus its gradient provides more appropriate direction of policy improvement. The Natural Actor-Critic method converges faster and with fewer samples than traditional actor critic methods. The authors showed that, on the cart balancing problem, the Natural Actor-Critic method achieves optimum learning eight times faster than true actor-critic method.

2.3.5 RL in Continuous Domains

The methods described in Section 2.3.3 work for finite state spaces because they require tabular representation of the state action space. However, the tabular representation limits the practicality of the value iteration methods since the size of the value function table grows exponentially with the number of DOF. For large discrete or continuous MDPs, the methods in Section 2.3.3 needs to be modified. The learning paradigm remains the same, but the function we learn, whether it is a state-value, action-value, or policy, need to be approximated [120, 27]. RL with approximators is more relevant to the MP field because the majority of MP

problems work in high-dimensional state spaces. The state space grows exponentially with the *C-space* dimensionality (see Section 2.1), thus often making the discretized space too large. Typically, the RL learns the value (cost) function and derives a greedy control policy with respect to the value.

Unlike RL in finite spaces, where exhaustive search is possible and theoretical guarantees of convergence to the optimal policy exist, the convergence conditions are sparse and a current topic of research in the continuous case [47]. However, in practice the algorithms work well and can be much faster than their discrete counterparts. This is because the major drawback of discrete algorithms is that they do not take into account the similarity between the states, and every state needs to be examined [27]. The approximation functions allow generalization over the state space based on similarity (proximal similarity or feature based). In any event, the approximation methods sample the system until the approximation is sufficient and the error is bounded. Choosing approximators is a current research area [135]. Well-chosen approximator functions produce solutions to MDPs that perform well, while poorly chosen functions produce no results.

It is theoretically possible to come up with the optimal policy in finite state spaces [130]. However, few problems in MP have truly finite state spaces. Thus, the step of discretization of the state space introduces approximation to the algorithm, and the derived policy is unlikely to be optimal [27]. Another issue to consider when discretizing the state and action space is that the discretization of actions needs to be compatible with the discretization of the state space, and preserve the Markovian property of the system [27]. For example, if the state space is discretized in too large areas, an action might not result in a change of state. The value functions use tabular representation for discrete state spaces. Thus, unnecessarily fine discretization of the state space will lead to larger function representation and longer learning time [27].

Two types of approximations are common: parametric and nonparametric [27]. Parametric approximation uses some function $\hat{V} : S \times \mathbb{R} \rightarrow \mathbb{R}$, the target function. A special case is a linear approximation, where the target function is represented with a linear map of a feature vector $\hat{V}(\mathbf{s}, \boldsymbol{\theta}) = \boldsymbol{\theta}^T \boldsymbol{F}(\mathbf{s})$ [27]. Because a feature maps the entire robot's state subspace to a single point, RL in continuous spaces is sensitive to feature selection [27]. A popular choice of feature vectors are discretization, and basis functions [135, 27, 120, 122, 47]. Discretization partitions the domain, scaling exponentially with the space dimensionality. Basis functions, such as kernels and radial base function networks, offer more learning flexibility. But they can require manual parameter tuning, and the feature number can increase exponentially with the space dimensionality [135, 27]. Nonparametric approximation creates features from the available data. Neural networks is an example [27]. Although it is much more difficult to show the convergence properties for RL with function approximators, convergence for linear parametric approximators is easier to show than for nonparametric approximators [27].

In continuous action RL, the decision-making step, which selects an input through a policy, becomes a multivariate optimization [47]. The optimization poses a challenge in the RL setting because the objective function is not known. Robots need to perform input selection many times per second, 50-100 Hz is not unusual [61]. The decision-making challenges brought action selection in continuous spaces to the forefront of current RL research, with the main idea that the gradient descent methods find maximums for known, convex value functions [47] and in actor-critic RL [44] (Section 2.3.4). Gradient descent methods for policy approximation work well in some convex cases. Hover, they require an estimate of the gradient, can be stuck in the local minima, and can take a long time to converge which requires descent step adjustments along the way, especially for near-linear functions [47]. Some gradient-free approaches such as Gibbs sampling [60], Monte Carlo methods [71], and sample-averages [12] have been tried. On-line

optimistic sampling planners have been researched [26, 28, 81, 128]. Specifically, Hierarchical Optimistic Optimization applied to Trees (HOOT) [81], uses hierarchical discretization to progressively narrow the search on the most promising areas of the input space, thus ensuring arbitrarily small error.

Approximate Value Iteration (AVI) (Algorithm 2.3) is a continuous-state extension of the value-iteration [36, 27, 120]. It produces an approximate solution to a MDP with continuous state spaces and a finite action set. The value function is approximated with a linearly parametrized feature vector. It is in an Expectation-Maximization (EM) algorithm which relies on a sampling of the state space transitions, an estimation of the feature vector parameters, and a linear regression to find the parameters that minimize the least square error. Algorithm 2.3 presents AVI [36]. To learn the approximation of the state-value function, AVI learns the parametrization $\boldsymbol{\theta}$. Starting with an arbitrary vector $\boldsymbol{\theta}$, in each iteration, the state space is uniformly, randomly sampled to produce a set of new state samples M. A new estimate of the state value function is calculated according to $V(\mathbf{s}) = R(\mathbf{s}) + \gamma \max_{\mathbf{a} \in A} \boldsymbol{\theta}^T \boldsymbol{F} \circ \mathbf{D}(\mathbf{s}, \mathbf{a})$ for all samples $\mathbf{s} \in M$. $0 < \gamma < 1$ is a discount factor, and $\mathbf{D}(\mathbf{s}, \mathbf{a})$ is sampled transition when action $\mathbf{a}$ is applied to state $\mathbf{s}$. A linear regression then finds a new value of $\boldsymbol{\theta}$ that fits the calculated estimates $V(\mathbf{s})$ into a quadratic form $\boldsymbol{\theta}^T \boldsymbol{F}(\mathbf{s})$. The process repeats until a maximum number of iterations is performed.

AVI does not directly depend on the time step size. Similarly, the algorithm is agnostic to the time it takes to reach the goal state and to any ordering there might be in the state space chain. It randomly explores the state space and learns a function that assigns a scalar representing a quality of a state. Its running time is $O(M^2 \cdot \mathit{iterations} \cdot \|\mathit{actions}\|)$, where M is the number of samples, *iterations*, the number of iterations to perform; and *actions* is the size of the discretized action space. AVI assumes that a simulator is available to find a new estimate of the

Algorithm 2.3 Approximate Value Iteration (AVI) adopted from [36].

Input: MDP (S, A, D, R)

Input: feature vector F

Output: $\boldsymbol{\theta}$

$\boldsymbol{\theta} \leftarrow$ zero vector
while NOT(max *_iterations*) **do**
  M, R_M = randomly select n states from s and observe rewards
  for each $(\boldsymbol{s}, r) \in (M, R_M)$ **do**
    for each $\boldsymbol{a} \in A$ **do**
      $q(\boldsymbol{a}) = r + \gamma \boldsymbol{\theta}^T \boldsymbol{F} \circ \boldsymbol{D}(\boldsymbol{s}, \boldsymbol{a})$
    end for
    $V(\boldsymbol{s}) = \max_{\boldsymbol{a} \in A} q(\boldsymbol{a})$
  end for
  $\boldsymbol{\theta} \leftarrow \text{argmin}_{\boldsymbol{\theta}}(\|\boldsymbol{\theta}^T \boldsymbol{F} - V\|^2)$
end while
return $\boldsymbol{\theta}$

state value, based on a greedy policy. A related approach is on-line Fitted Value Iteration [27], where a TD-like update is used for estimates and a gradient descent is used to find the parameter value. Both on-line and off-line value iterations can be done with either state value functions or with action value functions. Both methods are referred to as approximate Q learning.

2.3.6 RL Paradigms

This section introduces several learning paradigms in which RL operates in order to improve the learning efficiency. We discuss model-based RL learing from

demonstration, and learning transfer.

When a RL algorithm learns the transitional probabilities first, thus creating an internal model of the process, it is called *model-based reinforcement learning* [50]. In contrast, the methods that learn one of the value functions or a policy, directly are referred to as model-free. In general, model-based algorithms are able to obtain better policies with less observation than the model-free algorithms, although it might take them more steps to achieve the same level of performance [50]. One example is BECCA [110, 111]. In BECCA, a feature creator performs autonomous discretization of the state space while a reinforcement learner learns the discretized model and acts on it. BECCA was used to train a robotic arm to perform a pointing task [77, 79, 78].

Value functions are driven by the reward function and transitional probabilities of the system. So, instead of focusing on learning a value function or a policy, an algorithm might attempt to first learn the reward structure. We call the approach *inverse reinforcement learning* [96]. Apprenticeship learning [1], or apprenticeship via inverse reinforcement learning (AIRL), solves MDPs where demonstrations of the tasks are available without an evident reward signal from the environment. The algorithms use observations of an expert behavior to teach the agent the optimal actions in certain states of the environment. AIRL is a special case of the general area of Learning from Demonstration (LfD) [13, 14], where the goal is to learn a complex task by observing a set of expert demonstrations. AIRP is the intersection of LfD and RL [1]. At a high level, the reward is extracted from the demonstrations and formed such that the agent is discouraged from deviating from the demonstration. Separately, a model is learned and refined. The model that maximizes the reward is then used to generate a trajectory. To achieve autonomous helicopter flight as an example of apprenticeship learning, Abbeel et al. developed a sophisticated learning methodology and environment [2].

To address learning generalization and to attain knowledge transfer between tasks, RL researchers have been developing learning transfer methods. Taylor and Stone [125] proposed value function transfer between tasks in different state and action spaces using a behavior transfer function to transfer the value function to the new domain. Sherstov and Stone in [125, 126] examine action transfer between the tasks, learning the optimal policy and transferring only the most relevant actions from the optimal policy. We take the opposite approach. To save computational time, we learn a sub-optimal policy on a subset of actions, and transfer it to the expanded action space to produce a more refined plan. McMahan et al. [84] suggested learning a partial policy for fixed start and goal states. Such partial policies manage state space complexity by focusing on states that are more likely to be encountered. Another approach examines action transfer between the tasks, learns the optimal policy, and transfers only the most relevant actions from the optimal policy [115]. Another RL approach solves manipulation tasks with hard [119] and soft [65] constraints by learning in the low-dimensional task space.

2.3.7 RL and Planning

Section 2.1 mentions that planning is concerned with finding a path from a start to a goal state. To formulate the planning method in RL terms, the agent must receive some sort of reward at every transition, or equivalently each transition must be associated with a cost. The reward structure is a way of communicating to the agent when the goal was achieved. Since the rewards come from the outside world, they are external and unknown to the agent.

C-space corresponds directly to the MDP state space, and the action space is directly transferable between the planning paradigm and the RL framework. The transitional probabilities are also the same between the problem formulations,

and not known to the agent. Typically, the planning problem corresponds to an episodic task because the planning problem has a defined goal state. An episode concludes when an agent reaches the goal. Some planning problems rely on a heuristic to guide the search towards the goal. Value functions play the role of the heuristics in RL. RL approaches go one step further and update and improve the heuristic based on the empirical evidence.

Planning methods are traditionally off-line methods [69], meaning that the path is found and calculated ahead of time, in an open loop manner without the feedback of success. Some off-line RL methods work in a very similar way. However, more modern planning approaches require the agent to plan on-line, adjusting the plan as the robot moves through the environment. On-line RL algorithms are very well suited for these approaches. Sutton and Barto introduced the relationship between planning and RL [120]. In their view planning is the process of taking a model as an input and producing a policy (Figure 2.1). Transitional probabilities and rewards are used to generate a model. The model produces simulated experiences which use backups based on the Bellman optimality Equation (2.3) to derive the value function and therefore the policy [20]. This model considers only off-line, model-based RL to conform to the planning framework. The authors further differentiate planning from learning by pointing out that the planning derives plans from the simulated experience only, while the learning can be done either from the simulated experience or performed on the real data produced by an actual system [120].

LaValle [69] considers a traditional view to be a three-phase framework presented in [120], in which learning, planning, and execution phases are separated. LaValle coins the term *reinforcement planning* for the methods that overlap planning and RL. He uses a more modern interpretation of planning, and introduces a simulation-based framework where the model is not necessary the input to the

planning. Both model-based and model-free RL are considered planning. The simulation-based framework of reinforcement planning is typically used with a Monte Carlo simulator of the system, but can be used with the actual system. This research views planning and RL in the light of the simulation-based framework, and takes a holistic view not differentiating between learning and planning.

Action selection plays a central role in RL-based planning [128] as well. Outside of RL, Johnson and Hauser [55] developed a MP approach based on reachability analysis for trajectory tracking in acceleration-bound problems. Zhang et al. [139] proposed a sampling-based planner that finds intermediate goals for trajectory planning for systems with dynamical constraints.

2.4 Optimal Control and Lyapunov Stability Theory

This section gives a very brief introduction into the key concepts of control and Lyapunov stability theories relevant to the presented research. The topic of control theory is the study of the influence on a system that changes over time [17]. A system that changes over time, a *dynamical system*, is represented with a set of variables, state space, and is influenced with another set of variables, *control input*. Control input, or just input, changes the system's state variables over time. The control theory is concerned with developing control input laws that accomplish a certain goal, usually minimizing cumulative cost over a trajectory. For example, one control problem is developing a law for temperature change in a rocket that propels the spaceship without causing an explosion [17].

Specifically, robot motion over time can be described with a system of nonlinear difference equations. Because robots are mechanical systems that can be moved using an external force, a special case of nonlinear systems, *a discrete-time*

control affine system, models their motion [69, 53, 94]. Consider a robot with d_f DOFs. If an acceleration $\mathbf{a}(k) \in \mathbb{R}^{d_f}$ is applied to the robot's center of mass at time-step k, the new position-velocity vector (state) $\mathbf{s}(k+1) \in \mathbb{R}^{2d_f}$ is,

$$\mathbf{D}: \quad \mathbf{s}(n+1) = \boldsymbol{g}(\mathbf{s}(k)) + \boldsymbol{f}(\mathbf{s}(k))\mathbf{a}(k), \tag{2.8}$$

for some functions $\boldsymbol{f}, \boldsymbol{g}$. Note that the system, Equation (2.8), satisfies the Markovian property from Definition 2.2.1.

To solve the optimal control problem often a reference trajectory is produced off-line in an open loop fashion, and then at run-time a lower level controller tracks the reference trajectory, adapting to unmodeled system dynamics and environmental factors [72]. Another technique first linearizes the system and then applies linear-quadratic-regulator (LQR) method locally [59]. All classical methods for solving non-linear control problems require knowledge of the system dynamics [59].

A second question in control theory is how to predict the behavior of given a *controlled system*, that is the system and its control law [17]. Under this scenario, we are interested in assessing the system's end state given an initial state. For example, a marble rolling down a side of a bowl will either eventually come to rest in the bottom of the bowl, or if it is pushed too hard, will escape the bowl on the other side. The ending state, which the system will not leave once reached, is called *equilibrium* [17].

Lyapunov stability theory gives us tools to assess the stability of an equilibrium. We say that a equilibrium is *stable in sense of Lyapunov* if an outcome of any initial condition sufficiently close to equilibrium, after certain amount of time, remains arbitrarily close to the equilibrium. Formally [45, 17],

Definition 2.4.1. *We say that a equilibrium $\boldsymbol{x}$ is* Lyapunov stable on X *if starting at*

any state in vicinity of the equilibrium, the systems remains close to it,

$$\forall \epsilon > 0, \exists \delta > 0, \forall \boldsymbol{y}(0) \in X \text{ such that } \|\boldsymbol{x} - \boldsymbol{y}(\mathbf{0})\| < \delta \Rightarrow \forall n \in \mathbb{N}, \|\boldsymbol{x} - \boldsymbol{y}(n)\| < \epsilon$$

If the trajectory converges to the equilibrium, the equilibrium point is asymptotically stable in the sense of Lyapunov.

$$\forall \epsilon > 0, \exists \delta > 0, \forall \boldsymbol{y}(0) \in X \text{ such that } \|\boldsymbol{x} - \boldsymbol{y}(\mathbf{0})\| < \delta \Rightarrow \lim_{n \to \infty} \|\boldsymbol{x} - \boldsymbol{y}(n)\| = 0$$

The consequence is that, similar starting conditions give similar results [17]. If the two outcomes converge to each other over time, we call that point *asymptotically stable equilibrium in the sense of Lyapunov*. Often used to show stability of origin, Lyapunov direct method is based on Lyapunov stability theorem, which gives sufficient conditions for stability of the origin [59, 45]. The method requires a construction of a positive, semi-definite scalar function of state $W : X \to \mathbb{R}$ that monotonically decreases along a trajectory and reaches zero in equilibrium [45]. This function can be loosely interpreted as the system's energy, positive and deceasing over time until it is depleted and the system stops. When the function is strictly monotonically decreasing, the origin is asymptotically stable [59]. Formally adapted from [45],

Theorem 2.4.1. *(Lyapunov Stability Theorem for Controlled Discrete-Time Systems). If in a neighborhood X of the equilibrium state $\boldsymbol{x}_0 = \mathbf{0}$ of the system, Equation* (2.8)*, there exists a function $W : X \times \mathbb{N} \to \mathbb{R}$ such that:*

1. *Nonnegative outside of equilibrium, $W(\boldsymbol{x}, n) > 0, \boldsymbol{x}(n) \neq \mathbf{0}$*
2. *Zero in the equilibrium, $W(\mathbf{0}, n) = 0$*
3. *Decreases along the trajectory $W(\boldsymbol{x}(n+1), n+1) - W(\boldsymbol{x}(n), n) \geq 0$,*

the equilibrium is stable. *Further, if the rate of change is strictly decreasing*

$$W(\boldsymbol{x}(n+1), n+1) - W(\boldsymbol{x}(n), n) > 0$$

outside of the origin, the origin is asymptotically stable. *W is called control Lyapunov function.*

When the control Lyapunov function is a measure of distance from the desired state (error), the system is guaranteed to progress to its equilibrium in a controlled fashion without increasing the error even temporarily. This because the Lyapunov stable equilibrium does not increase its control Lyapunov function over time. Construction of a control Lyapunov function proves the Lyapunov stability of the origin. However, construction of such a function is a non-trivial problem. Classically, the control law is written, and then a control Lyapunov function is constructed [17].

2.4.1 Optimal Control and RL

Efficient, near-optimal nonlinear system control is an important topic of research both in feedback controls and RL. When the system dynamics is known [137] develops adaptive control for interconnected systems. When the system dynamics is not known, optimal [86, 35, 127] and near-optimal [85, 24, 90] control for interconnected nonlinear systems are developed for learning the state-value function using neural networks. A generalized Hamilton-Jacobi-Bellman (HJB) equation for control-affine systems can be approximately solved with iterative least-squares [29]. For linear unknown systems [54] gives an optimal control using approximate dynamic programming. Convergence proofs exist for neural network-based approximate value iteration dynamic programming for linear [7] and control-affine systems [30], both with known dynamics.

Finally, when designing control for physical systems, safety is an important factor to consider. Task completion of a RL-planned motion can be assessed with Lyapunov stability theory. Perkins and Barto show how safe RL can be accomplished by choosing between predetermined control laws to secure Lyapunov stability and guarantee task completion [104]. Combination of control Lyapunov functions with RL results in stable and efficient power grid control [43]. Integral quadratic constraint framework trains neutral networks with RL with guaranteed stability [11].

Note that RL developed from the controls [23, 27] uses different notation than RL methods developed through machine learning and MDPs [120, 132]. For example, notation for states, state space, actions, action space, reward, and policy in MDP notation are $\mathbf{s}, S, \mathbf{a}, A, R, \pi$, and in controls notation are $\boldsymbol{x}, X, \boldsymbol{u}, U, \rho, \boldsymbol{h}$, respectively. In this thesis MDP notation is used.

2.4.2 Stochastic Control and RL

To address control under disturbances, piecewise linearization was used for quadrotor trajectory tracking under wind-gust disturbances [8]. Another approach requiring system dynamics knowledge uses harmonic potential fields for UAV MP in environments with a drift field [83]. Another need for handling disturbances is in aerial robotics for obstacle avoidance. Path planning and obstacle avoidance in the presence of stochastic wind for a blimp was solved using dynamic programming with discrete MDPs [58], and later with augmented MDPs [57]. Other methods to handle MP and trajectory generation under uncertainties use low-level controllers for stabilization of trajectories within reach tubes [33], or trajectory libraries [75].

2.5 Applications

This section describes the work related to the applications we use to evaluate PEARL. We cover Unmanned Aerial Vehicle (UAV) control, , Multi-Agent Pursuit Tasks, and robust sorting.

Unmanned aerial vehicles (UAVs) show potential for use in remote sensing, transportation, and search and rescue missions [64]. One such UAV, the quadrotor, is an ideal candidate for autonomous cargo delivery due to its high maneuverability, vertical takeoff and landing, single-point hover, and ability to carry loads 50% to 100% of their body weight. For example, cargoes may consist of food and supply delivery in disaster-struck areas, patient transport, or spacecraft landing. The four rotor blades of quadrotors make them easier to maneuver than helicopters. However, they are still inherently unstable systems with complicated nonlinear dynamics. The addition of a suspended load further complicates the system's dynamics, posing a significant control challenge. Planning motions that control the load position is difficult, so automated learning methods are necessary for mission safety and success. Recent research has begun to develop control policies for mini UAVs, including approaches that incorporate learning [64]. Learning system dynamics model parametrization has been successful for adaptation to changes in aerodynamics conditions and system calibration [88]. Schoellig et al. [113] use an expectation-maximization learning algorithm to achieve quadrotor trajectory tracking with a target trajectory and simple linear model. Lupashin et al. [74] apply policy gradient descent techniques to perform aggressive quadrotor multi-flips that improve over repeated iterations. They improve upon it in [73] by segmenting the trajectory into keyframes and learning the parameters for each segment separately. Even the task of suspended load delivery has been addressed for UAVs [101, 117, 21, 99]. Palunko et al. successfully applied dynamic programming to solve swing-free trajectories for quadrotors [99, 100] where they

showed an example of manual obstacle avoidance of a quadrotor with suspended load by generating swing-free trajectories using dynamic programming. Bernard et al. [21] developed a controller for helicopters with suspended loads using two feedback loops. Hehn and D'Andrea developed a controller for a quadrotor balancing an inverted pendulum [49]. Further examples of quadrotors interacting with the environment can be seen in aerial grasping [87, 106], ball throwing [108], and aerial robotic construction [133].

Swing-free trajectories have been studied outside of the UAV domain. They are important in industrial robotics with applications such as cranes in construction sites and for cargo loading in ports [5, 136]. Residual oscillation reduction is applicable to manufacturing environments where parts need to be transported in a limited space. Zameroski et al. [138] and Starr et al. [118] applied dynamic programming to reduce residual vibrations of a freely suspended payload. Schultz et al. [114] developed a controller for suspended mass.

Various variants of the pursuit-prey tasks have been studied in the RL and multi-agent literature [97, 135, 107]. A typical predator-pursuit task works in a grid world with multiple-agents pursing a prey. In the classical variant, both pursuers and the prey are autonomous agents. The pursuers work together as a team of independent agents [135]. Ogren et al. [98] approach multi-agent coordination through the construction of a control Lyapunov function that defines a desired formation, similar to our proposed solution. But, their approach uses a given control law to move the agents, while we use an RL-derived planner.

Voronoi decomposition solves high-dimensional manipulation problems by projecting the robot's configuration space onto a low-dimensional task space [116]. Another RL approach solves manipulation tasks with hard [119] and soft [65] constraints.

Reinforcement programming has used Q-learning to generate an array sorting program using more traditional programming constructs, such as counters, *if* statements, *while* loops, etc., as MDP actions [131]. Our implementation, in contrast, considers element insertion as the only possible action. Ackley motivated software development practices for robust software where the programs are resilient to unrealiable information for array sorting [3].

Chapter 3

PrEference Appraisal Reinforcement Learning (PEARL)

This chapter presents PEARL, the proposed solution for solving PBT tasks in high-dimensional spaces for robotic systems with unknown dynamics. This chapter contributes:

- PEARL overview in Section 3.1.
- PBT definition and a feature extractor in Sections 3.1.1 and 3.1.2.
- A solution to high-dimensional robotics Multi-Agent Pursuit Task in Section 3.2.

PBTs are defined with a recognizable goal state, and a set of often opposing preferences that the trajectory should regard while progressing the system to the goal. These tasks can be difficult or impossible to demonstrate, e.g., a Mars landing, or quadrotor Minimal Residual Oscillations Task (Definition 4.1.4). But the preferences, guiding principles, can be described. Unlike soft and hard-constraints, discussed in Section 2.3.6, our tasks do not have known constraints

and bounds; they are set up as preferences to guide dynamically feasible trajectory generation. What is unknown, though, are the priorities between the preferences that generate dynamically feasible trajectories to the goal. To cope with the learning complexity, the PEARL features define a basis in the task space similarly to the methods introduced in Section 2.5. However, PEARL autonomously learns the relationship between the features. The number of features is invariant to the problem dimensionality, and the computation time scales polynomially with the state space dimension. Given the feature selection, the Multi-Agent Pursuit Task, used for demonstration, learns with only three pursuers. The resulting planner plans trajectories for a large number of agents, working with several thousand dimensions offline, and in real-time at 50 Hz for 100-dimensional problems.

In contrast to the related work in Section 2.4.1, PEARL designs features and constructs policy approximation that ensures Lyapunov stability in the deterministic case. We empirically verify the goal's stability and the quality of the resulting trajectories.

3.1 PEARL for Robotic Problems

PEARL solves PBT in two phases, learning and acting (Figure 3.1), adhering to the batch RL paradigm. Recall from Section 2.3.1 that on-line learning must perform on-going exploration, which can be potentially unsafe for the physical hardware and the environment. For that reason, our solution uses batch RL. The learning phase uses one of the AVI-based RL algorithms. The value function is approximated with a with linear map of features, or preferences. The learning agent discovers the relative weights between the preferences (preference appraisal). Once the value function approximation is learned, we can use it to generate any number of trajectories. These trajectories can have different starting and ending positions

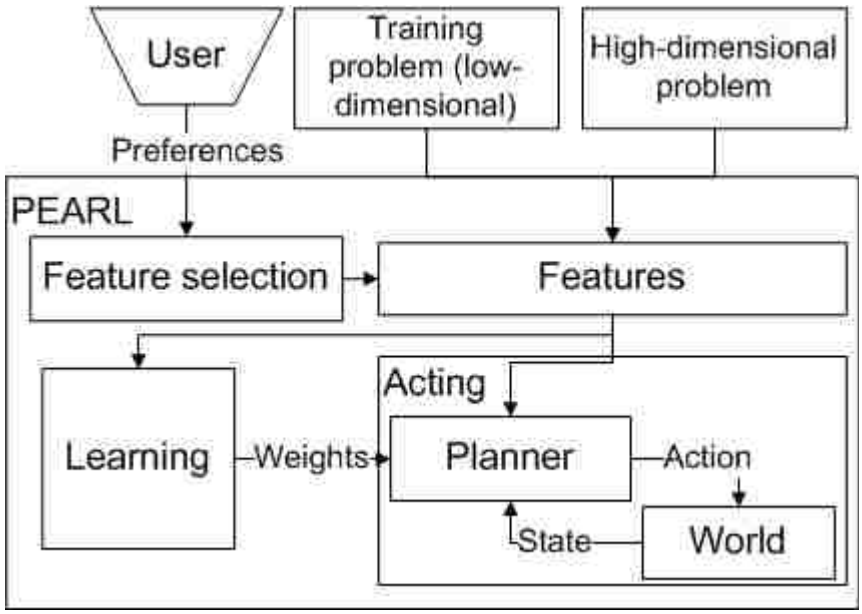


Figure 3.1: PrEference Appraisal Reinforcement Learning (PEARL) framework for learning and executing PBT. The user-provided preferences are encoded into polymorphic features. The learning agent appraises preference priorities on a low-dimensional training problem. The planner takes an initial state of a high-dimensional problem and produces actions in a closed feedback loop.

and use different (but compatible) models. In Chapter 4 we find the sufficient criteria to allow the transfer of the learned policy to a variety of situations.

The acting phase produces a trajectory that performs the task using the preferences and the learned weights. The acting is a closed-loop feedback system, or a decision-making as described in Chapter 2, that can work on-line or plan trajectories off-line in simulation. We will examine in detail the learning and acting phases in Chapters 4, 5, and 6. Chapter 4 will present a basic greedy planner that works with discrete actions. Chapter 5 will give policy approximations for continuous actions MDPs, and will show convergence to the task goal for control-affine systems. Chapter 6 will modify the greedy policy for the stochastic systems.

The learner and planner share features. We have seen in Chapter 2 that RL is sensitive to the feature selection. Our proposed features, presented in Section 3.1.2 are polymorphic and separate PEARL from standard batch RL. They enable both the learning on small problems and the policy transfer to high-dimensional problems. Later, in Chapter 4, we will analyze conditions for changing MDPs (states, actions, and simulators) between learning and planning.

3.1.1 MDP Setup

Because single-robot problems are a special cases of multi-robot problems, we present a MDP setup and a feature creation for a generalized multi-robot problem. The multi-robot system consists of d_r robots, where i^{th} robot has d_{ri} DOFs. We assume the robots work in continuous state and action spaces, are controlled through acceleration applied to their center of mass, and have the dynamics that is not explicitly known. Let $\mathbf{s}_i,\ \dot{\mathbf{s}}_i,\ \ddot{\mathbf{s}}_i \in \mathbb{R}^{d_{ri}}$ be the i^{th} robot's position, velocity, and acceleration, respectively. The MDP state space is $S = \mathbb{R}^{d_s}$, where $d_s = 2\sum_{i=1}^{d_r} d_{ri}$ is the dimensionality of the joint multi-robot system. The state $\mathbf{s} \in S$ is joint vector $\mathbf{s} = [\mathbf{s}_1, ..., \mathbf{s}_{d_r}, \dot{\boldsymbol{s}}_1, ..., \dot{\boldsymbol{s}}_{d_r}]^T$, and action $\mathbf{a} \in A = \mathbb{R}^m$ is the joint acceleration vector, $\mathbf{a} = [\ddot{\boldsymbol{s}}_1, ..., \ddot{\boldsymbol{s}}_{d_r}]^T$. The state transition function that we assume is unknown, is a joint dynamical system of individual robots $\mathbf{D} = \mathbf{D}_1 \times ... \times \mathbf{D}_{d_r}$, each being control-affine (2.8).

We assume the presence of a simulator or dynamics samples for each robot, for training purposes. The reward R is set to one when the joint multi-robot system achieves the goal, and zero otherwise. The tuple defines the joint MDP for the multi-robot problem. A feature vector linear map, $\hat{V}(\mathbf{s}) = \boldsymbol{\theta}^T \boldsymbol{F}(\mathbf{s}, d_r)$, approximates the state-value function.

3.1.2 Feature Selection

PBTs that we consider, are defined with n_o objectives, $\boldsymbol{o}_1, ..., \boldsymbol{o}_{n_o}$, and preferences with respect to the objectives. The objectives are points in positional or velocity space, $\boldsymbol{o}_i \in \mathbb{R}^{d_{ri}}$, $i = 1, .., n_o$. There are two types of preferences that can be associated with an objective: *distance-reducing* and *intensity-reducing*; both preference types have the goal of reducing their measure to an objective. The *distance-reducing* preferences aim at reducing the distance to the objective, while the *intensity-reducing* preferences reduce the inverse of the square of distance (intensity). For example, swing-free quadrotor flight (Chapters 4 and 5) has four distance-reducing features, distance from the goal, speed magnitude, load's distance from the resting position, and load's speed magnitude. An intensity-reducing feature would be the distance from an obstacle.

To learn PBT with n_o objectives, $\boldsymbol{o}_1, ..., \boldsymbol{o}_{n_o}$, we form a feature for each objective. Assuming the low-dimensional task space and high-dimensional MDP space $n_o \ll d_s$, we consider *task-preference features*, $\boldsymbol{F}(\mathbf{s}, d_s) = [F_1(\mathbf{s}, d_s), ..., F_{n_o}(\mathbf{s}, d_s)]^T$. Parametrized with the state space dimensionality, d_s, the features map the state space S to the preference space and, depending on the preference type, measure either the squared distance to the objective or intensity. Let $S_i \subset \{1, .., d_r\}$ be a subset of robots that an objective $\boldsymbol{o}_i$ applies to, and $\boldsymbol{p}_j^{\boldsymbol{o}_i}(\mathbf{s})$ be a projection of j^{th} robot's state onto minimal subspace that contains $\boldsymbol{o}_i$. For instance, when an objective $\boldsymbol{o}_i$ is a point in a positional space, $\boldsymbol{p}_j^{\boldsymbol{o}_i}(\mathbf{s})$ is the j^{th} robot's position. Similarly when $\boldsymbol{o}_i$ is a point in a velocity space, $\boldsymbol{p}_j^{\boldsymbol{o}_i}(\mathbf{s})$ is j^{th} robot's velocity. Then, *distance-reducing features* are defined with

$$F_i(\mathbf{s}, d_s) = \sum_{j \in S_i} \|\boldsymbol{p}_j^{\boldsymbol{o}_i}(\mathbf{s}) - \boldsymbol{o}_i\|^2,$$

and *intensity-reducing features* are defined with

$$F_i(\mathbf{s}, d_s) = (1 + \sum_{j \in S_i} (\|\boldsymbol{p}_j^{\boldsymbol{o}_i}(\mathbf{s}) - \boldsymbol{o}_i\|^2))^{-1}.$$

Algorithm 3.4 summarizes the feature selection procedure.

Algorithm 3.4 PEARL feature selection.

Input: $\boldsymbol{o}_1, ..., \boldsymbol{o}_{n_o}$ objectives, $pt_1, ..., pt_{n_o}$ preference types

Input: MDP $\mathcal{M}$ $(S, A, \mathbf{D}, R)$,

Output: $\boldsymbol{F}(\mathbf{s}, d_s) = [F_1(\mathbf{s}, d_s), ..., F_n(\mathbf{s}, d_s)]^T$

1: **for** $i = 1, \ldots, n_o$ **do**
2: **if** $pt1$ is intensity **then**
3: $Fi_i(\mathbf{s}, d_s) = (1 + \sum_{j \in S_i} (\|\boldsymbol{p}_j^{\boldsymbol{o}_i}(\mathbf{s}) - \boldsymbol{o}_i\|^2))^{-1}$ {inensity preference}
4: **else**
5: $F_i(\mathbf{s}, d_s) = \sum_{j \in S_i} \|\boldsymbol{p}_j^{\boldsymbol{o}_i}(\mathbf{s}) - \boldsymbol{o}_i\|^2$, {distance preference}
6: **end if**
7: **end for**
8: return $\boldsymbol{F}(\mathbf{s}, d_s)$

Features selected in this manner have the following properties allowing PEARL to have the potential to learn on small problems and transfer the learning to larger problems:

- *Feature domain:* The features are Lipschitz continuous and defined for the entire state space, $F_i : \mathbb{R}^m \to \mathbb{R},\ i = 1, .., d_s$, in contrast to tiling and radial basis functions [135, 27] that are active only on a section of the problem domain.
- *Projection to preference space:* Features project state subspace into a point that measures the quality of the preferences. Thus, the state-value function approximation $\hat{V} = \boldsymbol{\theta}^T \boldsymbol{F}(\mathbf{s}, d_s)$ is an d_s-dimensional manifold in the preference

space, and their number does not change as the team size or domain space change.

- *State space polymorphism:* Because they are based on the vector norm and projection, the features are polymorphic with respect to the domain dimensionality. Since the learning is more computationally intensive than the planning, we use lower-dimensional problems for training. The feature vector size is invariant to the number of agents, state space dimensionality, and physical space dimensions. If the agents operate in 2D space, the features consider only planar space. But, when the same agents are placed in an 3D environment, the feature set remains unchanged although the 3D space is considered in feature calculations.

- *Polynomial computation time:* The feature computation time is polynomial in state space dimensionality.

Problems with only distance reducing features are guaranteed to complete the task if the learning converges to all negative parameters as we will see in Theorem 5.1.2. In the problems with mixed objectives, the agents will follow preferences, but there are no formal completion guarantees. Thus, an empirical study evaluates the method.

3.2 Case Study: Large-scale Pursuit Task

We first demonstrate PEARL on a Multi-Agent Pursuit Task. Our variant of the task considers only the pursuers chasing the prey, while the prey follows a predetermined trajectory unknown to the pursuers. The problem formulation we use differs from the typical formulation (see Section 2.5) in three ways. First, it works in an arbitrarily large continuous state and action space rather than the discretized

grid world. Second, it learns and plans in joint agent-position-velocity space, and handles variable team size without additional learning. Lastly, we require the pursuers to keep distance among themselves, thus we learn and plan in the joint pursuers' space. We formulate the task as follows:

Definition 3.2.1. *(Pursuit task.) A set of r agents must follow the prey agent in close proximity, while maintaining distance among themselves. The agents know the current position and velocity of the prey (leader), but do not know the future headings.*

To solve the pursuit task we set up a MDP and a feature vector (preferences) as outlined in Section 3.1.2. Assuming d_r agents, with d_f DOF each, the state space is the joint agent-position-velocity vector space $S = \mathbb{R}^{2d_f d_r}$. State $\mathbf{s}$'s coordinates $\mathbf{s}_{ij}, \dot{\mathbf{s}}_{ij}$, $i = 1,..,d_r$, $j = 1..d_f$ denote i^{th}'s agent position and velocity in direction of axis j. The action space A are acceleration vectors on the agents, $A = \mathbb{R}^{d_f d_r}$, $\ddot{\mathbf{s}}_{ij}$ is i^{th} agent's acceleration in the j direction. To design the feature vector for the pursuit task, we look at the qualitative preferences from the task definition. From the problem description, three preferences and objectives are desired: close proximity to the prey (positional objective), following the prey (velocity objective), and maintaining distance between the agents (positional objective). The first two are *distance-reducing*, while the last one is an *intensity-reducing* feature. Thus, the feature vector has three components $\boldsymbol{F}(\mathbf{s}, d_f, d_r) = [F_1(\mathbf{s}, d_f, d_r) \;\; F_2(\mathbf{s}, d_f, d_r) \;\; F_3(\mathbf{s}, d_f, d_r)]^T$. Following the method from Section 3.1.2, we express the distance to the prey as $F_1(\mathbf{s}, d_f, d_r) = \sum_{i=1}^{d_r} \sum_{j=1}^{d_f} (\mathbf{s}_{ij} - \boldsymbol{p}_j)^2$, and *following the prey*, as minimizing the difference in velocities between the agents and the prey, $F_2(\mathbf{s}, d_f, d_r) = \sum_{i=1}^{d_r} \sum_{j=1}^{d_f} (\dot{\mathbf{s}}_{ij} - \dot{\boldsymbol{p}}_j)^2$ where $\boldsymbol{p}_i$ and $\dot{\boldsymbol{p}}_i$ are prey's position and velocity in the direction i. The last feature is maximizing the distance between the agents, $F_3(\mathbf{s}, d_f, d_r) = (1 + \sum_{i,j=1}^{d_r} \sum_{k=1}^{d_f} (\mathbf{s}_{ik} - \mathbf{s}_{jk})^2)^{-1}$.

Learning results: To learn the pursuit task, we use 3 holonomic planar agents ($d_f = 2$, $d_r = 3$). The pursuit team of three robots is the smallest team for

which all features are non-trivial. The maximum absolute accelerations less than $3\,\mathrm{m\,s^{-2}}$. We run CAFVI (Algorithm 5.7) as the learning agent in PEARL with Axial Sum Policy (Equation (5.11)) for 300 iterations to learn the feature vector weights. The sampling space is inside a six-dimensional hypercube $[-0.4\,\mathrm{m},\, 0.4\,\mathrm{m}]^6$. The prey is stationary at the origin during learning. The resulting weights are $\boldsymbol{\theta} = [-16.43 - 102.89 - 0.77]^T$. Both learning and planning were performed on a single core Intel Xeon W3520 running Matlab 2013a. All simulations are done at 50 Hz. The time to learn is 145 s.

Planning results: To plan a task, we assign a trajectory to the prey, and increase the number of agents. The prey starts at the origin, and the pursuers start at random locations within 5 m from the origin. We plan a 20 s trajectory.

Figure 3.2 depicts the planning computational time for a 20 s trajectory as the number of pursuers increase. State and action spaces grow with the team size, and the planning time stays polynomial. This is because the feature vector scales polynomially with the state size. The method plans the problems with continuous state spaces up to $\mathbb{R}^{100}$ and continuous actions up to $\mathbb{R}^{50}$ in real-time (below the green line in Figure 3.2), as trajectory computation takes less time than its execution.

The next evaluation looks at 25-agent pursuit of a prey that tracks either a spiral (Figs. 3.3a - 3.3c) or lemniscate de Gerono curve (Figs. 3.3d- 3.3f). The agents start at random locations uniformly drawn within 5 m from the leader's initial position (origin). Figs. 3.3a and 3.3d show the movement of the agents in the xy-plane. Although the pursuers do not know the future leader's positions, they start tracking in increasingly close formation. Figs. 3.3b and 3.3e show the x-axis, and Figs. 3.3c and 3.3f show the y-axis trajectory over time. Note that the initial positions are uniformly distributed, and after 7 s all agents are close to the leader, and remain in phase with it. The only exception is the y-coordinate of the

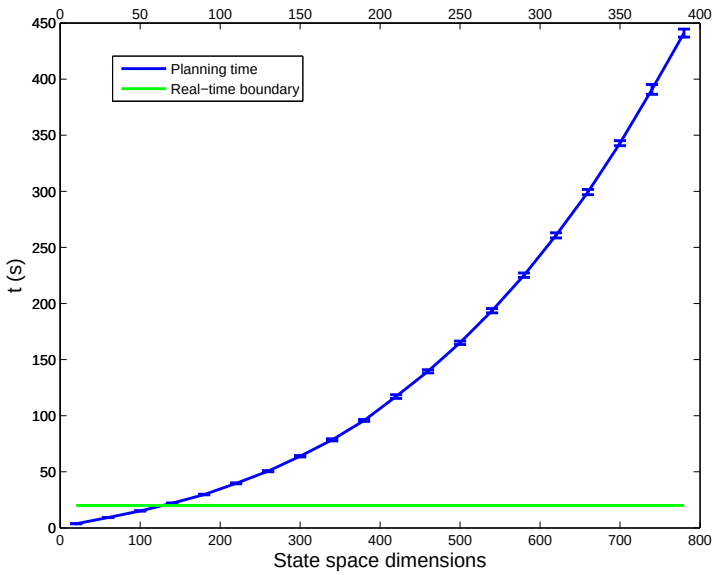


Figure 3.2: Time to plan a 20 s Multi-Agent Pursuit trajectory per state and action space dimensionality averaged over 10 trials. The problem sizes below the solid green line can be solved in real-time.

lemniscate (Figure 3.3f). Here, the agents never catch up with the leader. The higher frequency oscillations in the y-direction make the prey hard to follow, but the agents maintain a constant distance from it.

Figs. 3.3g - 3.3i depict a 20 s trajectory of a 1000-pursuer task. The state space in this case is $\mathbb{R}^{4000}$ and the action space is $\mathbb{R}^{2000}$. Axial Sum Policy successfully plans a trajectory in this large-scale space. The leader's trajectory has Brownian velocity (its acceleration is randomly drawn from a normal distribution). Following this trajectory, the prey travels further from the origin, than the previous two examples, the spiral and the lemniscate.

Table 3.1 shows the summary of ending trajectory characteristics averaged over 100 trials. The leader follows one of the three trajectories: straight line, spiral, and lemniscate. For each trajectory we plan the pursuit task with a varying number of pursuers. For all tasks the 20 s pursuit takes less than 20 s to com-

pute even in the 100-dimensional state spaces (Table 3.1), confirming the timing benchmark in Figure 3.2. The next column, *Leader distance*, shows that the average pursuer-prey distance remains below 30 cm. The minimal standard deviation suggests consistent results, in agreement with the trajectory plots (Figure 3.3). The last column shows the average distance between the agents that was to be maximized. The distance ranges between 11 cm and 46 cm. It decreases as the pursuer team grows, reflecting higher agent density around the prey and a tighter formation. The motion animation[1] shows that while approaching the leader, the agents maintain the same formation and relative positioning, thus avoiding collisions.

The pursuit task case study empirically demonstrates PEARL for the high-dimensional multi-robot problems outlined in Section 3.1.2. The RL with problem-specific hand-crafted features are evaluated and compared to other methods such as dynamic programming and control-laws (Section 4.3.2 and 5.2.2) and outperform the other methods both in speed and precision. In the instance of UAV with the supsended load, the experiments on physical robots perform within centimeters from simulation predictions verifying the fidelity of the simulation results (Section 4.3.2). For these reasons, we focused here on the demonstration of the feature creation method rather than comparison with other methods. Note that the *intensity-reducing* feature that maximizes the space between the agents does not conform to the *distance-reducing* features considered in Sections 4.3.2 and 5.2.2. The value function approximation for this problem, does not have a single maximum, instead it has a cleft between two ridges. This extension and condition relaxation, expands the applicability of the continuous action value approximate value iteration with Axial Sum Policy to a wider class of problems. We also demonstrated that the method produces the same qualitative results in very high-dimensional spaces as well as lower dimensional spaces.

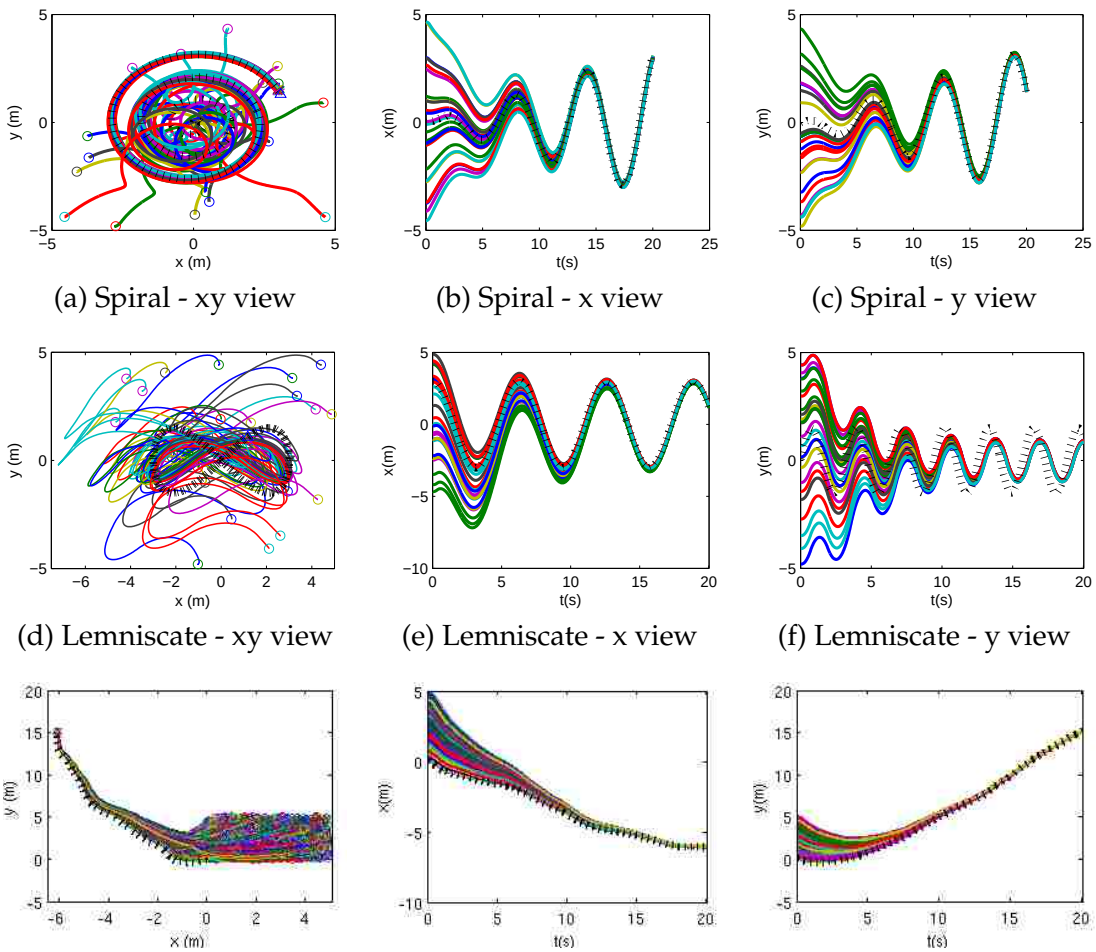
[1] Available at https://cs.unm.edu/amprg/People/afaust/afaustCh3.mp4

Table 3.1: Trajectory characteristics after 20 s of simulation as a function of the leader's trajectory and the number of concurrent agents. State (# State Dim.) and action (# Action Dim.) space dimensionality, computational time (Comp. Time), average distance from the leader (Leader Dist.), and distance between the agents (Dist. Agents) are shown. All results are averaged over 100 trajectories starting from randomly drawn initial states within 5 m from the origin.

Leader Trajectory	# Agents	# State Dim.	# Action Dim.	Comp. Time (s) μ	Comp. Time (s) σ	Leader Dist. (m) μ	Leader Dist. (m) σ	Dist. Agents (m) μ	Dist. Agents (m) σ
Line	5	20	10	4.18	0.09	0.29	0.01	0.46	0.00
	10	40	20	7.26	0.10	0.16	0.00	0.23	0.00
	15	60	30	10.62	0.14	0.11	0.00	0.16	0.00
	20	80	40	13.62	0.16	0.09	0.00	0.12	0.00
	25	100	50	17.21	0.03	0.08	0.00	0.11	0.00
Spiral	5	20	10	4.05	0.07	0.34	0.01	0.46	0.00
	10	40	20	7.22	0.13	0.24	0.01	0.23	0.00
	15	60	30	10.21	0.13	0.22	0.01	0.15	0.00
	20	80	40	13.31	0.05	0.22	0.01	0.12	0.00
	25	100	50	16.98	0.04	0.22	0.01	0.10	0.00
Lemniscate	5	20	10	4.08	0.05	0.36	0.01	0.46	0.00
	10	40	20	7.18	0.09	0.29	0.01	0.22	0.00
	15	60	30	10.21	0.07	0.27	0.01	0.15	0.00
	20	80	40	13.33	0.14	0.26	0.00	0.11	0.00
	25	100	50	16.90	0.09	0.26	0.00	0.09	0.00

3.3 Conclusion

This chapter presented PEARL framework, a solution for high-dimensional preference-balancing motion problems. The method uses features that work in continuous domains, scale polynomially with the problem's dimensionality, and are polymorphic with respect to the domain dimensionality.



(a) Spiral - xy view (b) Spiral - x view (c) Spiral - y view

(d) Lemniscate - xy view (e) Lemniscate - x view (f) Lemniscate - y view

(g) Brownian vel. - xy view (h) Brownian vel. - x view (i) Brownian vel. - y view

Figure 3.3: Multi-Agent Pursuit Task learning transfer. Three different pursuit tasks planned with the same learning. The prey following a spiral (a-c) and a lemniscate curve (d-f) chased by 25 agents. 1000-agent pursuit task of a prey that follows a random acceleration trajectory (g-i). The prey's trajectory is a dotted black line.

Chapter 4

PEARL for Deterministic Discrete Action MDPs

This chapter uses PEARL in discrete action spaces to create a fully automated software system that plans and generates trajectories in obstacle-laden environments for Cargo Delivery Task. The chapter's content is based on [39, 40]. The contributions of this chapter are:

- RL agent that creates trajectories with minimal residual oscillations for a suspended load bearing UAV in obstacle-free spaces (Sections 4.2.1 and 4.2.2).
- RL path following agent that reduces the load displacement (Section 4.2.3),
- A framework for trajectory generation with constraints that avoids obstacles (Section 4.2.5),
- RL agent's integration with sampling-based path planning (Section 4.2.5), and
- Development of an efficient rigid body model to represent a quadrotor carrying a suspended load (Section 4.2.4).

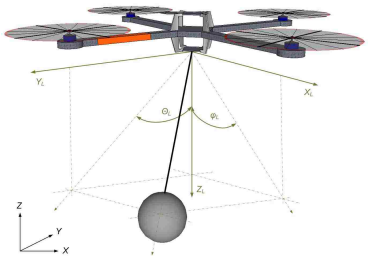
To learn control policy for minimizing the load displacement, we use PEARL with AVI as a learning agent with a specifically designed feature vector for state-value function approximation. There are two challenges we face. First, the AVI requires a state-space sampling domain to learn the policy. The learning must have sufficient sample-density to be successful. To provide sufficient sample-density while learning, we sample in the small state subspace around the goal but choose a feature vector that is defined beyond the sampling subspace. The second challenge we face, is that the planning for minimal residual oscillations motion consists of a large action space (over 10^6 actions). Such a large action space is impractical for learning, thus we learn in an action subspace, and plan with the larger action space. Relying on Lyapunov stability theory, the chapter contributes sufficient conditions that the MDP formulation (system dynamics, action space, and state-value function approximation) must meet to ensure the cargo delivery with minimal residual oscillations. Within the conditions we are free to change MDP properties as it suits our needs and to transfer learning to compatible MDPs. For example, for practicality we learn in a 2-dimensional action subspace and transfer the state-value function approximation to MDP with a 3-dimensional action space to plan altitude changing trajectories. As another example, we develop a novel path-following agent that minimizes the load displacement by action-space adaptation at every planning step. In the context of related work in the obstacle-free case, we transfer to MDPs with state and action supersets and noisy dynamics using a behavior transfer function that transfers directly the learned value function approximation to the new domain with no further learning.

In contrast to value function transfer with behavior transfer functions we transfer the learned value function approximation to tasks with state and action space supersets and changed dynamics. We find sufficient characteristics of the target tasks for learning transfer to occur successfully. The direct transfer of the value function is sufficient and requires no further learning. In contrast to trans-

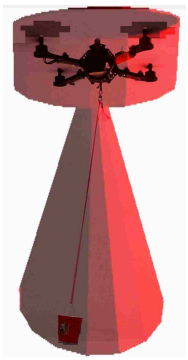
ferring only the most relevant actions, in obstacle-free spaces, we take the opposite approach; to save computational time, we learn a sub-optimal policy on a subset of actions, and transfer it to the expanded action space to produce a more refined plan. When planning a path-following trajectory, we work with a most relevant subset of the expanded action space. Unlike the partial policies that focus on the states are most likely to be encountered, described in Section 2.3.6, we are interested in finding Minimal Residual Oscillations Trajectories from different start states, but we do have a single goal state. Thus, all trajectories will pass near the goal state, and we learn the partial policy only in the vicinity of the goal state. Then, we may apply it to any start state.

For the problem of suspended load swing reduction, we apply RL to automatically generate, test, and follow swing-free trajectories for a quadrotor. RL is integrated into both the motion planning and trajectory generation for a rotorcraft equipped with a suspended load and in an environment with static obstacles. In Section 4.3.2 we show that RL could be used to reduce load displacement at the arrival to the goal state. In Section 4.3.4, the agent is placed in a larger context o f time-sensitive cargo delivery tasks. In this class of applications, the payload should be delivered free of collision as soon as possible, possibly following a reference path, while bounded load displacement is maintained throughout the trajectory. Beyond aerial robotics, the methods presented here are applicable to any PBT posed on a dynamical system, because aerial cargo delivery problem is such a task. We test the proposed methodology in Section 5.2 both in simulation and experimentally on a physical system (Figure 4.1a).

(a) Experimental system



(b) Position definition



(c) Geometric model

Figure 4.1: Quadrotor carrying a suspended load.

4.1 Preliminaries

This chapter is concerned with joint UAV-suspended load systems. The load is suspended from the UAV's center of mass with an inelastic cable. A ball joint models the connection between the UAV's body and the suspension cable. We now define several terms that are used in this paper extensively.

Definition 4.1.1. *(Load Displacement or Load Swing): is the position of the load, at time t, expressed in coordinates $\boldsymbol{\eta}(t) = [\phi(t)\ \omega(t)]^T$ with the origin in quadrotor's center of the mass (see Figure 4.1b).*

Definition 4.1.2. *(Minimal Residual Oscillations trajectory): A trajectory of duration T is a* minimal residual oscillations *trajectory if for a given constant $\epsilon > 0$ there is a time $0 \leq t_1 \leq T$, such that for all $t \geq t_1$, the load displacement is bounded with ϵ, i. e. $\|\boldsymbol{\eta}(t)\| < \epsilon$.*

Definition 4.1.3. *(Bounded Load Displacement or Swing-free trajectory): A trajectory is a* swing-free *trajectory if it is the minimal residual oscillation trajectory for time $t_1 = 0$,*

in other words if the load displacement is bounded by a given constant throughout the entire trajectory, i. e. $\|\boldsymbol{\eta}(t)\| < \epsilon,\ t \geq 0$.

Now, we present problem formulations for two aerial cargo delivery tasks, which specify the trajectory characteristic, task completion criteria, and dynamical constraints on the system.

Definition 4.1.4. *(Minimal Residual Oscillations Task): Given start and goal positional states* $\mathbf{s}_s, \mathbf{s}_g \in$ *C-free, and a swing-constraint* ϵ*, find a minimal residual oscillations trajectory* $\mathbf{s} = \boldsymbol{\tau}(t)$*,* $(0 \leq t \leq T)$ *from* $\mathbf{s}_s$ *to* $\mathbf{s}_g$ *for a holonomic UAV carrying a suspended load. The task is completed when the system comes to rest at the goal state: for some small constants* ϵ_d, ϵ_v*, at time* $t_g \leq T$ $\|\boldsymbol{\tau}(t_g) - \mathbf{s}_g\| \leq \epsilon_d$ *and* $\|\dot{\boldsymbol{\tau}}(t_g)\| \leq \epsilon_v$*. The dynamical constraints of the system are the bounds on the acceleration vector.*

In the above definition, the system completes the task if it reaches and remains inside a small interval around origin in the system's position-velocity space. The small constants ϵ_d, and ϵ_v define the interval size. In contrast, the swing-constraint can be met at a different time during the trajectory. Aerial cargo delivery, described next, requires the load displacement, ϵ, to be bounded throughout the trajectory, while the position and velocity constants, ϵ_d, ϵ_v, must be reached at the end of the trajectory at some time t_g.

Definition 4.1.5. *(Cargo Delivery Task): Given start and goal positional states* $\mathbf{s}_s, \mathbf{s}_g \in$ *C-free, and a swing-constraint* ϵ*, find a swing-free trajectory* $\mathbf{s} = \boldsymbol{\tau}(t)$ *from* $\mathbf{s}_s$ *to* $\mathbf{s}_g$ *for a holonomic UAV carrying a suspended load. The task is completed when the system comes to rest at the goal state: for some* ϵ_d, ϵ_v*, at time* t_g $\|\boldsymbol{\tau}(t_g) - \mathbf{s}_g\| \leq \epsilon_d$ *and* $\|\dot{\boldsymbol{\tau}}(t_g)\| \leq \epsilon_v$*. The dynamical constraints of the system are the bounds on the acceleration vector.*

4.2 Methods

Our goal is to develop a fully automated agent that calculates aerial cargo delivery (Problem 4.1.5) trajectories in environments with static obstacles. The trajectory must be collision-free, the load displacement must be bounded, and the UAV must arrive at the given goal coordinates. Within these constraints, the task needs to complete in the shortest time given the physical limitations of the system combined with the task constraints.

Figure 7.1a presents the proposed architecture. We learn the policy that reduces load displacement separately from the space geometry, and then combine them to generate trajectories with both characteristics. The minimal residual oscillations policy, described in Section 4.2.1, performs minimal residual oscillations task (Problem 4.1.4). Once the policy is learned, it can be used with varying state and action spaces to refine system performance according to the task specifications. Based on Lyapunov theory Section 4.2.2 discusses the sufficient conditions for these modifications. In Section 4.2.3, we adapt the policy to follow a path, by modifying the action space at every step. To enable obstacle-avoidance, we use a planner to generate a collision-free path. Section 4.2.4 describes the geometric model of the cargo-bearing UAV that we use for efficient collision detection. After the policy is learned and a roadmap constructed, we generate trajectories that are both collision-free, and guarantee load displacement below a given upper bound. For given start and goal coordinates, the planner calculates a collision-free path between them. A path-following trajectory that maintains the acceptable load displacement is calculated using the method described in Section 4.2.5.

The architecture has two clear phases: *learning* and *planning*. The learning for a particular payload is performed once, and used many times in any environment. The problem geometry is learned, and PRMs are constructed once, for a given

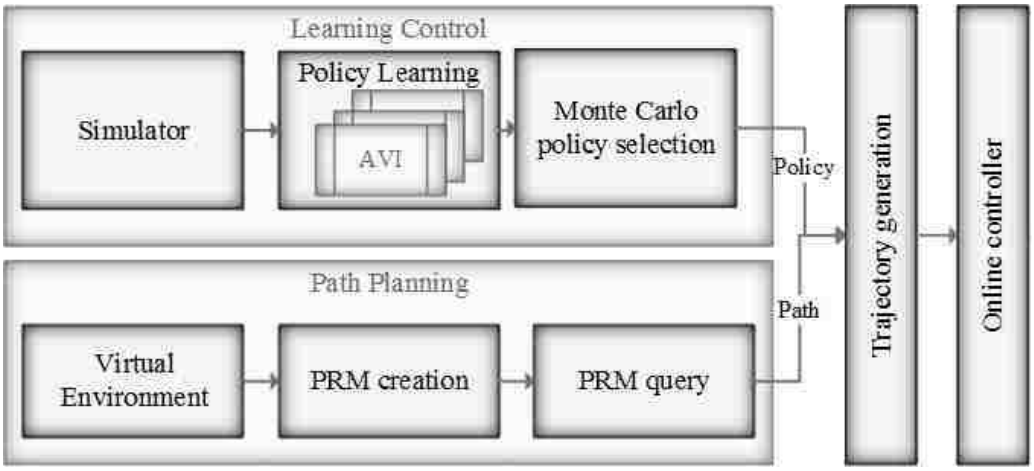


Figure 4.2: Automated aerial cargo delivery software architecture.

environment and a maximum allowed load displacement. When constructed, roadmaps can be used for multiple queries in the environment for tasks requiring the same or smaller load displacement. The distinct learning phases and the ability to reuse both, the policy and the roadmap, are desirable and of practical use, because the policy learning and roadmap construction are time consuming.

4.2.1 Learning Minimal Residual Oscillations Policy

In this section[1], our goal is to find fast trajectories with minimal residual oscillations for rotorcraft aerial robots carrying suspended loads in obstacle-free environments. We assume that we know the goal state of the vehicle; the initial state may be arbitrary. Furthermore, we assume that we have a *black box* system's simulator (or a generative model) available, but our algorithm makes no assumptions

[1]© 2013 IEEE. This section is reprinted, with permission, from Aleksandra Faust, Ivana Palunko, Patricio Cruz, Rafael Fierro, and Lydia Tapia, "Learning Swing-free Trajectories for UAVs with a Suspended Load," IEEE International Conference on Robotics and Automation (ICRA), May 2013

about the system's dynamics.

In our implementation, we use a deterministic MDP. The state space S is set of all vectors $\mathbf{s} \in S$, such that $\mathbf{s} = [x\, y\, z\, \dot{x}\, \dot{y}\, \dot{z}\, \phi_L\, \omega_L\, \dot{\phi}_L\, \dot{\omega}_L]^T$ (Figure 4.1b). Vector $\boldsymbol{p} = [x\ y\ z]^T$ is the position of the vehicle's center of mass, relative to the goal state. The vehicle's linear velocity is vector $\boldsymbol{v} = [\dot{x}\ \dot{y}\ \dot{z}]^T$. Vector $\boldsymbol{\eta} = [\phi_L\ \omega_L]^T$ represents the angles that the suspension cable projections onto xz and yz planes form with the z-axis (see Figure 4.1b). The vector of the load's angular velocities is $\dot{\boldsymbol{\eta}}_L = [\dot{\phi}_L\ \dot{\omega}_L]^T$. L is the length of the suspension cable. Since L is constant in this work, it will be omitted. To simplify, we also refer to the state as $\mathbf{s} = [\boldsymbol{p}^T\ \boldsymbol{v}^T\ \boldsymbol{\eta}^T\ \dot{\boldsymbol{\eta}}^T]^T$ when we do not need to differentiate between particular dimensions in $\mathbf{s}$. The action space, A is a set of linear acceleration vectors $\mathbf{a} = [\ddot{x} \quad \ddot{y} \quad \ddot{z}]^T$ discretized using equidistant steps centered around zero acceleration.

The reward function, R, penalizes the distance from the goal state, and the load displacement angle. It also penalizes the negative z coordinate to provide a bounding box and enforce that the vehicle must stay above the ground. Lastly, the agent is rewarded when it reaches the goal. The reward function $R(\mathbf{s}) = \boldsymbol{c}^T \boldsymbol{r}(\mathbf{s})$ is a linear combination of basis rewards $\boldsymbol{r}(\mathbf{s}) = [r_1(\mathbf{s})\ r_2(\mathbf{s})\ r_3(\mathbf{s})]^T$, weighted with vector $\boldsymbol{c} = [c_1\ c_2\ c_3]^T$, for some constants a_1 and a_2, where:

$$r_1(\mathbf{s}) = -\|\boldsymbol{p}\|^2,$$

$$r_2(\mathbf{s}) = \begin{cases} a_1 & \|\boldsymbol{F}(\mathbf{s})\| < \epsilon \\ -\|\boldsymbol{\eta}\|^2 & \text{otherwise} \end{cases}, \text{ and}$$

$$r_3(\mathbf{s}) = \begin{cases} -a_2 & z < 0 \\ 0 & z \geq 0 \end{cases}$$

To obtain the state transition function samples $\mathbf{D}(\mathbf{s}_0, \mathbf{a}) = \mathbf{s}$, we rely on a simplified model of the quadrotor-load system, where the quadrotor is represented by a holonomic model of a UAV [64, 40, 89]. The simulator returns the next system state $\mathbf{s} = [\boldsymbol{p}^T\ \boldsymbol{v}^T\ \boldsymbol{\eta}^T\ \dot{\boldsymbol{\eta}}]^T$ when an action $\mathbf{a}$ is applied to a state $\mathbf{s}_0 = [\boldsymbol{p_0}^T\ \boldsymbol{v_0}^T\ \boldsymbol{\eta_0}^T\ \dot{\boldsymbol{\eta}}_0{}^T]$. Equations

$$
\begin{aligned}
&\boldsymbol{v} = \boldsymbol{v_0} + \Delta t \mathbf{a};\\
&\boldsymbol{p} = \boldsymbol{p_0} + \Delta t \boldsymbol{v_0} + \frac{\Delta t^2}{2}\mathbf{a}\\
&\dot{\boldsymbol{\eta}} = \boldsymbol{\eta}_0 + \Delta t \ddot{\boldsymbol{\eta}};\\
&\boldsymbol{\eta} = \boldsymbol{\eta_0} + \Delta t \dot{\boldsymbol{\eta}}_{\mathbf{0}} + \frac{\Delta t^2}{2}\ddot{\boldsymbol{\eta}},\ \textit{where}\\
&\ddot{\boldsymbol{\eta}} = \begin{bmatrix} \sin\omega_0 \sin\phi_0 & -\cos\phi_0 & L^{-1}\cos\omega_0 \sin\phi_0 \\ -\cos\omega_0 \cos\phi_0 & 0 & L^{-1}\cos\phi_0 \sin\omega_0 \end{bmatrix} (\mathbf{a} - \boldsymbol{g'})
\end{aligned}
\tag{4.1}
$$

describe the simulator. A vector $\boldsymbol{g'} = [0\ 0\ g]^T$ represents the gravity force vector, and Δt is the duration of the time step. L is the length of the suspension cable measured as distance between the cargo's and joint quadrotor-load system's centers of mass. Because the cargo's mass determines the center of the mass of the joint system, it influences the load's motion indirectly, through the effective cable length L.

The state value function V is approximated with a weighted feature vector $\boldsymbol{F}(\mathbf{s})$. The feature vector chosen for this problem consists of four basis functions, corresponding to features we wish to minimize for task completion. In our case, it consists of squares of the vehicles distance to the goal, its velocity magnitude, and load's angular displacement and velocity magnitude:

$$
V(\mathbf{s}) = \boldsymbol{\theta}^T \boldsymbol{F}(\mathbf{s}), \quad \boldsymbol{F}(\mathbf{s}) = [\|\boldsymbol{p}\|^2 \quad \|\boldsymbol{v}\|^2 \quad \|\boldsymbol{\eta}\|^2 \quad \|\dot{\boldsymbol{\eta}}\|^2]^T \tag{4.2}
$$

where $\boldsymbol{\theta} \in \mathbb{R}^4$.

4.2.2 Minimal Residual Oscillations Trajectory Generation

After learning the parametrization $\boldsymbol{\theta}$ of value function approximation, we can plan trajectories using greedy policy $\pi : S \rightarrow A$ induced by the approximated value function, V,

$$\pi(\mathbf{s}) = \underset{\mathbf{a} \in A}{\operatorname{argmin}}(\boldsymbol{\theta}^T \boldsymbol{F} \circ \mathbf{D}(\mathbf{s}, \mathbf{a})), \tag{4.3}$$

where $\mathbf{D}(\mathbf{s}, \mathbf{a})$ is the sampled state resulting from applying action $\mathbf{a}$ to a state $\mathbf{s}$[2]. When applied to the system, the resulting action moves the system to the state associated with the highest estimated value. The algorithm starts with an arbitrary initial state. Then it finds an action according to the greedy policy defined in Equation (4.3). The action is used to transition to the next state. The process repeats until the goal is reached or the trajectory exceeds a maximum number of steps.

The Proposition 4.2.1 gives sufficient conditions that the state-value function approximation, action state space and system dynamics need to meet to guarantee a plan that leads to the goal state.

Proposition 4.2.1. *Let $\boldsymbol{s_g}$ be the goal state. If:*

1. *All components of vector $\boldsymbol{\theta}$ are negative, $\theta_i < 0$, for $\forall i \in \{1, 2, 3, 4\}$,*
2. *Action space, A, allows transitions to a higher-valued state, $\forall \mathbf{s} \in S \setminus \{\boldsymbol{s_g}\}, \exists \mathbf{a} \in A$ that $V(\pi_A(\mathbf{s})) > V(\mathbf{s})$, and*
3. *$\boldsymbol{s_g}$ is global maximum of function V,*

[2] © 2013 IEEE. This section is reprinted, with permission, from Aleksandra Faust, Ivana Palunko, Patricio Cruz, Rafael Fierro, and Lydia Tapia, "Learning Swing-free Trajectories for UAVs with a Suspended Load," IEEE International Conference on Robotics and Automation (ICRA), May 2013

then the $\boldsymbol{s_g}$ is an asymptotically stable point. Coincidentally, for an arbitrary start state $\mathbf{s} \in S$, greedy policy, Equation (4.3)*, with respect to Vdefined in Equation* (4.2) *and A, leads to the goal state $\boldsymbol{s_g}$. In other words, $\forall \mathbf{s} \in S, \exists n, \pi^n(\mathbf{s}) = \boldsymbol{s_g}$.*

Proof. To show that s_q is an asymptotically stable point, we need to find a discrete time control Lyapunov function $W(\mathbf{s})$, such that

1. $W(\mathbf{s}(k)) > 0, \quad \text{for } \forall \mathbf{s}(k) \neq 0$
2. $W(\boldsymbol{s_g}) = 0,$
3. $\triangle W(\mathbf{s}(k)) = W(\mathbf{s}(k+1)) - W(\mathbf{s}(k)) < 0, \quad \forall k \geq 0$
4. $\triangle W(\mathbf{s}_g) = 0, \quad \text{where } \boldsymbol{s_g} = [0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0]^T$

Let $W(\mathbf{s}) = -V(\mathbf{s}) = -\boldsymbol{\theta}^T[\|\boldsymbol{p}\|^2 \|(\boldsymbol{v})\|^2 \|\boldsymbol{\eta}\|^2 \|\dot{\boldsymbol{\eta}}\|^2]$. Then $W(\mathbf{0}) = 0$, and for all $\mathbf{s} \neq \boldsymbol{s_g}$, $W(\mathbf{s}) > 0$, since $\theta_i < 0$.

$\triangle W(\mathbf{s}(k)) = -(V(\mathbf{s}(k+1)) - V(\mathbf{s}(k))) < 0$ because of the assumption that for each state there is an action that takes the system to a state with a higher value. Lastly, since $V(\boldsymbol{s_g})$ is global maxima, $W(\boldsymbol{s_g})$ is global minima, and $\triangle W(\boldsymbol{s_g}) = -(V(\boldsymbol{s_g}) - V(\boldsymbol{s_g})) = 0$

Thus, W is a control Lyapunov function with no constraints on $\mathbf{s}$, and is globally asymptotically stable. Therefore, any policy-following function W (or V) will lead the system to the unique equilibrium point. □

Proposition 4.2.1 connects the MDP problem formulation (states, actions, and transition function) and state-value approximation with Lyapunov stability analysis theory. If MDP and V satisfy the conditions, the system is globally uniformly stable, i.e. a policy generated under these conditions will drive the quadrotor-load system from any initial state $\mathbf{s}$ to the goal state $\boldsymbol{s_g} = \mathbf{0}$. We empirically show

that the conditions are met. Proposition 4.2.1 requires all components of vector $\boldsymbol{\theta}$ to be negative. As we will see in the 4.3.2, the empirical results show that is the case. These observations lead to several practical properties of the induced greedy policy that we will verify empirically:

- The policy is agnostic to the simulator used; the simulator defines the transition function, and along with the action space, defines the set of reachable states. Thus, as long as the conditions of Proposition 4.2.1 are met, we can switch the simulators we use. This means that we can train on a simple simulator and generate a trajectory on a more sophisticated model that would predict the system better.

- The policy can be learned on a state space subset that contains the goal state, and the resulting policy will work on the whole domain where the conditions above hold, i.e., where the value function doesn't have other maxima. We show this property in Section 4.3.2.

- The induced greedy policy is robust to noise; as long as there is a transition to a state with a higher value, the action will be taken and the goal will be attained. Section 4.3.2 presents the empirical evidence for this property.

- The action space between learning and the trajectory generation can change, and the algorithm will still produce a trajectory to the goal state. For example, to save computational time, we can learn on a smaller, more coarse discretization of the action space to obtain the value function parameters, and generate a trajectory on a more refined action space which produces a smoother trajectory. We demonstrate this property during the altitude changing flight experiment in Section 4.3.2. This property also allows us to create a path-following algorithm in Section 4.2.3 by restricting the action space only to actions that maintain proximity to the reference trajectory.

Since we use an approximation to represent a value function and obtain an estimate iteratively, the question of algorithm convergence is twofold. First, the parameters that determine the value function must converge to a fixed point. Second, the fixed point of the approximator must be close to the true value function. Convergence of the algorithm is not guaranteed in the general case. Thus, we show empirically that the approximator parameters stabilize. To show that the policy derived from a stabilized approximator is sound, we examine the resulting trajectory. The trajectory needs to be with minimal residual oscillations at the arrival at the goal state, and be suitable for the system.

Thus far, we used RL to learn the minimal residual oscillations task (Problem 4.1.4). The learning requires a feature vector, and a generative model of system dynamics, to come up with the feature vector parametrization. Then, in the distinct trajectory generation phase, using the same feature vector, and possibly different simulators, states, and action spaces, we create trajectories. The learning is done once for a particular payload, and the learned policy is used for any number of trajectories. Once the trajectory is created, it is passed to the lower-level vehicle controller.

4.2.3 Swing-free Path-following

To plan a path-following trajectory that reduces load's oscillations, we develop a novel approach that takes advantage of findings from Section 4.2.2 by applying constraints on the system. In particular, we rely on the ability to change the action space and still complete the task.

Let $\boldsymbol{P} = [\boldsymbol{r_1}, .., \boldsymbol{r_n}]^T$ be a reference path, given as the list of quadrotor's center of mass Cartesian coordinates in 3D space, and let $d_{\boldsymbol{P}}(\mathbf{s})$ be shortest Euclidean

distance between the reference path and the system's state $\mathbf{s} = [\boldsymbol{p}^T\ \boldsymbol{v}^T\ \boldsymbol{\eta}^T\ \dot{\boldsymbol{\eta}}^T]^T$:

$$d_{\boldsymbol{P}}(\mathbf{s}) = \min_i \|\boldsymbol{p} - \boldsymbol{r_i}\|. \tag{4.4}$$

Then we can formulate the Swing-free Path-following Task, as:

Problem 4.2.1. *(Swing-free Path-following Task): Given a reference path $\boldsymbol{P}$, and a start and a goal positional states $\mathbf{s}_s, \mathbf{s}_g \in \boldsymbol{P}$, find a minimum residual oscillations trajectory from $\mathbf{s}_s$ to $\mathbf{s}_g$ that minimizes the accumulated squared error $\int_0^\infty (d_{\boldsymbol{P}}(\mathbf{s}))^2\, dt$.*

To keep the system in the proximity of the reference path, we restrict the action space $A_\mathbf{s} \subset A$ to only the actions that transition the system in the proximity of the reference path, using proximity constant $\delta > 0$

$$A_\mathbf{s} = \{\mathbf{a} \in A | d_{\boldsymbol{P}}(D(\mathbf{s}, \mathbf{a})) < \delta\}. \tag{4.5}$$

In the case $A_\mathbf{s} = \emptyset$, we use an alternative action subset that transitions the system to the k closest position to the reference trajectory,

$$A_\mathbf{s} = \underset{\mathbf{a} \in A}{\text{argmin}^k}(d_{\boldsymbol{P}}(D(\mathbf{s}, \mathbf{a}))), \tag{4.6}$$

where argmin^k denotes k smallest elements. We call k the candidate action set size constant. Admissible action set $A_\mathbf{s} \subset A$ represents constraints on the system.

The action selection step, or the policy, becomes the search for action that transitions the system to the highest valued state chosen from the $A_\mathbf{s}$ subset,

$$\pi(\mathbf{s}) = \underset{\mathbf{a} \in A_\mathbf{s}}{\text{argmin}}(\boldsymbol{\theta}^T \boldsymbol{F} \circ \mathbf{D}(\mathbf{s}, \mathbf{a})). \tag{4.7}$$

The policy given in Equation (4.7) ensures state transitions in the vicinity of the reference path $\boldsymbol{P}$. If it is not possible to transition the system within given ideal proximity δ, the algorithm selects k closest positions and selects an action that produces the best minimal residual oscillations characteristics upon transition (see Algorithm 4.5). Varying δ, which is the desired error margin, controls how close we desire the system to be to the reference path. Parameter k gives the weight to whether we prefer the proximity to the reference trajectory, or load displacement reduction. Choosing very small k and δ results in trajectories that are as close to the reference trajectory as the system dynamics allows, at the expense of the load swing. Larger values of the parameters allow more deviation from the reference trajectory, and better load displacement results.

Algorithm 4.5 Swing-free Path-following.

Input: $\boldsymbol{s_0}$ start state, MDP (S, A, D, R) state space,

Input: $\boldsymbol{\theta}$, $\boldsymbol{F}(\mathbf{s})$,*max_steps*

Input: $\boldsymbol{P}$, δ, k

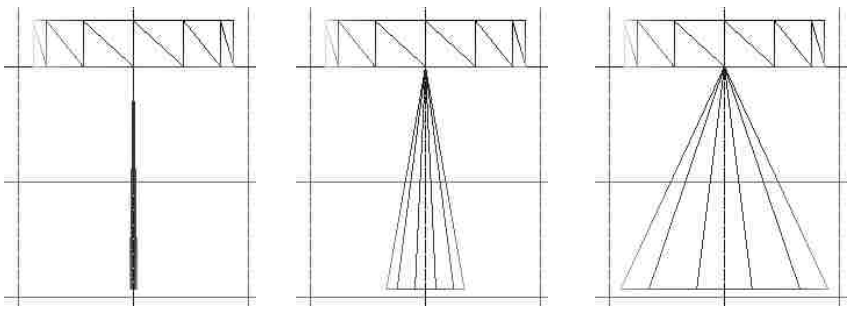
Output: *trajectory*

$\mathbf{s} \leftarrow \boldsymbol{s_0}$
trajectory $\leftarrow$ *empty*
while not(goal reached or *max_steps* reached) **do**
  $A_{\mathbf{s}} \leftarrow \{\mathbf{a} | d(\boldsymbol{s'}, \boldsymbol{P}) < \delta, \mathbf{s}' = D(\mathbf{s}, \mathbf{a}), \mathbf{a} \in A\}$
  if $A_{\mathbf{s}} == \emptyset$ **then**
    $A_{\mathbf{s}} \leftarrow \text{argmin}^{k}_{\mathbf{a} \in A}(d(D(\mathbf{s}, \mathbf{a}), P))$
  end if
  $\mathbf{a} \leftarrow \text{argmin}_{\mathbf{a} \in A_{\mathbf{s}}} \boldsymbol{\theta}^T \boldsymbol{F} \circ \mathbf{D}(\mathbf{s}, \mathbf{a}))$
  add $(\mathbf{s}, \mathbf{a})$ to *trajectory*
  $\mathbf{s} \leftarrow D(\mathbf{s}, \mathbf{a})$
end while
return *trajectory*

4.2.4 UAV Geometric Model

We use PRMs to obtain a collision-free path. The PRMs require a geometric model of the physical space, and of a robot, to construct an approximate model of C_{space}. In order to simplify the robot model, we use its bounding volume. The selection of the robot model has implications for the computational cost of the collision detection [69]. The bounding volume can be represented with fewer degrees of freedom by omitting certain joints, and to address local uncertainty of the configuration space [66].

For the Cargo Delivery Task that bounds the load displacement with a given fixed limit, we consider the quadrotor-load system to be two rigid bodies joined with a ball joint and an inelastic suspension cable. This system is geometrically contained in a joint cylinder-cone volume. Modeling the quadrotor's body with a cylinder that encompasses it, allows us to ignore yaw for the path planning purposes. Similarly, Cargo Delivery Task is a non-aggressive maneuver, and quadrotor's pitch and roll are negligible, and are contained in a cylinder. For the load, we need only to consider the set of its possible positions, which is contained in a right circular cone with an apex that connects to the quadrotor's body, and with its axis perpendicular to quadrotor's body. The relationship between the cone's aperture, ρ, and load displacement $\boldsymbol{\eta} = [\phi \ \omega]^T$ that need to be satisfied for collision-free trajectories is $\cos(\rho/2) > (1 + \tan^2\phi + \tan^2\omega)^{-0.5}$. Connecting the cylindrical model of the quadrotor body with the cone model of its load gives us the geometrical model of the quadrotor carrying a suspended load. Since the quadrotor's body stays orthogonal to the z-axis, the cylinder-cone model can be treated as a single rigid body with 3 degrees of freedom. Figure 4.1c shows the fit of the AscTec Hummingbird quadrotor carrying a suspended load fitting into the cylinder-cone bounding volume. The clearance surrounding the quadrotor's body area needs greater than the maximal path-following error, $\max_t d_{\boldsymbol{P}}(\mathbf{s}(t))$.



(a) 1° task (b) 10° task (c) 20° task

Figure 4.3: Orthographic wireframe of the quadrotor carrying a suspended load geometric model for three different tasks.

The model, will produce paths that leave enough clearance between the obstacles to accommodate for the maximum allowable load displacement.

Approaching the problem in this manner, allows PRM to perform fast collision-checking without burdening path planning with the knowledge of complicated system dynamics. Because the bounding volumes for tasks with smaller bounds on the load displacement are contained within volumes for tasks that allow greater swing, a more conservative model can be used for multitude of tasks having lesser upper bounds (see Figure 4.3). The limitation of the reduced model is that it does not take a direction of the load displacement into account. This could be easily remedied with replacing a circular cone with elliptic one for example, and then pairing with a compatible RL agent that ensures the trajectory adheres to the geometric model.

4.2.5 Path Planning and Trajectory Generation Integration

Figure 7.1a shows the flowchart of the trajectory generation process. This method learns the dynamics of the system through AVI, as outlined in Section 4.2.1. Monte Carlo selection picks the best policy out of the multiple learning trials. Indepen-

dently and concurrently, a planner learns the space geometry. When the system is queried with particular start and stop goals, the planner returns a collision-free path. The PRM egdes are reference paths, and the nodes are waypoints. AVI provides the policy. The trajectory generation module generates a Minimal Residual Oscillations trajectory along each edge in the path, using Algorithm 4.5 and the modified AVI policy. If the trajectory exceeds the maximum allowed load displacement, the edge is bisected, and the midpoint is inserted in the waypoint list (see Algorithm 4.6). The system comes to a stop at each waypoint and starts the next trajectory from the stopped state. All swing-free trajectories are translated and concatenated to produce the final multi-waypoint, collision-free trajectory. This trajectory is sent to the low-level quadrotor controller [99] that performs trajectory tracking. The result is fully automated method that solves Cargo Delivery Task (Problem 4.1.5).

Line 12 in Algorithm 4.6 creates a trajectory segment that follows the PRM calculated path between adjacent nodes (w_i, w_{i+1}). The local planer determines the path geometry. Although any local planner can be used with PRMs, in this setup we choose a straight line local planner to construct the roadmap. Thus, the collision-free paths are line segments. Prior to creating a segment, we translate the coordinate system such that the goal state is in the origin. Upon trajectory segment calculation, the segment is translated so that it ends in w_{i+1}. The Swing-free Path-following corresponds to Algorithm 4.5. When the reference path is a line segment with start in $\mathbf{s}_0$ and end in the origin, such as in this setup, the distance d calculation defined in Equation (4.4), and needed to calculate the action space subsets defined in Equations (4.5) and (4.6), can be simplified and depends only the start state $\mathbf{s}_0$:

$$d_{\mathbf{s}_0}(\mathbf{s}) = \sqrt{\frac{\|\mathbf{s}\|^2\|\mathbf{s}_0\|^2 - (\mathbf{s}^T\mathbf{s}_0)^2}{\|\mathbf{s}_0\|}}.$$

Algorithm 4.6 leads the system to the goal state. This is an implication of the asymptotic stability of the AVI algorithm. It means that the policy produces trajectories that, starting in any initial state $\boldsymbol{s_0}$, come to rest at the origin. Because Algorithm 4.6 translates the coordinate system such that the next waypoint is always in the origin, the system passes through all waypoints until it reaches the end of the path.

4.3 Results

Table 4.1: Approximate value iteration hyper-parameters.

Parameter	3D Configuration	2D Configuration
γ	0.9	
Min action ($\mathrm{m\,s^{-2}}$)	(-3, -3, -3)$\mathrm{m\,s^{-2}}$	(-3, -3, 0)
Action step ($\mathrm{m\,s^{-2}}$)	0.5	0.05
Min sampling space	$\boldsymbol{p} = (-1, -1, -1)\mathrm{m}$, $\boldsymbol{v} = (-3\,\mathrm{m\,s^{-1}}, -3\,\mathrm{m\,s^{-1}}, -3\,\mathrm{m\,s^{-1}})$ $\boldsymbol{\eta} = (-10°, -10°)$, $\dot{\boldsymbol{\eta}} = (-10, -10)$	
Max sampling space	$\boldsymbol{p} = (1\,\mathrm{m}, 1\,\mathrm{m}, 1\,\mathrm{m})$ $\boldsymbol{v} = (3\,\mathrm{m\,s^{-1}}, 3\,\mathrm{m\,s^{-1}}, 3\,\mathrm{m\,s^{-1}})$ $\boldsymbol{\eta} = (10°, 10°)$, $\dot{\boldsymbol{\eta}} = (10, 10)$	
Sampling	Linear	Constant (200)
Simulator	Deterministic	
Frequency	50 Hz	
Number of iterations	1000	800
Number of trials	100	40
Reward function	$c_1 = 10000$, $c2 = 750$, $c_3 = 1$ $a_1 = 14$, $a_2 = 10000$, $\epsilon = 0.05$	

To evaluate the Aerial Cargo Delivery software architecture, we first check each of the components separately, and then evaluate the architecture as a whole. Section 4.3.1 validates the learning and its convergence to a single policy. Section 4.3.2 evaluates policy for minimal residual oscillation trajectory generation and its performance under varying state spaces, action spaces, and system simulators, as

Proposition 4.2.1 predicts. In Section 4.3.3, we address the quality of the Swing-free Path-following algorithm. After results for all components are evaluated, we verify the end-to-end process in Section 4.3.4.

All of the computations were performed on a single core of an Intel i7 system with 8 GB of RAM, running Linux operating system. AVI and trajectory calculations were obtained with Matlab 2011. The experiments are performed using an AscTec quadrotor UAV carrying a small ball or a paper cup filled with water in a MARHES multi-aerial vehicle testbed [82]. This testbed and its real-time controller are described in detail in [99]. The testbed's real-time controller tracks the planned trajectory and controls the quadrotor. The testbed is equipped with a Vicon high-precision motion capture system to collect the results of the vehicle and the load positioning during the experiments with the resolution of 1 mm at minimum 100 Hz. The quadrotor is 36.5 cm in diameter, weighs 353 g without a battery, and its payload is up to 350 g [15]. Meanwhile, the ball payload used in the experiments weighs 47 g and its suspension link length is 0.62 m. The suspended load is attached to the quadrotor at all times during the experiments. In experiments where the coffee cup is used, the weight of the payload is 100 g.

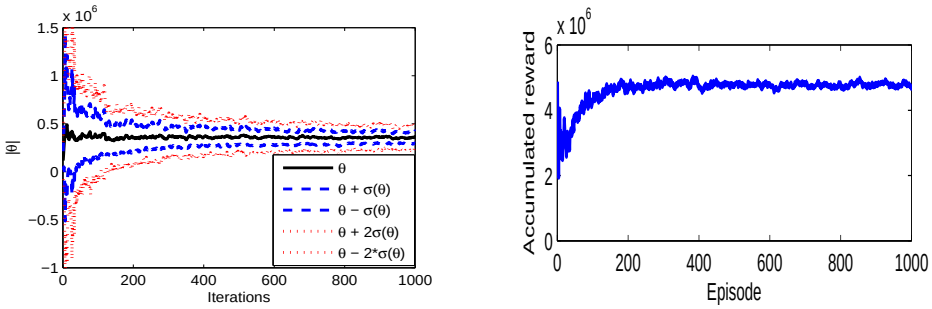
4.3.1 Learning Minimal Residual Oscillations Policy

To empirically verify the learning convergence,[3] we run AVI in two configurations: 2D and 3D (see Table 4.1). Both configurations use the same discount parameter $\gamma < 1$ to ensure that the value function is finite and are learning in the sampling box 1 m around the goal state with the load displacements under 10°.

[3] © 2013 IEEE. This section is reprinted, with permission, from Aleksandra Faust, Ivana Palunko, Patricio Cruz, Rafael Fierro, and Lydia Tapia, "Learning Swing-free Trajectories for UAVs with a Suspended Load," IEEE International Conference on Robotics and Automation (ICRA), May 2013

The configurations also share the deterministic simulator, Equation (4.1). The 3D configuration trains the agent with a coarse three-dimensional action vector. Each direction of the linear acceleration is discretized in 13 steps, resulting in over 2000 total actions. In this phase of the algorithm, we are shaping the value function, and this level of coarseness is sufficient and practical. The most computationally intensive part of the learning is greedy policy, Equation (4.3). Performed for each sample in all iterations, the greedy policy evaluation scales with the number of actions. The learning transfer that learns with a coarse and plans with fine-grain action space, speeds up learning 1000 times, thus making it practically feasible. AVI's approximation error decays exponentially with the number of iterations. A gradual increase in the sampling over iterations yields less error as the number of iterations increases [37]. Thus, we increase sampling linearly with the number of iterations in the 3D configuration.

To assess the stability of the approximate value iteration, we ran the AVI 100 times, for 1000 iterations in the 3D configuration. Figure 4.4a shows the trend of the norm of value parameter vector $\boldsymbol{\theta}$ with respect to L_2 norm. We can see that the $\|\boldsymbol{\theta}\|$ stabilizes after about 200 iterations with the mean of 361170. The empirical results show that the algorithm is stable and produces a consistent policy over different trials. The mean value of $\boldsymbol{\theta} = [-86290\ -350350\ -1430\ -1160]^T$ has all negative components, which means that the assumption for Proposition 4.2.1 holds. To evaluate learning progress, a trajectory generation episode was run after every learning iteration. Figure 5.4 depicts the accumulated reward per episode, averaged over 100 trials. The accumulated reward converges after 200 iterations as well. Figure 4.5 depicts trajectories with the quadrotor's starting position at $(-2, -2, 1)$m over 100 trials after 1000 learning iterations. The trajectories are created as described in Section 4.2.2 with Equation (4.1) simulating the movement at 50 Hz. Although there are slight variations in duration (see Figure 4.5a), all the trajectories are similar in shape and are consistent, giving us confidence that the



(a) $\boldsymbol{\theta}$ convergence (b) Learning curve

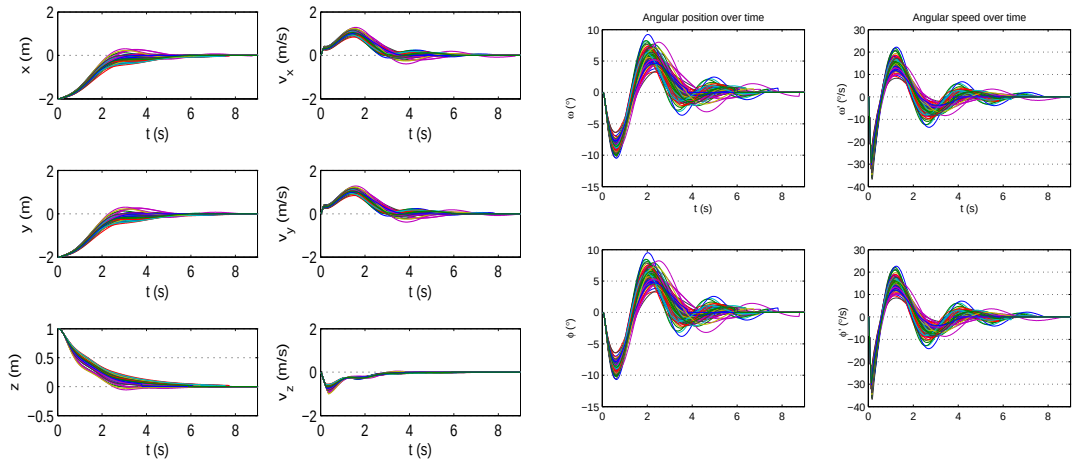
Figure 4.4: Convergence of feature parameter vector $\boldsymbol{\theta}$'s norm (a) and corresponding learning curve (b) over 1000 iterations. The results are averaged over 100 trials. One and two standard deviations are shown. After initial learning phase, $\|\boldsymbol{\theta}\|$ stabilizes to a constant value.

AVI converges. The load initially lags behind as the vehicle accelerates (see Figure 4.5b), but then stabilizes to end in minimal swing. We can also see that the swing is bounded throughout the trajectory, maintaining the displacement under 10° for the duration of the entire flight (see Figure 4.5b).

4.3.2 Minimal Residual Oscillations Trajectory Generation

In this section[4] we evaluate effectiveness of the learned policy. We show the policy's viability in the expanded state and action spaces in simulation in Section 4.3.2. Section 4.3.2 assesses the discrepancy between the load displacement predictions in simulation and encountered experimentally during the flight. The section also compares experimentally the trajectory created using the learned policy to two other methods: a cubic spline trajectory, which is a minimum time C^3-class trajectory without any pre-assumptions about the load swing, and, to a dynamic

[4]© 2013 IEEE. This section is reprinted, with permission, from Aleksandra Faust, Ivana Palunko, Patricio Cruz, Rafael Fierro, and Lydia Tapia, "Learning Swing-free Trajectories for UAVs with a Suspended Load," IEEE International Conference on Robotics and Automation (ICRA), May 2013



(a) Quadrotor position and velocity (b) Load position and velocity

Figure 4.5: Resulting trajectories from 100 policy trials. Trajectories starting at (-2, -2, 1) m for each of the 100 trials of the vehicle (a), and its load (b) using 3D configuration for training and deterministic simulator with fine-grain action space for trajectory generation.

programming trajectory [99], an optimal trajectory for a fixed start position with respect to its MDP setup.

State and action space expansion We evaluate the quality and robustness of a trained agent in simulation by generating trajectories from different distances for two different simulators. The first simulator is a *deterministic* simulator described in Equation (4.1) and used in the learning phase. The second simulator is a *stochastic* simulator that adds up to 5 % uniform noise to the state predicted with the deterministic simulator. Its intent is to simulate the inaccuracies and uncertainties of the physical hardware motion between two time steps. We compare the performance of our learned, generated trajectories with model-based dynamic programming and cubic trajectories. The cubic and dynamic programming trajectories are generated using methods described in [99], but instead of relying on the full quadrotor-load system, we use the simplified model given by Equation (4.1). The cubic and dynamic programming trajectories are of the same

duration as corresponding learned trajectories. The agent is trained in 3D configuration. For trajectory generation, we use a fine-grain, discretized 3D action space $A = (-3 : 0.05 : 3)^3$. This action space is ten times per dimension finer, and contains over 10^6 different actions. The trajectories are generated at 50 Hz with a maximum trajectory duration of 15 s. All trajectories were generated and averaged over 100 trials. To assess how well a policy adapts to different starting positions, we choose two different fixed positions, (-2,-2, 1) m and (-20,-20,15) m, and two variable positions. The variable positions are randomly drawn from between 4 m and 5 m, and within 1 m from the goal state. The last position measures how well the agent performs within the sampling box. The rest of the positions are well outside of the sampling space used to learn the policy, and evaluate generalization to an extended state space.

Table 4.2 presents the averaged results with their standard deviations. We measure the end state and the time when the agent reaches the goal, the percentage of trajectories that reach the goal state within 15 s, and the average maximum swing experienced among all 100 trials. With the exception of the stochastic simulator at the starting position (-20,-20,15) m, all experiments complete the trajectory within 4 cm of the goal, and with a swing of less than 0.6°, as Proposition 4.2.1 predicts. The trajectories using the stochastic simulator from a distance of 32 m (-20,-20,15) don't reach within 5 cm because 11 % of the trajectories exceed the 15 s time limit before the agent reaches its destination. However, we still see that the swing is reduced and minimal at the destination approach, even in that case. The results show that trajectories generated with *stochastic* simulator on average take 5 % longer to reach the goal state, and the standard deviations associated with the results is larger. This is expected, given the random nature of the noise. However, all of the stochastic trajectories approach the goal with about the same accuracy as the deterministic trajectories. This finding matches our prediction from Section 4.2.2.

The maximum angle of the load during its entire trajectory for all 100 trials depends on the inverse distance from the initial state to the goal state. For short trajectories within the learning sampling box, the swing always remains within 4°, while for very long trajectories it could go up to 46°. As seen in Figure 4.5, the peak angle is reached at the beginning of the trajectory during the initial acceleration, and as the trajectory proceeds, the swing reduces. This makes sense, given that the agent is minimizing the combination of the swing and distance. When very far away from the goal, the agent moves quickly toward the goal state and produces increased swing. Once the agent is closer to the goal state, the swing component becomes dominant in the value function, and the swing reduces.

Figure 4.6 shows the comparison of the trajectories with the same starting position (-2, -2, 1) m and the same $\boldsymbol{\theta}$ parameter, generated using the models above (AVI trajectories) compared to cubic and dynamic programming trajectories. First, we see that the AVI trajectories share a similar velocity profile (Figure 4.6a) with two velocity peaks, both occurring in the first half of the flight. Velocities in dynamic programming and cubic trajectories have a single maximum in the second half of the trajectory. The resulting swing predictions (Figure 4.6b) shows that in the last 0.3 s of the trajectory, the cubic trajectory exhibits a swing of 10°, while the dynamic programming trajectory ends with a swing of less than 5°. The AVI generated trajectories produce load displacement within 2° in the same time period. To assess energy of the load's motion and compare different trajectories in that way, Figure 4.3.2 shows one-sided power spectral density (PSD) of the load displacement angles. PSD, a signal processing tool, calculates amount of energy per frequency in a time series. Smaller energy per frequency value and narrower frequency range correspond to less load displacement. The area below the curve is total energy needed to produce the motion of the load. We calculated the PSD over the entire load trajectory signal, using Matlab's *periodogram* method from the Signal Processing Toolbox. We see that the energy per

frequency of the cubic trajectory is above the other three trajectories. Inspecting the average energy of AVI deterministic ($E([\phi\ \omega]) = [0.0074\ 0.0073]$), stochastic AVI ($E([\phi\ \omega]) = [0.0050\ 0.0050]$), and dynamic programming trajectories ($E([\phi\ \omega]) = [0.0081\ 0.0081)$ load position signals, we find that AVI deterministic trajectory requires the least energy over the entire trajectory.

Experimental results As another approach to addressing the learning practicality, we first train the agent in fine-grain 2D configuration. The configuration uses a $0.05\,\mathrm{m\,s^{-2}}$ action tile size, although only in the x and y directions. There are 121 actions in each direction, totaling to 121^2 actions in the discretized space. Note that this action space, although it has over 10^4 actions, is still two orders of magnitude smaller that the fine-grain 3D action space used for the planning with over 10^6 actions. This configuration uses a fixed sampling methodology. The approximation error stabilizes to a roughly constant level after the parameters stabilize [37]. Once the agent is trained, we generate trajectories with the same planning parameters as in Section 4.3.2, for two experiments: constant altitude flight and flight with changing altitude. The planned trajectories are sent the physical quadrotor in the testbed.

In the constant altitude flight, the quadrotor flew from (-1,-1,1) m to (1,1,1) m. Figure 4.7 compares the vehicle and load trajectories for the learned trajectory as flown and in simulation, with cubic and dynamic programming trajectories of the same length and duration. The vehicle trajectories in Figure 4.7a suggest a difference in the velocity profile, with the learned trajectory producing a slightly steeper acceleration between 1 s and 2.5 s. The learned trajectory also contains a 10 cm vertical move up toward the end of the flight. To compare the flown trajectory with the simulated trajectory, we look at the load trajectories in Figure 4.7b. We notice the reduced swing, especially in the second half of the load's ϕ coordinate. The trajectory in simulation never exceeds 10°, and the actual flown

trajectory reaches its maximum at 12°. Both learned load trajectories follow the same profile with three distinct peaks around 0.5 s, 2.2 s and 2.2 s into the flight, followed by rapid swing control and reduction to under 5°. The actual flown trajectory naturally contains more oscillations that the simulator didn't model. Despite that, the limits, boundaries, and profiles of the load trajectories are close between the simulation and flown trajectories. This verifies the validity of the simulation results: the load trajectory predictions in the simulator are reasonably accurate. Comparing the flown learned trajectory with a cubic trajectory, we see a different swing profile. The cubic load trajectory has higher oscillation, four peaks within 3.5 s of flight, compared to three peaks for the learned trajectory. The maximum peak of the cubic trajectory is 14° at the beginning of the flight. The most notable difference happens after the destination is reached during the hover (after 3.5 s in Figure 4.7b). In this part of the trajectory, the cubic trajectory shows a load swing of 5°-12°, while the learned trajectory controls the swing to under 4°. Figure 4.7b shows that the load of the trajectory learned with RL stays within the load trajectory generated using dynamic programming at all times: during the flight (the first 3.4 s) and the residual oscillation after the flight. PSD in Figure 4.7c shows that, during experiments, the learned trajectory uses less energy per frequency than cubic and dynamic programming trajectories.

In the second set of experiments, the same agent was used to generate changing altitude trajectories that demonstrate ability to expand action space between. Note that the trajectories generated for this experiment used value approximator parameters learned on a 2D action space, in the xy-plane, and produced a viable trajectory that changes altitude because the trajectory generation phase used 3D action space. This property was predicted by Proposition 4.2.1 since the extended 3D action space allows transitions to the higher value states. The experiment was performed three times and the resulting quadrotor and load trajectories are depicted in Figure 4.8. The trajectories are consistent between three trials and follow

closely simulated trajectory (Figure 4.8a). The load displacement (Figure 4.8b) remains under 10° and exhibits minimal residual oscillations. Figure 4.8c shows that the energy profile between the trials remains consistent.

This section evaluated in detail AVI trajectories both in simulation and experimentally. The evaluations show that the method is robust to motion noise, can learn in small and plan in large spaces, and the simulation predictions are within 5° from the experimental observations.

Algorithm 4.6 Collision-free Cargo Delivery Trajectory Generation.

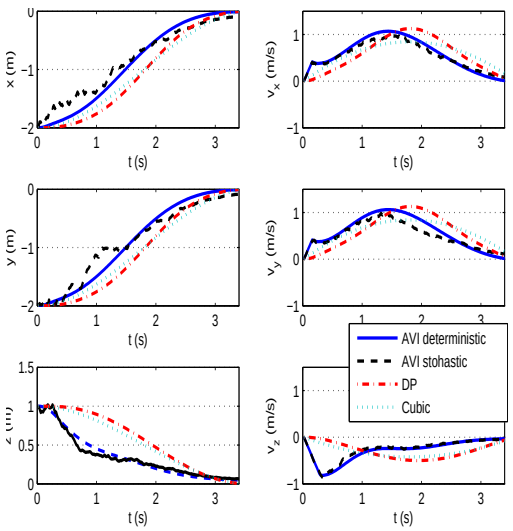
Input: *start*, *goal*,

Input: *space_geometry*, *robot_model*,

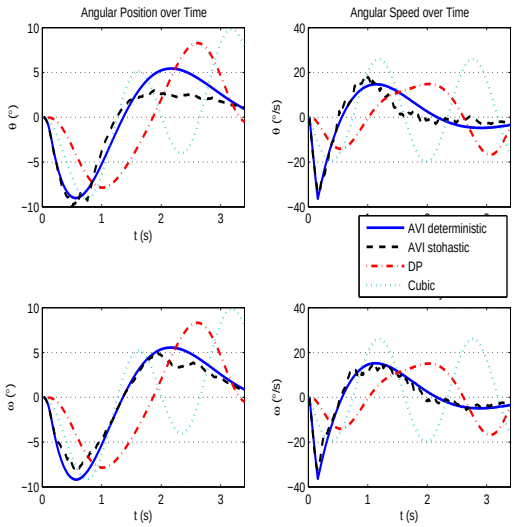
Input: *dynamics_generative_model*, *max_swing*

Output: *trajectory*

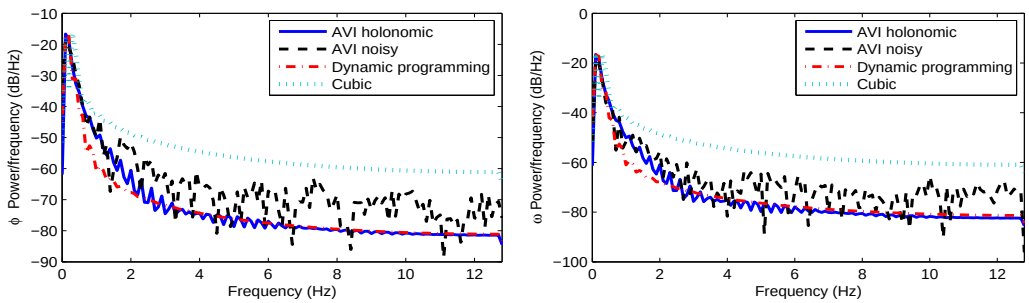
if *prm* not created **then**
  $prm.create(space\_geometry, robot\_model)$
end if
if *policy* not created **then**
  $policy \leftarrow avi.learn(dynamics\_generative\_model)$
end if
$trajectory \leftarrow []$
$path \leftarrow prm.query(start, goal)$
$currentStart \leftarrow path.pop()$
while not path.empty() **do**
  $distance \leftarrow currentStart - currentGoal$
  $tSegmeent \leftarrow avi.executeTrack(distance, policy)$
  if $tSegmeent.maxSwing > max\_swing$ **then**
    $midpoint \leftarrow \frac{currentStart+currentGoal}{2}$
    $path.push(midpoint)$
    $currentGoal \leftarrow midpoint$
  else
    $tSegment \leftarrow translate(tSegment, currentGoal)$
    $trajectory \leftarrow trajectory + tSegment$
    $currentStart \leftarrow currentGoal$
    $currentGoal \leftarrow path.pop()$
  end if
end while
return *trajectory*



(a) Quadrotor position and velocity



(b) Load position and velocity

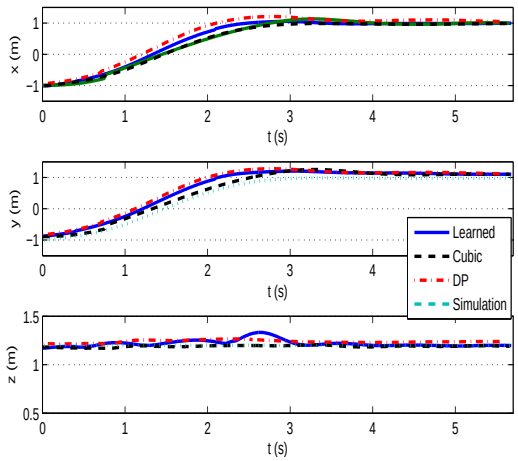


(c) Power spectral density

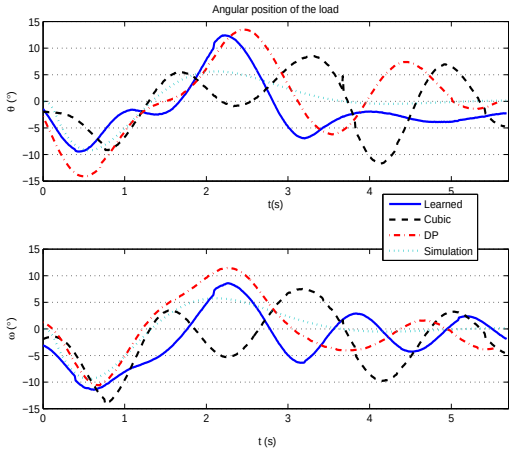
Figure 4.6: AVI trajectory comparison to DP and cubic trajectories in simulation. Trajectories of the (a) vehicle, (b) its load , and (c) load displacement's power spectral density where the training was performed in 3D configuration, and the trajectories were generated using generic and stochastic simulators compared to the cubic and dynamic programming trajectories of the same duration.

Table 4.2: Summary of AVI trajectory results for different starting position averaged over 100 trials: percent completed trajectories within 15 s, time to reach the goal, final distance to goal, final swing, and maximum swing.

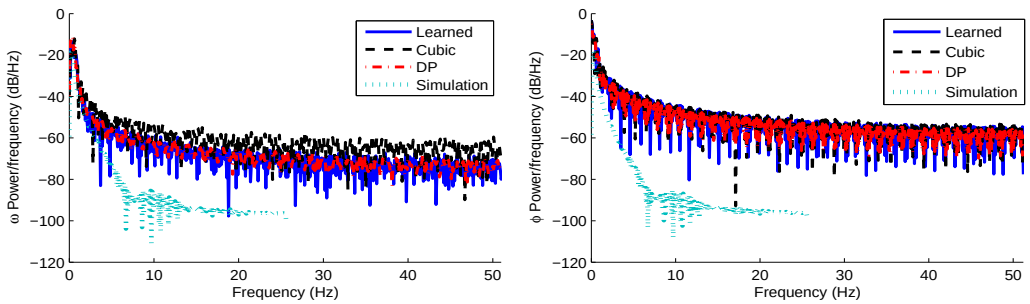
State		Goal reached	t (s)		$\| \boldsymbol{p} \|$ (m)		$\| \boldsymbol{\eta} \|$ $(^\circ)$		max $\| \boldsymbol{\eta} \|$ $(^\circ)$	
Location	Simulator	(%)	μ	σ	μ	σ	μ	σ	μ	σ
(-2,-2,1)	Deterministic	100	6.13	0.82	0.03	0.01	0.54	0.28	12.19	1.16
	Stochastic	100	6.39	0.98	0.04	0.01	0.55	0.30	12.66	1.89
(-20,-20,15)	Deterministic	99	10.94	1.15	0.04	0.01	0.49	0.33	46.28	3.90
	Stochastic	89	12.04	1.91	0.08	0.22	0.47	0.45	44.39	7.22
((4,5),(4,5),(4,5))	Deterministic	100	7.89	0.87	0.04	0.01	0.36	0.31	26.51	2.84
	Stochastic	100	7.96	1.11	0.04	0.01	0.44	0.29	27.70	3.94
((-1,1),(-1,1),(-1,1))	Deterministic	100	4.55	0.89	0.04	0.01	0.33	0.30	3.36	1.39
	Stochastic	100	4.55	1.03	0.04	0.01	0.38	0.29	3.46	1.52



(a) Quadrotor position

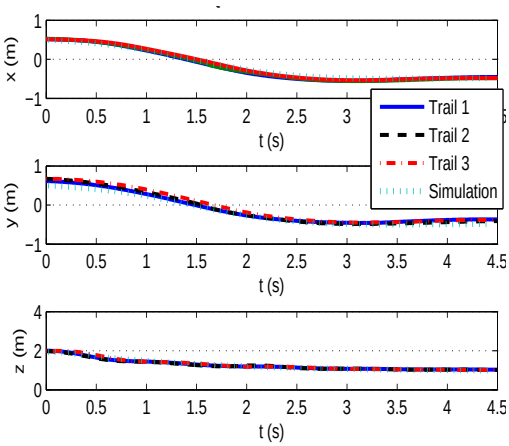


(b) Load position

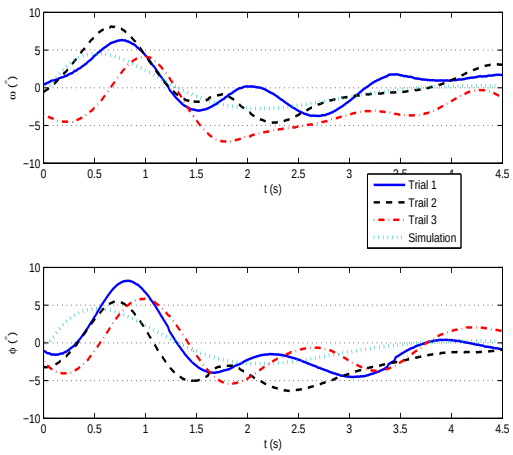


(c) Power spectral density

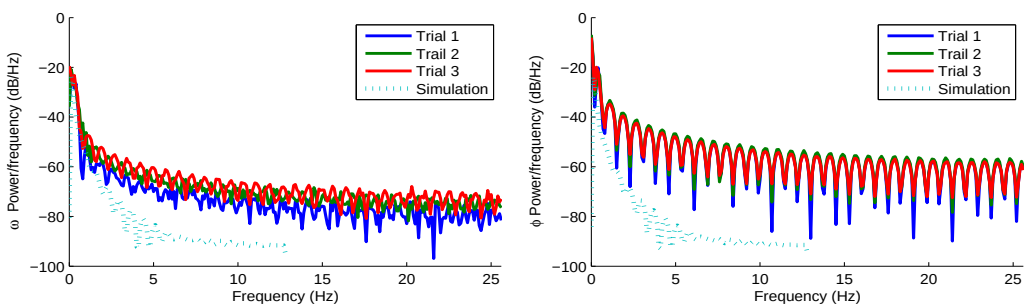
Figure 4.7: AVI experimental trajectories. Quadrotor (a), load (b) trajectories, and power spectral density (c) as flown, created through learning compared to an experimental cubic, experimental DP, and simulated AVI trajectories.



(a) Quadrotor position



(b) Load position



(c) Power spectral density

Figure 4.8: AVI experimental results of altitude changing flight. Quadrotor (a), load (b) trajectories, and power spectral density (c) as flown and in simulation, over three trials in the altitude changing test trained in planar action space.

4.3.3 Swing-free Path-following

Path-following with reduced load displacement evaluation compares the load displacement and path-following errors of the Algorithm 4.5 with two other methods: a minimal residual oscillations method described in Section 4.2.2, and path-following with no load displacement reduction. The path-following only method creates minimum time path-following trajectories, by choosing actions that transition the system as close as possible to the reference path in the direction of the goal state with no consideration to the load swing. We use three reference trajectories: a straight line, a two-line segment, and a helix. To generate a trajectory, we use action space discretized in $0.1\,\mathrm{m\,s^{-2}}$ equidistant steps, and the same value function parametrization, $\boldsymbol{\theta}$, used in the evaluations in Section 4.3.2 and learned in Section 4.2.1.

Table 4.3 examines the role of the proximity parameter δ, and candidate actions set size parameter k. Recall that the Algorithm 4.5 uses δ as a distance from the reference trajectory where all actions that transition the system within δ distance are be considered for load swing reduction. If there were no such actions, then the algorithm selects k actions that transition the system the closest to the reference trajectory, regardless of the actual physical distance. We look at two proximity factors and two action set size parameters. In all cases, the proposed method, Swing-free Path-following, exhibits smaller path-following error than minimal residual oscillations method, and smaller load displacement results than path-following only method. Also for each reference path, there is a set of parameters that provides good balance between path-following error and load displacement reduction.

General trend is that the load displacement decreases and tracking error increases with the increase of k and δ. This is expected because the larger the two parameters are, the larger is the action set from which to choose action for load

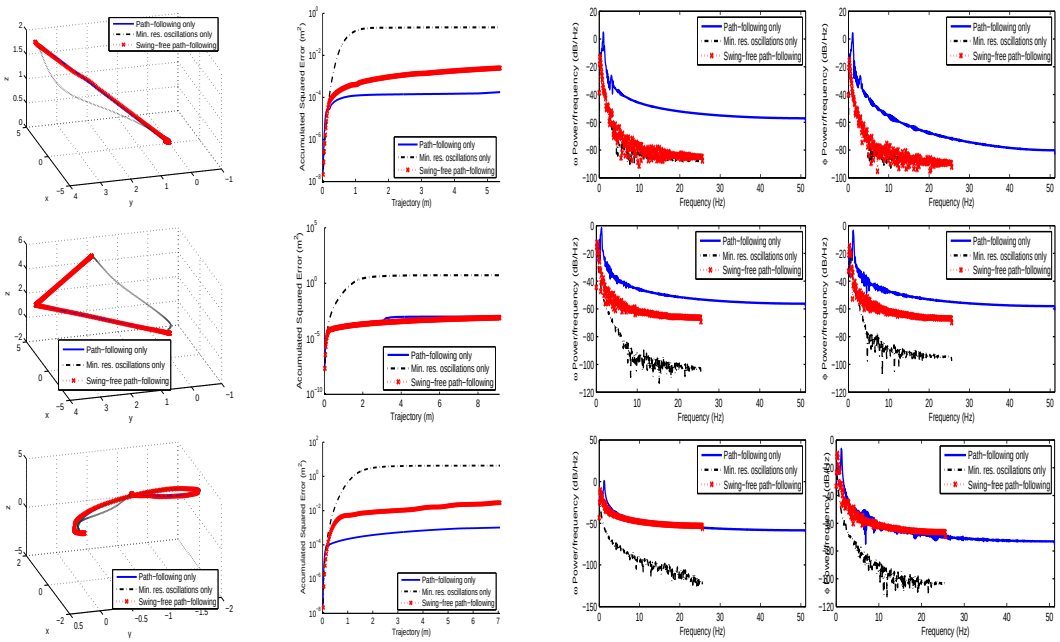
Table 4.3: Summary of path-following results for different trajectory geometries: reference path, k, proximity factor δ, trajectory duration, maximum swing, and maximum deviation from the reference path (error). Best results for reference path are highlighted.

Ref. path	k	δ (m)	t (s)	$\|\boldsymbol{\eta}\|$ (°)	Error (m)
Line	**100**	**0.01**	**11.02**	**21.94**	**0.02**
	500	0.01	11.02	20.30	0.03
	100	0.05	6.64	23.21	0.08
	500	0.05	7.06	23.22	0.11
Path-following only	100	0.01	11.02	73.54	0.01
Minimal residual oscillations	-	-	6.74	18.20	0.49
Multi-segment line	**100**	**0.01**	**7.52**	**20.17**	**0.04**
	500	0.01	7.52	23.11	0.50
	100	0.05	7.52	28.24	0.10
	500	0.05	7.52	26.70	0.88
Path-following only	100	0.01	11.02	44.76	0.04
Minimal residual oscillations	-	-	6.74	16.15	2.19
Helix	100	0.01	5.72	29.86	0.39
	500	0.01	5.72	26.11	0.44
	100	0.05	5.76	32.13	0.17
	500	**0.05**	**5.72**	**22.20**	**0.11**
Path-following only	100	0.01	11.02	46.01	0.02
Minimal residual oscillations	-	-	7.68	4.30	1.80

control, thus the system is more likely to select an action that transitions system further from the reference trajectory, in order to either move closer to the goal and control the load displacement. However, there are several exceptions to the general trend. For helix trajectory, $k = 500$ and $\delta = 0.05$ yield best results across all other parameter values. This is because of the constant curvature of helix requires to take into consideration states are a bit farther from the reference trajectory. The second exception is the line trajectory with $\delta = 0.01$ and $k = 500$; it has better load displacement control than when k. The difference happens at the beginning of the trajectory where value function, defined in Equation (4.2), favors arriving to the goal over controlling the load displacement. With the broader action set to choose from, the case when k is larger arrives to destination faster (6.94 s versus 11.02 s), with a bit more load displacement and deviation from the reference trajectory.

The last exceptions occur during the multi-segment test; when $k = 500$ the system experiences significant tracking error and slightly worse load displacement compared to larger k. This is because of the sudden turn in direction. With m sufficiently small (100) the system never veers far off the trajectory, thus the tracking error remains small. Last, in all cases the tracking with load displacement displays better tracking error than load displacement only method, and better load displacement results that tracking only method.

Figure 4.9 presents results of the path-following and load displacement errors for the line and helix reference paths. We compare them to path-following only and minimal residual oscillations AVI algorithms. Figure 4.9a displays the trajectories. In all three cases, the minimal residual oscillations algorithm took a significantly different path from the other two. Its goal was to minimize the distance to the goal state as soon as possible while reducing the swing, and it does not follow the reference path. Figure 4.9b quantifies the accumulated path-following error. Path-following error for Algorithm 4.5 remains close to the path-following only trajectory. For both trajectories, the accumulated path-following error is one to two orders of magnitude smaller than for the minimal residual oscillations method. Examining the load displacement characteristics though power spectral analysis of the vector $\boldsymbol{\eta} = [\phi \ \omega]^T$ time series, we notice that frequency profile of the trajectory generated with Algorithm 4.5 resembles closely that of the minimal residual oscillations trajectory (Figure 4.9c). In contrast, power spectral density of path-following only trajectories contain high frequencies absent in the trajectories created with the other two methods. Thus, the proposed method for Swing-free Path-following, with its path-following error characteristics similar to the path-following only method, and its load displacement characteristics similar to the minimal residual oscillations method, offers a solid compromise between the two extremes.



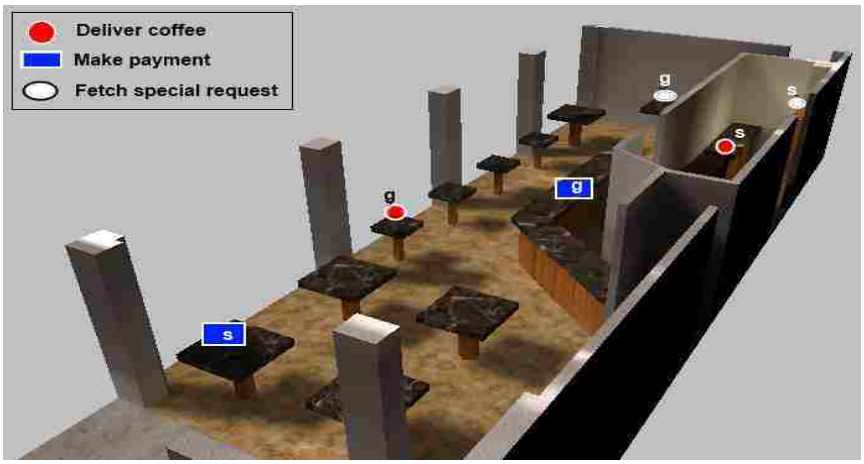
(a) Trajectory (b) Line path-following error (c) Power spectral density

Figure 4.9: Swing-free Path-following for line, multi-segment, and helix reference trajectories, compared to path-following only, and to load displacement control only (a). Path-following error (b) is in logarithmic scale, and power spectral density (c).

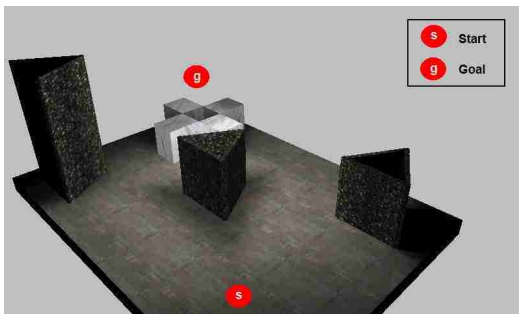
4.3.4 Automated Aerial Cargo Delivery

In this section we evaluate the methods for an Cargo Delivery Task developed in Section 4.2.5. Our goal is to verify that the method finds collision-free paths and creates trajectories that closely follow the paths while not exceeding the given maximal load displacement. The method's performance in simulation is discussed in Section 4.3.4, and its experimental evaluation, in Section 4.3.4.

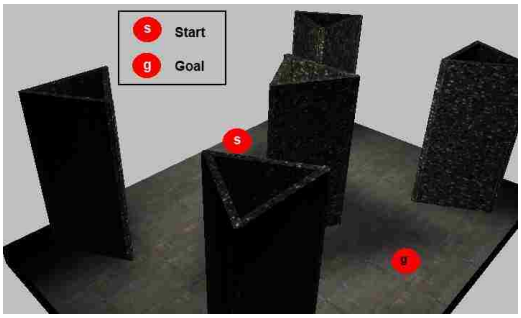
The simulations and experiments were performed using the same setup as for minimal residual oscillations tasks in obstacle-free environments, described in Section 4.3.2. PRMs work by first sampling configurations (nodes), connecting those samples with local transitions (edges), thus creating a roadmap of valid



(a) Cafe



(b) Testbed Environment 1



(c) Testbed Environment 2

Figure 4.10: Benchmark environments studied in simulation (a) and experimentally (b-c).

paths. In our PRMs set up, we use uniform random sampling of configurations for node generation, identify the 10 nearest neighbors to each configuration using Euclidean distance, and attempt connections between neighbors using a straight line planner with a resolution of 5 cm for edge creation. The path planning was done using the Parasol Motion Planning Library from Texas A&M University [102].

Trajectory planning in Cafe To explore conceptual applications of the quadrotors to domestic and assistive robotics and to test the method in a more challenging environment, we choose a virtual coffee shop setting for our simulation testing. In

this setting, the Cargo Delivery Tasks (Problem 4.1.5) include: delivering coffee, delivering checks to the counter, and fetching items from high shelves (see Figure 4.10a). The UAV needs to pass a doorway, change altitude, and navigate between the tables, shelves, and counters. In all these tasks, both speed and load displacement are important factors. We want the service to be in a timely manner, but it is important that the swing of the coffee cup, which represents the load, is minimized so that the drink is not spilled. The Cafe is 30 m long, 10 m wide, and 5 m tall.

We generate the paths, and create trajectories for three different maximal load displacements (1°, 10°, and 25°). Since the path planning is the most complex for the largest bounding volume, and the same path can be reused for smaller maximal load displacements, we create the path once using the bounding volume with a 25° cone half aperture, and create path-following trajectories requiring the same or lower maximal load displacement. The proximity factor is $\delta = 5\,\text{cm}$ and the candidate action sets size is $k = 500$. We evaluate the number of added waypoints to meet the load displacement requirement, the trajectory duration, the maximum swing, and the maximum path-following error.

Table 4.4 summarizes the results of the trajectory characteristics for the three tasks in the coffee shop for different maximal load displacement. It demonstrates that we can follow the path with an arbitrary small load displacement. The maximum swing along a trajectory always stays under the required load displacement bound. The path-following error stays within 10 cm, regardless of the path and decreases as the maximal load displacement bound decreases. Therefore, by designing the bounding volume with 10 cm of clearance, the quadrotor-load system will not collide with the environment. The delivery time and number of added waypoints increase with the decrease of the required load displacement bound, as expected. Although, it is common sense that slower trajectories produce less

Table 4.4: Summary of path and trajectory results for different tasks in the Cafe setting: task name and maximum allowed load displacement ($\|\boldsymbol{\eta}\|$), collision-free path length, and number of waypoints, trajectory waypoints after bisection, trajectory durations (t), maximum swing ($\|\boldsymbol{\eta}\|$), and maximum deviation from the path (error).

Task		Path		Trajectory			
Name	$\|\boldsymbol{\eta}\|$ (°)	Length (m)	Pts.	Pts.	t (°)	$\|\boldsymbol{\eta}\|$ (°)	Error (m)
Coffee Delivery	45	23.20	3	5	33.16	27.73	0.09
	25	23.20	3	7	43.02	24.60	0.08
	10	23.20	3	17	67.18	9.34	0.06
	5	23.20	3	23	85.58	4.47	0.05
	1	23.20	3	97	258.62	0.86	0.03
Pay	45	15.21	1	3	25.98	18.23	0.06
	25	15.21	1	3	25.98	18.23	0.06
	10	15.21	1	7	28.64	8.94	0.05
	5	15.21	1	15	53.72	3.73	0.03
	1	15.21	1	63	172.28	0.92	0.01
Special request	45	32.57	1	7	52.54	24.11	0.07
	25	32.57	1	7	52.54	24.11	0.07
	10	32.57	1	22	82.76	9.58	0.05
	5	32.57	1	31	108.12	4.79	0.03
	1	32.57	1	128	349.86	0.92	0.01

swing, the agent automatically chooses waypoints so not to needlessly slow down the trajectories.

Figure 4.11 depicts the trajectory of the quadrotor and the load during the Coffee Delivery Task for required maximal load displacements of 1°, 5°, 10°, 15°, 25°, and 45°. The trajectories are smooth, and the load's residual oscillations are minimal in all of them. The trajectory and path overlay is presented in Figures 4.12b-4.12e. The number of inserted waypoints increases for the smaller angles. The path-following error over the trajectory length is displayed in Figure 4.12f. The accumulated squared error profiles differ based on the load displacement angle bound, with the 1° trajectory having accumulated error significantly smaller than other two trajectories.

To analyze how the roadmap size influences the trajectory, we generated 100

random queries in the Cafe environment. We also created four roadmaps each with 10, 50, 100, and 500 nodes and 56, 532, 1272, and 8060 edges, respectively. Then we tested to see which of the random queries could be solved in each roadmap. Figure 4.13 (a) shows that the probability of finding a path grows exponentially with the roadmap size. The roadmap with 10 nodes was sufficient to solve only 66 % of queries, the roadmap with 50 nodes was sufficient for 96 % queries, while the roadmap with 100 nodes, solved 99 % of queries. Lastly, the roadmap with 500 nodes was able to produce a path for all the queries.

Figure 4.13 depicts the trajectory characteristics for different roadmap sizes. As the roadmap grows, the number of waypoints in a path grows as well (b). The variance in the number of waypoints increases with the roadmap size as expected. Since on average the path segments are shorter with the larger roadmap size, the maximal load displacement and its standard deviation decreases (c), and the trajectories take a longer time (d). This gives us experimental confirmation that we can use roadmap size to control the amount of swing in trajectories and to fine-tune the balance between the load-displacement and the time of delivery. For example, consider the case of the example query shown with dashed lines in Figure 4.13 (b-d). If we needed the swing to be less than 25°, a 100-node roadmap would have been sufficient. If however, we required the load displacement to be 15°, we would need a roadmap with 500 nodes.

We can see that with the increased (Figure 4.13) roadmap size, the maximum load displacement decreases while the duration of the trajectory increases, although not as fast. This is interesting, because the time it takes to generate the trajectory does not depend on its length. It depends linearly on its duration. So, trajectories generated with larger roadmaps will have more waypoints (and be smoother, more direct), but will take slightly longer to compute than less direct, longer trajectories with fewer waypoints.

Experimental evaluation The goal of the experiment is to show the discrepancy between the simulation results and the observed experimental trajectories, and to demonstrate the safety and feasibility of the method by using a quadrotor to deliver a cup of water. To check the discrepancy between the simulation predictions of the maximum load displacement in simulation and experimentally, we run queries in two testbed environments to generate trajectories in simulation. The testbed environments are all 2.5 m by 3 m, with the ceiling height varying from 1.5 m to 2.5 m. The obstacles are uniform triangular prisms with 60 cm sides. Shorter obstacles are 60 cm tall, while the tall ones are 1.2 m tall. They differ in number of obstacles, their sizes, and locations. The first testbed environment contains three obstacles positioned diagonally across the room and the same landing platform (see Figure 4.10b). Two obstacles are 0.6 m tall, while the third one is 1.2 m tall. The second testbed environment is filled with five 1.2 m tall obstacles. They are in the corners and in the middle of a 2.5 m by 2 m meters rectangle centered in the room (Figure 4.10c). This is a challenging, cluttered space that allows us to experimentally test an urban environment setting.

Table 4.5 summarizes the difference in observed versus predicted maximum load displacements for these tasks over three trials. The observed maximum displacement is between 4° and 5° higher experimentally than in simulation. This is expected and due to unmodeled system dynamics, noise, wind influence and other factors, and it matches load displacement observed during hover.

Figure 4.14 shows three trials of the experimentally flown trajectory in Environment 2 (Figure 4.10c), with the predicted simulation. The vehicle trajectory (Figure 4.14a) matches very closely between trials and the simulation. The load's trajectories (Figure 4.14b) show higher uncertainty over the position of the load at any given time. However, the load displacement is bounded and stays within 10°. The accumulated path-following error (Figure 4.15) is slightly larger in ex-

Table 4.5: Summary of experimental results in different obstacle configurations. Path and simulated and experimental trajectory characteristics for different obstacle configurations:task name and maximum allowed load displacement ($\boldsymbol{\eta}$); obstacle-free path length (l), and its number of waypoints (#); simulated trajectory: waypoints after bisection (#), planned trajectory durations (t), maximum swing ($\boldsymbol{\eta}$), and maximum deviation from the path (error); experimental trajectory: maximum swing ($\boldsymbol{\eta}$) and maximum deviation from the path (error). The experimental results are average over three trials.

Task		Path		Simulation				Experiment	
Test.	$\|\boldsymbol{\eta}\|$ (°)	l (m)	#	#	t (s)	$\|\boldsymbol{\eta}\|$ (°)	Error (m)	$\|\boldsymbol{\eta}\|$ (°)	Error (m)
Env. 1	10	3.86	3	3	12.64	7.25	0.05	11.32	0.16
	5	3.86	3	4	15.68	3.63	0.03	7.99	0.11
	1	3.86	3	16	44.98	0.98	0.04	5.05	0.07
Env. 2	10	3.23	2	2	10.18	5.51	0.05	9.94	0.16
	5	3.23	2	4	15.80	2.66	0.05	6.72	0.11
	1	3.23	2	16	42.24	0.69	0.02	4.98	0.07

periments than in simulation, but follows the similar profile. The power spectral density (Figure 4.14c) is consistent between the three trials, and the simulated trajectory lacks higher frequencies, which is expected.

The demonstration of the practical feasibility and safety of this method was demonstrated by using the system to deliver a cup of water to a static human subject. In this demonstration, the quadrotor's load is a 250 mL paper cup filled with 100 mL of water. In Environment 1 (see Figure 4.10b), a quadrotor needs to fly diagonally through the room, avoiding a set of three obstacles. The path and trajectory used for this demonstration are the same referenced in Table 4.5. A human subject was seated at the table. As the quadrotor completed the flight, it set the cup of water in front of the human, who detached it from the quadrotor [5]. As the experiment demonstrated, the small amount of the liquid does not negatively impact the trajectory. The dynamics depends on the load's center of the mass. Given that the liquid is confined to a small container, its center of the mass does not move significantly. Moreover, since we are ensuring swing-free trajectory tracking we

[5] A video of the human-quadrotor interaction and other experiments can be found at `https://www.cs.unm.edu/amprg/Research/Quadrotor/`

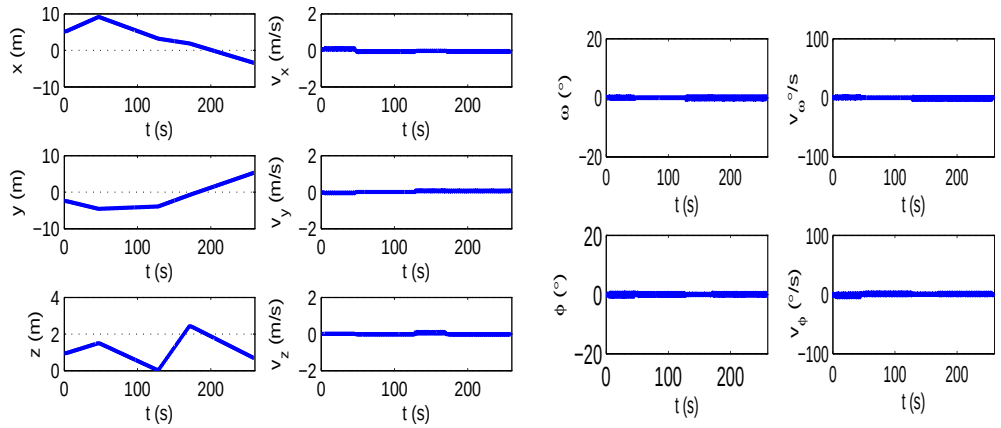
are minimizing the movement of the fluid inside of the container as well. For a larger container, it would be beneficial to thoroughly analyze the effect of liquid transport in comparison with solid objects.

4.4 Conclusion

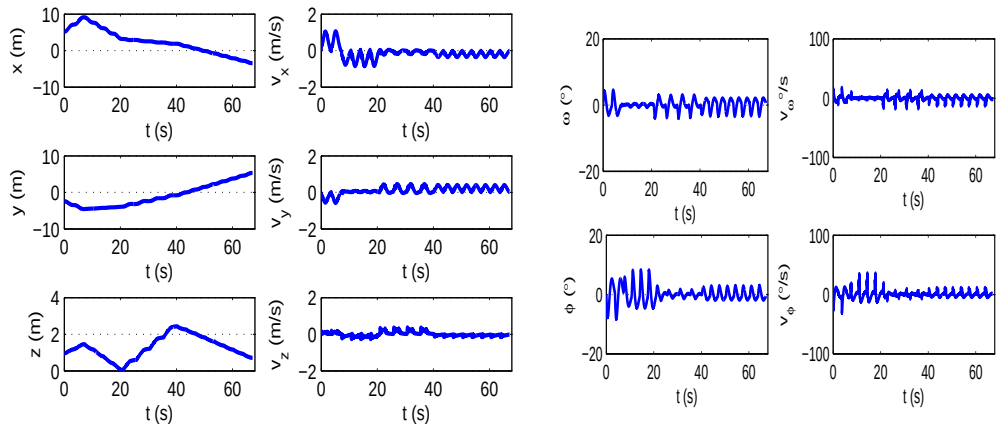
In this chapter we proposed an autonomous Aerial Cargo Delivery agent that works in environments with static obstacles to plan and create trajectories with bounded load displacements. At the heart of the method is the RL policy for minimal residual oscillations trajectories. Planning Minimal Residual Oscillations Trajectories consists of a large action space (over 10^6 actions) that deems learning impractical. For that reason, we find conditions that allow us to learn the minimal residual oscillations policy in an action subspace several orders of magnitude smaller. In addition, the learning generalizes to an extended state space. Then, we show how a RL policy learned through a single training can be adapted to perform different tasks leveraging different discretized action spaces. We modified the policy by restricting the domain's action space to plan a path-following trajectory with a bounded load displacement. Finally, we integrated the Swing-free Path-following and sampling-based path planning to solve the Cargo Delivery Task. We evaluated each of the modules separately, and showed that learning converges to a single policy, and that performs minimal residual oscillation delivery task, that the policy is viable with expending state and action spaces. The simulations quality is assessed through the comparison with experimental load displacement on a physical robot. Then we evaluated the Swing-free Path-following on three reference paths for varying values of the path-following parameters. The results demonstrated that the proposed method attains both good path-following and good load displacement characteristics. Lastly, results of the integration with

sampling-based motion planning show that the method creates collision-free trajectories with bounded load displacement for arbitrarily small bounds. We experimentally demonstrated the feasibility and safety of the method by having a quadrotor deliver a cup of water to a human subject. This chapter lays a foundation for Aerial Cargo Delivery in environments with static obstacles. In further work, trajectory smoothing can be applied to the generated trajectories to accelerate them by not stopping in the waypoints while at the same time ensuring that the trajectories remain collision-free. Moving obstacles can be handled through an on-line trajectory tracking that adapts to dynamical changes in environment.

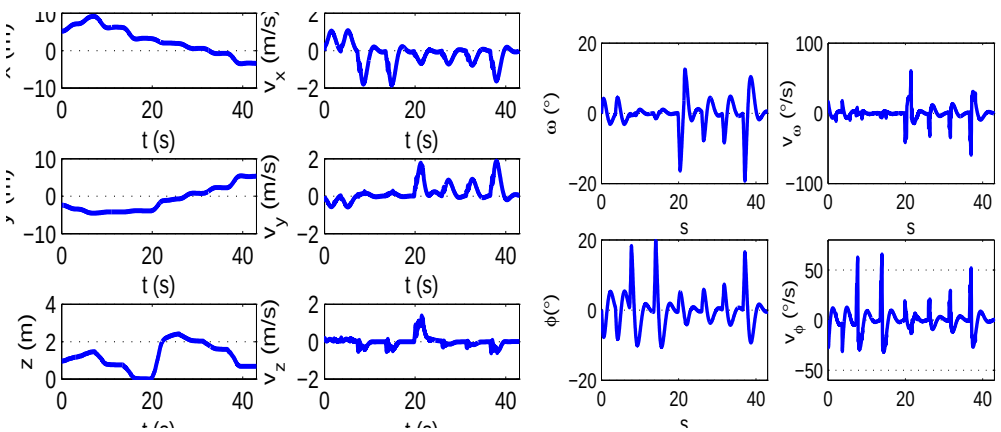
Beyond Aerial Cargo Delivery, this chapter address three important question relevant to AI. First, it applies reinforcement learning to a problem with very large action space. To address the curse of dimensionality, it proposes learning in relevant subspaces several orders of magnitude smaller, and planning in the full action space. The chapter contributes methods for finding the suitable subspaces. Second, it shows that using the feature vectors defined on a larger domain, the learned policy generalizes well outside of the training domain. Lastly, the chapter proposes learning PBTs on systems with non-linear dynamics by designing feature vectors that are linear maps over the constraints.



(a) $\sigma = 1°$

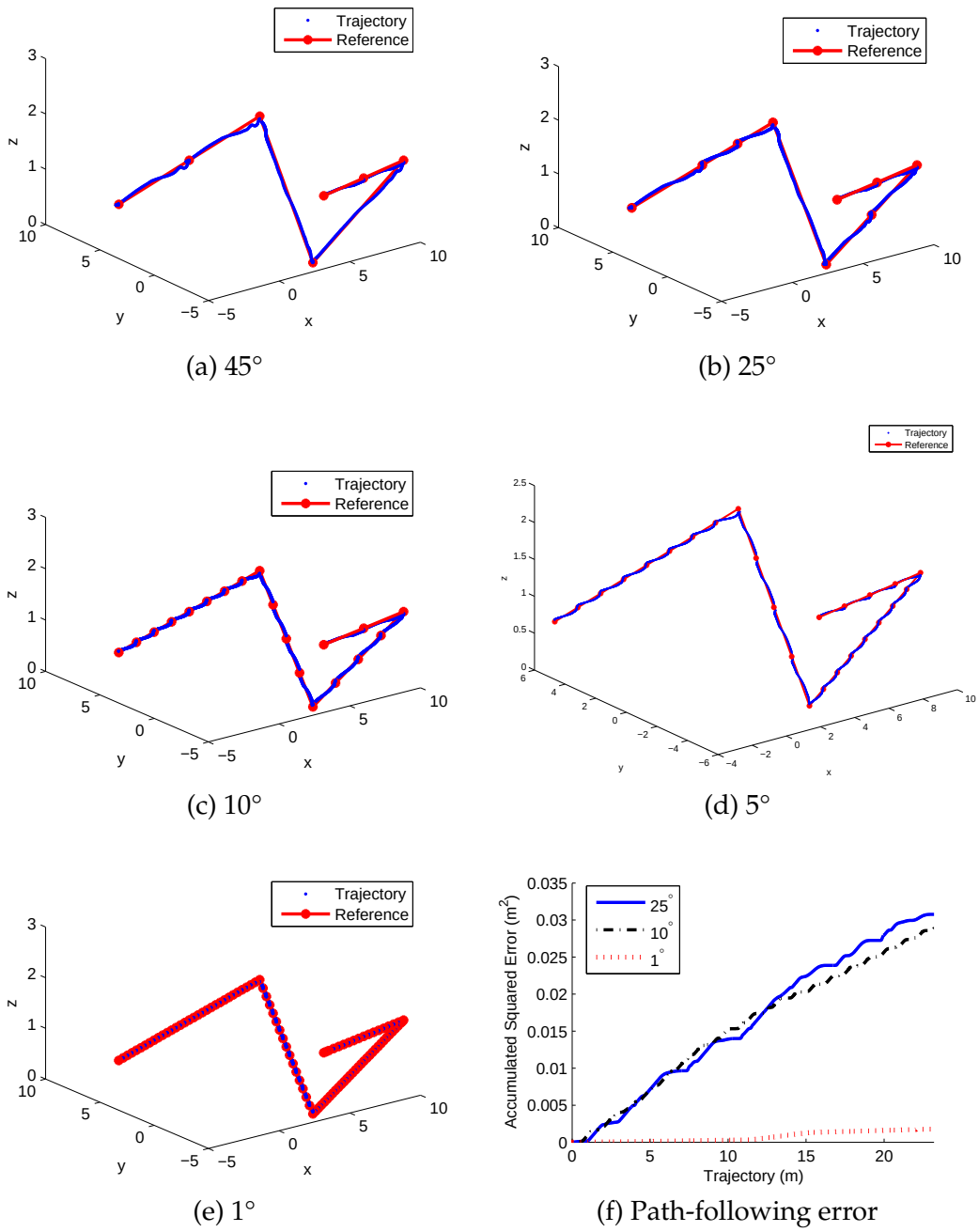


(b) $\sigma = 10°$



(c) $\sigma = 25°$

Figure 4.11: Quadrotor and load trajectories in Coffee Delivery Task in the Cafe for maximal allowed load displacement of 1°, 10°, and 25°. Zero altitude is the lowest point that a cargo equipped quadrotor can fly without the load touching the ground.

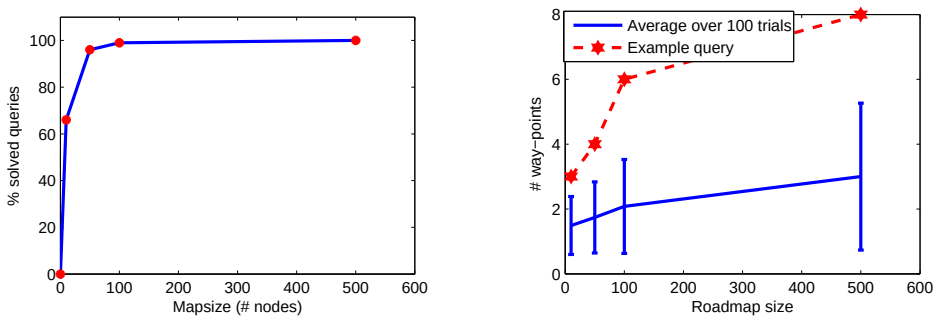


(a) 45° (b) 25°

(c) 10° (d) 5°

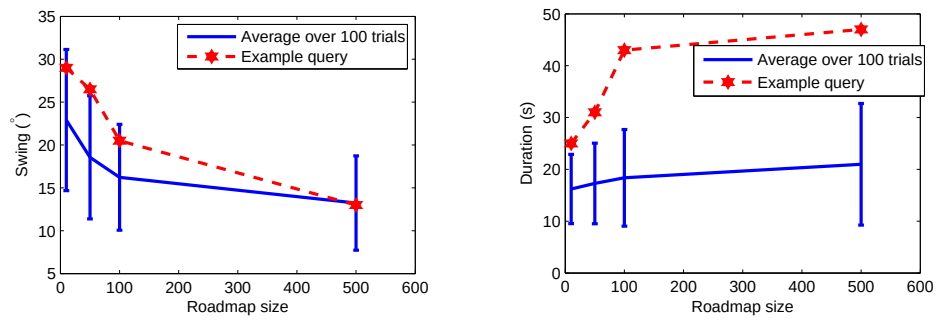
(e) 1° (f) Path-following error

Figure 4.12: Path bisecting for Coffee Delivery Task based on the maximum allowed load displacement (a-e). Accumulated squared path-following error along the trajectory for different maximum allowed load displacements (f).



(a) Percentage of solved queries

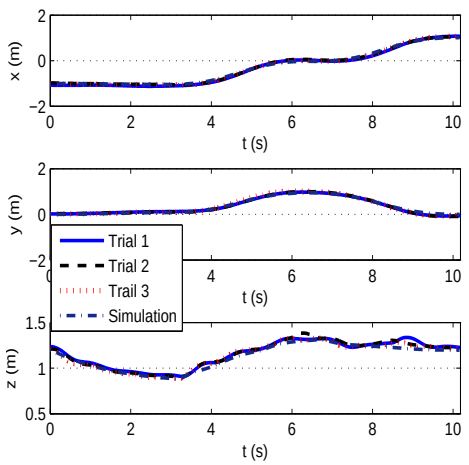
(b) number of waypoints



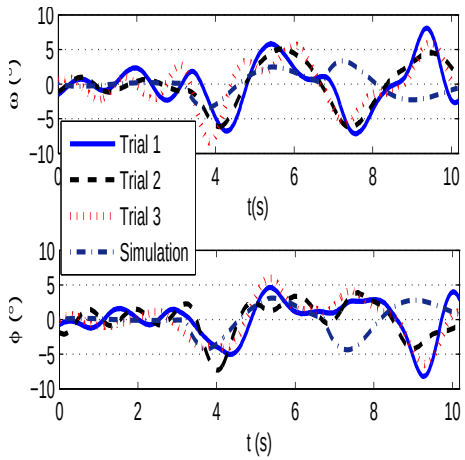
(c) Maximum load displacement

(d) Trajectory duration

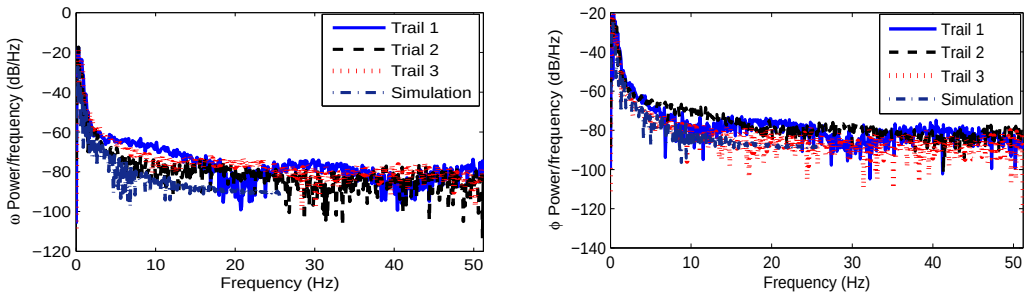
Figure 4.13: Roadmap size analysis results. Probability of solving a query (a), number of waypoints (b) maximum load displacement (c), and trajectory duration (d) as a function of roadmap size.



(a) Quadrotor position



(b) Load position



(c) Power spectral density

Figure 4.14: Experimental quadrotor (a) and load (b) trajectories, and power spectral density (c) in the second testbed configuration.

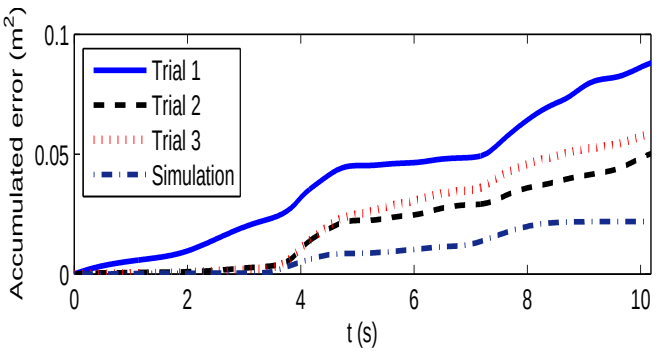


Figure 4.15: Accumulated path-following error in the second testbed configuration.

Chapter 5

PEARL for Deterministic Continuous Action MDPs

This chapter extends PEARL with distance-reducing preferences to continuous actions spaces. The chapter is based on [41] and sections of [42]. The specific contributions are:

1. Admissible family of policy approximations in Section 5.1.2,
2. Conditions for task goal's stability for control-affine systems (Section 5.1.2),
3. Continuous Action Fitted Value Iteration (CAFVI) in Section 5.1.3
4. Solution to a multi-agent Rendezvous Task (Section 5.2.3), and
5. Solution to Flying Inverted Pendulum (Section 5.2.4).

We extend AVI, a discrete action learning agent in PEARL to continuous action space to develop CAFVI. The novelty of CAFVI is a joint work with both value functions, state-value and action-value, to learn how to control the system.

CAFVI learns, globally to the state space, state-value function, which is negative of the Lyapunov. On the other hand, in the estimation step, it learns an state-value function locally around a state to estimate its maximum. CAFVI is critic-only and because the system dynamics is unknown, the value-function gradient is unavailable [44]. Thus, we develop a gradient-free method that divides-and-conquers the problem by finding the optimal input in each direction, and then combines them. Although problem decomposition via individual dimensions is a common technique for dimensionality reduction [124], this chapter shows that single-component policies lead to a stable system, offers three examples of such policies to turn the equilibrium into an asymptotically stable point, and characterizes systems for which the technique is applicable. The policies we develop are not only computationally efficient, scaling linearly with the action space dimensionality, but they produce *consistent* near-optimal actions; their outcome does not depend on the action samples used for calculation. The reinforcement learning agent is evaluated on a Minimal Residual Oscillations Task [41], a heterogeneous robot Rendezvous Task [41], and Flying Inverted Pendulum [42].

This chapter gives methods to implement an AVI with linear map approximation for a PBT, on control-affine systems [69] with unknown dynamics and in presence of a bounded drift. These tasks require the system to reach a goal state, while minimizing opposing preferences along the trajectory. While this approximation will not provide a solution for all learning tests, the method is fast and easy to implement, thus rendering an inexpensive tool to attempt before more heavy-handed approaches are attempted.

This chapter addresses the same optimal control problem as the related work described in Section 2.4.1. However, we use linearly parametrized state-value functions with linear regression rather than neural networks for parameter learning. Our method also learns the value function, which corresponds to the gener-

alized HBJ equation solution, through iterative minimization of the least squares error. However, we learn from samples and linear regression rather than neural networks. We are concerned with AVI methods in the RL setting - without knowing the system dynamics.

Discrete actions AVI has solved the Minimal Residual Oscillations Task for a quadrotor with a suspended load and has developed the stability conditions with a discrete action MDP (Chapter 4). Empirical validation in Chapter 4 shows that the conditions hold. This chapter characterizes basis vector forms for control-affine systems, defines admissible policies resulting in an asymptotically stable equilibrium, and analytically shows the system stability. The empirical comparison with discrete action AVI in Section 5.2.2 shows that CAFVI is both faster and performs the task with higher precision. This is because the decision-making quality presented here is not limited to the finite action space and is independent of the available samples. We also show wider applicability of the methods developed here by applying them to a multi-agent Rendezvous Task and a Flying Inverted Pendulum.

5.1 Methods

This section consists of four parts. First, Section 5.1.1 specifies the problem formulation for a task on a control-affine system suitable for approximate value iteration with linear basis vectors. Based on the task, the system and the preferences, we develop basis functions and write state-value function in control Lyapunov quadratic function form. Second, Section 5.1.2 develops sample-efficient policies that take the system to the goal and can be used for both planning and learning. Third, Section 5.1.3 places the policies into AVI setting to present a learning algorithm for the goal-oriented tasks. Together they give practical implementation

tools for solving PBTs through reinforcement learning on control-affine systems with unknown dynamics. We discuss these tools in Section 5.1.4.

5.1.1 Problem Formulation

Consider a discrete time, control-affine system with no disturbances as described in Equation (2.8) in Section 2.4. We assume that states are $\mathbf{s}(k) \in S \subseteq \mathbb{R}^{d_s}$, actions are defined on a closed interval around origin, $\mathbf{a}(k) \in A \subseteq \mathbb{R}^{d_a}$, $d_a \leq d_s$, $\mathbf{0} \in A$, and $\boldsymbol{g} : S \to \mathbb{R}^{d_s} \times \mathbb{R}^{\nu}$, $\boldsymbol{g}(\mathbf{s}(k))^T = [\boldsymbol{g}_1(\mathbf{s}(k)) \ ... \ \boldsymbol{g}_{d_a}(\mathbf{s}(k))]$ is regular for $\mathbf{s}(k) \in S \setminus \{\mathbf{0}\}$, nonlinear, and Lipschitz continuous. Drift $\boldsymbol{f} : S \to \mathbb{R}^{d_a}$, is nonlinear, and Lipschitz. Assume that the system is controllable [59]. We are interested in autonomously finding actions $\mathbf{a}(k)$ that take the system to its origin in a timely manner while reducing $\|\boldsymbol{C}\mathbf{s}\|$ along the trajectory, where $\boldsymbol{C}^T = [\boldsymbol{c}_1, ..., \boldsymbol{c}_{n_o}] \in \mathbb{R}^{n_o} \times \mathbb{R}^{d_s}$, $n_o \leq d_s$ is nonsigular. A discrete time, deterministic MDP with continuous state and action spaces

$$\mathcal{M} : \ (X, U, \boldsymbol{D}, \rho) \tag{5.1}$$

describes the problem as outlined in Section 3.1.1.

We learn state-value function, V, because its approximation can be constructed to define a control Lyapunov candidate function, and in tandem with the right policy it can help assess system stability. For discrete action MDPs, greedy policy, Equation (4.3), is a brute force search over the available samples. When action space is continuous, Equation (4.3) becomes an optimization problem over unknown function $\mathbf{D}$. We consider analytical properties of $Q(\mathbf{s}, \mathbf{a})$ for a fixed state $\mathbf{s}$ and knowing V, but having only knowledge of the structure of the transition function $\mathbf{D}$. The key insight we exploit is that existence of a maximum of the action-value function $Q(\mathbf{s}, \mathbf{a})$, as a function of input $\mathbf{a}$, depends only on the learned parametrization of the state-value function V.

AVI algorithms with linear map approximators require basis vectors. Given the state preference minimization, we choose quadratic basis functions

$$\boldsymbol{F}_i(\mathbf{s}) = \|\boldsymbol{c}_i^T \mathbf{s}\|^2, \quad i = 1, ..., n_o. \tag{5.2}$$

so that state-value function approximation, V, is a control Lyapunov candidate function. Consequently, V is,

$$V(\mathbf{s}) = \sum_{i=1}^{d_g} \theta_i \boldsymbol{F}_{\boldsymbol{i}}(\mathbf{s}) = (\boldsymbol{C}\mathbf{s})^T \boldsymbol{\Theta} (\boldsymbol{C}\mathbf{s}) = \mathbf{s}^T \boldsymbol{\Lambda} \mathbf{s} \tag{5.3}$$

for a diagonal matrix $\boldsymbol{\Theta} = \text{diag}(\theta_1, \theta_2, ..., \theta_{no})$, and a symmetric matrix $\boldsymbol{\Lambda}$. Let's assume that $\boldsymbol{\Lambda}$ has full rank. AVI learns the parametrization $\boldsymbol{\Theta}$ using a linear regression. Let $\boldsymbol{\Gamma} = -\boldsymbol{\Lambda}$. Note, that if $\boldsymbol{\Theta}$ is negative definite, $\boldsymbol{\Lambda}$ is as well, while $\boldsymbol{\Gamma}$ is positive definite, and vice versa. Let also assume that when $\boldsymbol{\Gamma} > 0$ the system drift is bounded with $\mathbf{s}$ with respect to $\boldsymbol{\Gamma}$-norm, $\boldsymbol{f}(\mathbf{s})^T \boldsymbol{\Gamma} \boldsymbol{f}(\mathbf{s}) \leq \mathbf{s}^T \boldsymbol{\Gamma} \mathbf{s}$. This characterizes system drift, conductive to the task. We empirically demonstrate its sufficiency in the robotic systems we consider.

To summarize the system assumptions used in the reminder of the chapter:

1. The system is controllable and the equilibrium is reachable. In particular, we use,

   $$\exists i, 1 \leq i \leq d_u, \text{ such that } \boldsymbol{f}(\mathbf{s}) \boldsymbol{\Gamma} \boldsymbol{g}_i(\mathbf{s}) \neq 0, \tag{5.4}$$

   and that $\boldsymbol{g}(\mathbf{s})$ is regular outside of the origin,

   $$\boldsymbol{g}(\mathbf{s})^T \boldsymbol{\Gamma} \boldsymbol{g}(\mathbf{s}) > 0, \mathbf{s} \in S \setminus \{\mathbf{0}\} \tag{5.5}$$

2. action is defined on a closed interval around origin,

   $$\mathbf{0} \in A \tag{5.6}$$

Table 5.1: Summary of chapter-specific key symbols and notation.

Symbol	Description
$V : X \to \mathbb{R}$, $V(\mathbf{s}) = \mathbf{s}^T \boldsymbol{\Lambda} \mathbf{s}$	state-value function
$\boldsymbol{C}\mathbf{s}$	preferences to minimize
$\boldsymbol{\Lambda} = \boldsymbol{C}^T \boldsymbol{\Theta} \boldsymbol{C}$	Combination of task preferences and value function parametrization
$\boldsymbol{\Gamma} = -\boldsymbol{\Lambda}$	Task-learning matrix
$\Delta Q(\mathbf{s}, \hat{\boldsymbol{a}})$	Policy $\hat{\pi^Q}$ in state $\mathbf{s}$
$\boldsymbol{e_n}$	n$^{\text{th}}$ axis unit vector
$\mathbf{a} \in U$	action vector
$a \in \mathbb{R}$	Univariate action variable
$\boldsymbol{a_n} \in \mathbb{R}$	Set of vectors in direction of n$^{\text{th}}$ axis
$\hat{a}_n \in \mathbb{R}$	Estimate in direction of the n$^{\text{th}}$ axis
$\hat{\boldsymbol{a}}_{\boldsymbol{n}} = \sum_{i=1}^{n} \hat{a}_n \boldsymbol{e_i}$	Estimate over first n axes
$\hat{\boldsymbol{a}}$	Estimate of Q's maximum with a policy
$Q^{(\boldsymbol{p})}_{\mathbf{s},n}(a) = Q(\mathbf{s}, \boldsymbol{p} + u\boldsymbol{e_n})$	Univariate function in the direction of axis $\boldsymbol{e_n}$, passing through point $\boldsymbol{p}$

3. The drift is bounded,

$$\boldsymbol{f}(\mathbf{s})^T \boldsymbol{\Gamma} \boldsymbol{f}(\mathbf{s}) \leq \mathbf{s}^T \boldsymbol{\Gamma} \mathbf{s}, \text{ when } \boldsymbol{\Gamma} > 0 \tag{5.7}$$

Table 5.1 presents a summary of the key symbols.

5.1.2 Policy Approximation

This section looks into an efficient and a consistent policy approximation for Equation (4.3) that leads the system, Equation (2.8), to a goal state in the origin. Here, we learn the action-value function Q on the axes, and assume a known estimate of the state-value function approximation V. For the policy to lead the system to the origin from an arbitrary state, the origin must be asymptotically stable. Negative of the state-value function V can be a control Lyapunov function, and the V

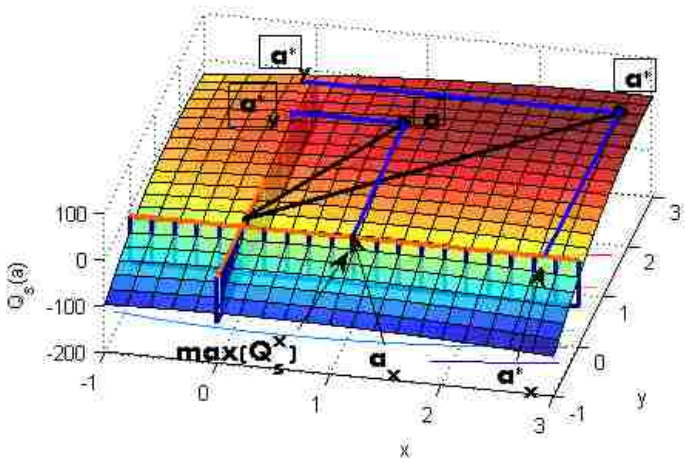


Figure 5.1: Example of two dimensional action and a quadratic value function. $\mathbf{a}^*$ is the optimal action, $\mathbf{a}$ is the one selected.

function needs to be increasing in time. That only holds true when the policy approximation makes an improvement, i.e., the policy needs to transition the system to a state of a higher value ($V(\mathbf{s}_{n+1}) > V(\mathbf{s}_n)$). To ensure the temporal increase of V, the idea is to formulate conditions on the system dynamics and value function V, for which Q, considered as a function only of the action, is concave and has a maximum. In this work, we limit the conditions to a quadratic form Q. When we establish maximum's existence, we approximate it by finding a maximum on the axes and combining them together. Figure 5.1 illustrates this idea. To reduce the dimensionality of the optimization problem, we propose a divide and conquer approach. Instead of solving one multivariate optimization, we solve d_a univariate optimizations on the axes to find a highest valued point on each axis, u_i. The composition of the axes' action selections is the selection vector $u = [u_1 \; .. \; u_{d_a}]^T$. This section develops the policy approximation following these steps:

1. show that Q is a quadratic form and has a maximum (Proposition 5.1.1)

2. define admissible policies that ensure the equilibrium's asymptotic stability

(Theorem 5.1.2), and

3. find a sampling-based method for calculating consistent, admissible policies in $O(d_u)$ time with no knowledge of the dynamics (Theorem 5.1.4).

Since the greedy policy, Equation (4.3), depends on action-value Q, Proposition 5.1.1 gives the connection between value function, Equation (4.2), and corresponding action-value function Q.

Proposition 5.1.1. *Action-value function $Q(\mathbf{s}, \mathbf{a})$, Equation (2.6), is a quadratic function of action $\mathbf{a}$ for all states $\mathbf{s} \in X$ for a MDP, Equation (5.1), with state-value function V Equation (4.2). When $\boldsymbol{\Theta}$ is negative definite, the action-value function Q is concave and has a maximum.*

Proof. Evaluating $Q(\mathbf{s}, \mathbf{a})$ for an arbitrary state $\mathbf{s}$, we get

$$\begin{aligned} Q(\mathbf{s}, \mathbf{a}) &= V(\boldsymbol{D}(\mathbf{s}, \mathbf{a})) = V(\boldsymbol{f}(\mathbf{s}) + \boldsymbol{g}(\mathbf{s})\mathbf{a}), \quad \text{from Equation (2.8)} \\ &= (\boldsymbol{f}(\mathbf{s}) + \boldsymbol{g}(\mathbf{s})\mathbf{a}))^T \boldsymbol{\Lambda} (\boldsymbol{f}(\mathbf{s}) + \boldsymbol{g}(\mathbf{s})\mathbf{a}) \end{aligned}$$

Thus, Q is a quadratic function of action $\mathbf{a}$ at any state $\mathbf{s}$. To show that Q has a maximum, we inspect Q's Hessian,

$$HQ(\mathbf{s}, \mathbf{a}) = \begin{bmatrix} \frac{\partial^2 Q(\mathbf{s},\mathbf{a})}{\partial u_1 \partial u_1} & \cdots & \frac{\partial^2 Q(\mathbf{s},\mathbf{a})}{\partial u_1 \partial u_{d_u}} \\ & \cdots & \\ \frac{\partial^2 Q(\mathbf{s},\mathbf{a})}{\partial u_{d_u} \partial u_1} & \cdots & \frac{\partial^2 Q(\mathbf{s},\mathbf{a})}{\partial u_{d_u} \partial u_{d_u}} \end{bmatrix} = 2\boldsymbol{g}(\mathbf{s})^T \boldsymbol{\Lambda} \boldsymbol{g}(\mathbf{s}).$$

The Hessian is negative definite because $\boldsymbol{g}(\mathbf{s})$ is regular for all states $\mathbf{s}$ and $\boldsymbol{\Theta} < 0$, which means that $\boldsymbol{\Lambda} < 0$ as well. Therefore, the function is concave, with a maximum. □

The state-value parametrization $\boldsymbol{\Theta}$ is fixed for the entire state space. Thus, Proposition 5.1.1 guarantees that when the parametrization $\boldsymbol{\Theta}$ is negative definite, the action-value function Q has a single maximum. Next, we show that the right policy can ensure the progression to the goal, but we first define the acceptable policies.

Definition 5.1.1. *Policy approximation $\hat{\boldsymbol{a}} = \hat{\pi^{Q}}(\mathbf{s})$ is admissible, if it transitions the system to a state with a higher value when one exists, i.e., when the following holds for policy's gain at state $\mathbf{s}$, $\Delta Q(\mathbf{s}, \hat{\boldsymbol{a}}) = Q(\mathbf{s}, \hat{\boldsymbol{a}}) - V(\mathbf{s})$:*

1. $\Delta Q(\mathbf{s}, \hat{\boldsymbol{a}}) > 0$, *for* $\mathbf{s} \in X \setminus \{\mathbf{0}\}$, *and*
2. $\Delta Q(\mathbf{s}, \hat{\boldsymbol{a}}) = 0$, *for* $\mathbf{s} = \mathbf{0}$.

Theorem 5.1.2 shows that an admissible policy is sufficient for the system to reach the goal.

Theorem 5.1.2. *Let $\hat{\mathbf{a}} = \hat{\pi}_{Q}(\mathbf{s})$ be an admissible policy approximation. When $\boldsymbol{\Lambda} < 0$, and the drift is bounded with Equation* (5.7)*, the system, Equation* (2.8)*, with value function, Equation* (4.2)*, progresses to an asymptotically stable equilibrium under policy $\hat{\pi^{Q}}$.*

Proof. Consider $W(\mathbf{s}) = -V(\mathbf{s}) = \mathbf{s}^{T}\boldsymbol{\Gamma}\mathbf{s}$. W is a candidate for control Lyapunov function because $\boldsymbol{\Gamma} > 0$.

To show the asymptotic stability, a W needs to be monotonically decreasing in time $W(\mathbf{s}_{n+1}) \leq W(\mathbf{s}_n)$ with equality holding only when the system is in the equilibrium, $\mathbf{s}_n = \mathbf{0}$. Directly from the definition of the admissible policy, for the state $\mathbf{s}_n \neq \mathbf{0}$,

$$\begin{aligned} W(\mathbf{s}_{n+1}) - W(\mathbf{s}_n) &= -Q(\mathbf{s}_n), \hat{\pi}^{Q}(\mathbf{s}_n) + V(\mathbf{s}_n) \\ &= V(\mathbf{s}_n) - Q(\mathbf{s}_n, \hat{\boldsymbol{a}}) < 0. \end{aligned}$$

When

$$\begin{aligned} \boldsymbol{x_n} = \mathbf{0}, &\implies \boldsymbol{x_{n+1}} = \boldsymbol{f}(\mathbf{0}) = 0, \text{ because of Equation (5.7)} \\ &\implies W(\mathbf{s}_{n+1}) = 0. \end{aligned}$$

□

Theorem 5.1.2 gives the problem formulation conditions for the system to transition to the goal state. Now, we move to finding sample-based admissible policies by finding maximums of Q in the direction parallel to an axis and passing through a point. Because Q has quadratic form, its restriction to a line is a quadratic function of one variable. We use Lagrange interpolation to find the coefficients of Q on a line, and find the maximum in the closed form. We first introduce the notation for Q's restriction in an axial direction, and its samples along the direction.

Definition 5.1.2. *Axial restriction of Q passing through point $\boldsymbol{p}$, is a univariate function* $Q^{(\boldsymbol{p})}_{\mathbf{s},i}(a) = Q(\mathbf{s}, \boldsymbol{p} + a\boldsymbol{e_i})$.

If $\boldsymbol{q_i} = [Q^{\boldsymbol{p}}_{\mathbf{s},1}(a_{i1})\ Q^{\boldsymbol{p}}_{\mathbf{s},2}(a_{i2})\ Q^{\boldsymbol{p}}_{\mathbf{s},3}(a_{i3})]^T$, are three samples of $Q^{(\boldsymbol{p})}_{\mathbf{s},i}(a)$ obtained at points $[a_{i1}\ a_{i2}\ a_{i3}]$, then $Q(\mathbf{s}, \boldsymbol{p} + a\boldsymbol{e_i})$, is maximized at

$$\begin{aligned} \hat{a}_i &= \min(\max(\hat{a^*}_i, a^l_i), a^u_i), \text{ where} \\ \hat{a}^*_i &= \frac{\boldsymbol{q_i}^T \cdot ([a^2_{i2} \quad a^2_{i3} \quad a^2_{i1}] - [a^2_{i3} \quad a^2_{i1} \quad a^2_{i2}])^T}{2\boldsymbol{q_i}^T \cdot ([a_{i2} \quad a_{i3} \quad a_{i1}] - [a_{i3} \quad a_{i1} \quad a_{i2}])^T}, \end{aligned} \tag{5.8}$$

on the interval, $a^l_i \leq a \leq a^u_i$. Equation (5.8) comes directly from Lagrange interpolation of a univariate second order polynomial to find the coefficients of the quadratic function, and then equating the derivative to zero to find its maximum. In the stochastic case, instead of Lagrange interpolation, linear regression yields the coefficients.

A motivation for this approach is that maximum finding in a single direction is computationally efficient and consistent. A single-component policy is calculated in constant time. In addition, the action selection on an axis calculated with Equation (5.8) is *consistent*, i.e. it does not depend on the sample points a_{ij} available to calculate it. This is direct consequence of quadratic function being uniquely determined with arbitrary three points. It means that a policy based on Equation (5.8) produces the same result regardless of the action samples used, which is important in practice where samples are often hard to obtain.

We now give a lemma that shows a single-component policy characteristics. Later we integrate them together into admissible policies. The lemma shows that the single component policy is stable on an interval around zero.

Lemma 5.1.3. *A single action policy approximation, Equation* (5.8)*, for an action component* i*,* $1 \leq i \leq d_a$ *has the following characteristics:*

1. *There is an action around zero that does not decrease system's state value upon transition, i.e.,* $\exists a_0 \in [a_l^i, a_u^i]$ *such that* $Q_{\mathbf{s},i}^{(\boldsymbol{p})}(a) \geq Q(\mathbf{s}, \boldsymbol{p})$.
2. $Q_{\mathbf{s},i}^{(\mathbf{0})}(\hat{a}_i) - V(\mathbf{s}) \geq 0$, *when* $\mathbf{s} \neq \mathbf{0}$
3. $Q(0, \hat{a}_i \boldsymbol{e_i}) - V(0) = 0$

The proof for Lemma 5.1.3 is in Appendix A.

We give three consistent and admissible policies as examples. First, the Manhattan Policy finds a point that maximizes Q's restriction on the first axis, then iteratively finds maximums in the direction parallel to the subsequent axes, passing through points that maximize the previous axis. The second policy approximation, Convex Sum Policy, is a convex combination of the maximums found independently on each axis. Unlike the Manhattan Policy that works serially, the Convex Sum Policy parallelizes well. Third, Axial Sum Policy is the

maximum of the Convex Sum Policy approximation and nonconvex axial combinations. This policy is also parallelizable. All three policies scale linearly with the dimensions of the action $O(d_u)$. Next, we show that they are admissible.

Theorem 5.1.4. *The MDP Equation* (5.1)*, with value function, Equation* (4.2)*, bounded drift, Equation* (5.7)*, and a negative definite* $\boldsymbol{\theta}$*, starting at an arbitrary state* $\mathbf{s} \in S$*, and on a set* A*, Equation* (5.6)*, progresses to an equilibrium in the origin under any of the following policies:*

1. *Manhattan Policy:*

$$\boldsymbol{\pi}_{\boldsymbol{m}}^{Q}: \begin{cases} \hat{a}_1 = \underset{a_l^1 \le u \le a_u^1}{\operatorname{argmax}} Q_{\mathbf{s},1}^{(\mathbf{0})}(a) \\ \\ \hat{a}_n = \underset{a_l^n \le a \le a_u^n}{\operatorname{argmax}} Q_{\mathbf{s},n}^{(\hat{\boldsymbol{a}}_{\boldsymbol{n-1}})}(a), \quad n \in [2, .., d_a], \hat{\boldsymbol{a}}_{\boldsymbol{n-1}} = \sum_{i=1}^{n-1} \hat{a}_i \boldsymbol{e_i}. \end{cases} \tag{5.9}$$

2. *Convex Sum Policy:*

$$\boldsymbol{\pi}_{\boldsymbol{c}}^{Q}: \ \hat{\boldsymbol{a}} = \sum_{i=1}^{d_u} \lambda_i \boldsymbol{e_i} \underset{a_l^i \le a \le a_u^i}{\operatorname{argmax}} Q_{\mathbf{s},i}^{(\mathbf{0})}(a), \quad \sum_{i=1}^{d_a} \lambda_i = 1 \tag{5.10}$$

3. *Axial Sum Policy:*

$$\boldsymbol{\pi}_{\boldsymbol{s}}^{Q}: \ \hat{\boldsymbol{a}} = \begin{cases} \boldsymbol{\pi}_{\boldsymbol{c}}^{Q}(\mathbf{s}), & Q(\mathbf{s}, \boldsymbol{\pi}_{\boldsymbol{c}}^{Q}(\mathbf{s})) \ge Q(\mathbf{s}, \boldsymbol{\pi}_{\boldsymbol{n}}^{Q}(\mathbf{s})) \\ \\ \boldsymbol{\pi}_{\boldsymbol{n}}^{Q}(\mathbf{s}), & \textit{otherwise} \end{cases} \tag{5.11}$$

where

$$\boldsymbol{\pi}_{\boldsymbol{n}}^{Q}(\mathbf{s}) = \sum_{i=1}^{d_a} \boldsymbol{e_i} \underset{a_l^i \le a \le a_u^i}{\operatorname{argmax}} Q_{\mathbf{s},i}^{(\mathbf{0})}(a)$$

The proof for the Theorem 5.1.4 is in Appendix B.

A consideration in reinforcement learning, applied to robotics and other physical systems, is balancing exploitation and exportation [120]. Exploitation ensures the safety of the system, when the policy is sufficiently good and yields no learning. Exploration forces the agent to perform suboptimal steps, and the most often used ϵ-greedy policy performs a random action with probability ϵ. Although the random action can lead to knowledge discovery and policy improvement, it also poses a risk to the system. The policies presented here fit well in on-line RL paradigm, because they allow safe exploration. Given that they are not optimal, they produce new knowledge, but because of their admissibility and consistency, their action of choice is safe to the physical system. For systems with independent actions, Axial Sum Policy is optimal (see Appendix C).

5.1.3 Continuous Action Fitted Value Iteration (CAFVI)

We introduced an admissible, consistent, and efficient decision-making method for learning action-value function Q locally, at fixed state $\mathbf{s}$, and fixed learning iteration (when $\boldsymbol{\theta}$ is fixed) without knowing the system dynamics. Now, the decision-making policies are integrated into a AVI framework [36, 27] to produce a RL agent for continuous state and action MDPs tailored for control-affine nonlinear systems. The algorithm learns the parameterization $\boldsymbol{\theta}$, and works much like approximate value iteration [36] to learn state-value function approximation $\boldsymbol{\theta}$, but the action selection uses sampling-based policy approximation on the action-value function Q. Algorithm 5.7 shows an outline of the proposed *continuous action fitted value iteration*, CAFVI. It first initializes $\boldsymbol{\theta}$ with a zero vector. Then, it iteratively estimates Q function values and uses them make a new estimate of $\boldsymbol{\theta}$. First, we randomly select a state $\mathbf{s}_s$ and observe its reward. Line 6 collects the samples. It uniformly samples the state space for $\mathbf{s}_{l_s}$. Because we need three data points for Lagrangian interpolation of a quadratic function, three action samples

per action dimensions are selected. We also obtain, either through a simulator or an observation, the resulting state $\mathbf{s}'_{ij}$ when $\mathbf{a}_{ij}$ is applied to $\mathbf{s}_{l_s}$. Line 7 estimates the action-value function locally, for $\mathbf{s}_{l_s}$ and $\mathbf{a}_{ij}$ using the current $\boldsymbol{\theta_l}$ value. Next, the recommended action is calculated, $\hat{\boldsymbol{a}}$. Looking up the available samples or using a simulator, the system makes the transition from $\mathbf{s}_{l_s}$ using action $\hat{\boldsymbol{a}}$. The algorithm makes a new estimate of $V(\mathbf{s}_{l_s})$. After n_s states are processed, Line 10 finds new $\boldsymbol{\theta}$ that minimizes the least squares error for the new state-value function estimates v_{l_s}. The process repeats until either $\boldsymbol{\theta}$ converges, or a maximum number of iterations is reached.

Algorithm 5.7 Continuous Action Fitted Value Iteration (CAFVI).

Input: S, A, discount factor γ

Input: basis function vector $\boldsymbol{F}$

Output: $\boldsymbol{\theta}$

1: $\boldsymbol{\theta}_0, \boldsymbol{\theta}_1 \leftarrow$ zero vector
2: $l \leftarrow 1$
3: **while** ($l \leq max\_iterations$) and $\|\boldsymbol{\theta}_l - \boldsymbol{\theta}_{l-1}\| \geq \epsilon$ **do**
4: **for** $l_s = 1, .., n_s$ **do**
5: sample state $\mathbf{s}_{l_s}$ and observe its reward R_{l_s}
6: $\{\mathbf{s}_{l_s}, \mathbf{a}_{ij}, \mathbf{s}'_{ij} | i = 1, .., d_a, j = 1, 2, 3\}$ {obtain system dynamics samples}
7: for all i,j, $q_{ij} \leftarrow \boldsymbol{\theta}_l^T \boldsymbol{F}(\mathbf{s}'_{ij})$ {estimate action-value function}
8: $\hat{\boldsymbol{a}} \leftarrow$ calculated with Equation (5.8)
9: obtain $\{\mathbf{s}_{l_s}, \hat{\boldsymbol{a}}, \mathbf{s}'_{l_s}, R_{l_s}\}$
10: $v_{l_s} = R_{l_s} + \gamma \boldsymbol{\theta}_l^T \boldsymbol{F}(\mathbf{s}'_{ls})$ {state-value function new estimate}
11: **end for**
12: $\boldsymbol{\theta}_{l+1} \leftarrow \text{argmin}_{\boldsymbol{\theta}} \sum_{l_s=1}^{n_s} (v_{l_s} - \boldsymbol{\theta}^T \boldsymbol{F}(\mathbf{s}_{ls}))^2$
13: $l \leftarrow l + 1$
14: **end while**
15: return $\boldsymbol{\theta_l}$

The novelties of the Algorithm 5.7 are continuous action spaces, and the joint work with both state and action-value functions (Lines 6 - 8), while AVI works with discrete, finite action sets and with one of the two functions [27], but not both. Although the outcome of the action-value function learning (Line 8) is independent of the action samples, the state-value function learning (Line 10) depends on the state-samples collected in Line 5, just like discrete action AVI [36].

5.1.4 Discussion

Considering a PBT, we proposed quadratic features vectors, and determined sufficient conditions for which admissible policies presented in Section 5.1.2 transition the system to the goal state obeying the task requirements. Finally, we presented a learning algorithm that learns the parametrization. There are several points that need to be discussed, convergence of the CAFVI algorithm, usage of the quadratic basis functions, and determination of the conditions from Section 5.1.1.

Full conditions under which AVI with discrete actions converges is still an active research topic [27]. It is known that it converges when the system dynamics is a contraction [27]. A detailed analysis of the error bounds for AVI algorithms with finite [93] and continuous [12] actions, finds that the AVI error bounds scale with the difference between the basis functional space and the inherent dynamics of the MDP. The system's dynamics and reward functions determine the MDP's dynamics. We choose quadratic basis functions, because of the nature of the problem we need to solve and for stability. But, basis functions must fit reasonably well into the true objective function, Equation (2.1), determined by the system dynamics and the reward, otherwise CAFVI diverges.

The goal of this chapter is to present an efficient toolset for solving PBTs on a control-affine system with unknown dynamics. Using quadratic basis functions,

Algorithm 5.7 learns the parametrization $\boldsymbol{\theta}$. Successful learning that converges to a $\boldsymbol{\theta}$ with all negative components, produces a controller based on Section 5.1.2 policies that is safe for a physical system and completes the task.

In Section 5.1.1, we introduced sufficient conditions for successful learning. The conditions are sufficient but not necessary, so the learning could succeed under laxer conditions. Done in simulation prior to a physical system control, the learning can be applied when we are uncertain if the system satisfies the criterion. When the learning fails to succeed, the controller is not viable. Thus, a viable controller is possible under laxer conditions verifiable through learning. CAFVI is time efficient so the toolset can be safely and easily attempted first, before more computationally intensive methods are applied. It can be also used to quickly develop an initial value function, to be refined later with another method.

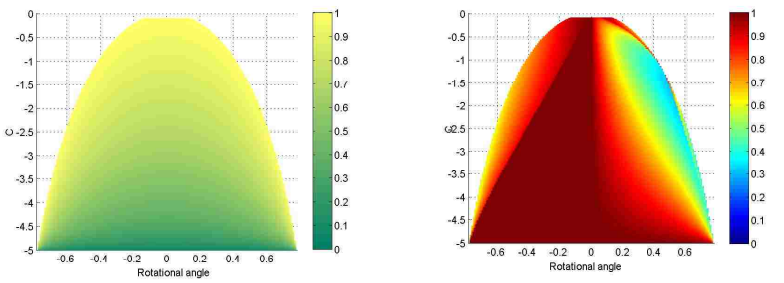
5.2 Results

This section evaluates the CAFVI. We first verify the policy approximations' quality and computational efficiency on a known function in Section 5.2.1, and then we showcase the method's learning capabilities in two case studies: a quadrotor with suspended payload (Section 5.2.2), a multi-agent Rendezvous Task (Section 5.2.3), and a Flying Inverted Pendulum (Section 5.2.4).

In all evaluations, the Convex Sum Policy was calculated using equal convex coefficients $\lambda_i = {d_a}^{-1}$. Discrete and HOOT [81] policies are used for comparison. The discrete policy uses an equidistant grid with 13 values per dimension. HOOT uses three hierarchical levels, each covering one tenth of the action size per dimension and maintaining the same number of actions at each level. All computation was performed using Matlab on a single core Intel Core i7 system with 8GB of

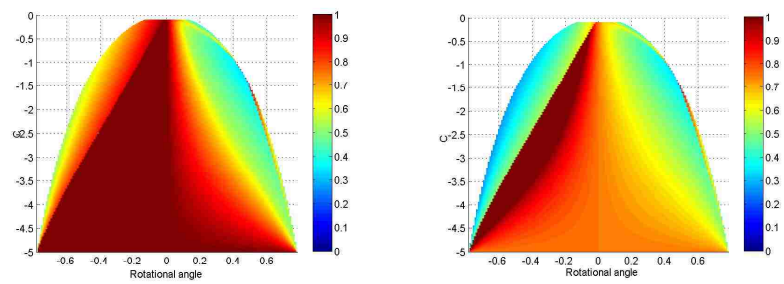
RAM, running the Linux operating system.

5.2.1 Policy Approximation Evaluation



(a) Eccentricity

(b) Manhattan



(c) Axial Sum

(d) Convex Sum

Figure 5.2: Eccentricity of the quadratic functions (a) related to policy approximation gain ratio (b-d) as a function of quadratic coefficient (C) and rotation of the semi-axes.

In Section 5.1.2 we proposed three policy approximations and showed their admissibility. To empirically verify the findings, we examine their behavior on known quadratic functions of two variables, elliptical paraboloids with a maximum. Table 5.2 depicts maximum and minimum values for $\Delta Q(\mathbf{s}, \boldsymbol{\pi}^Q(\mathbf{s}))$ as Q ranges over the class of concave elliptical paraboloids. Since the ΔQ is always positive for all three policies, the empirical results confirm our findings from Proposition 5.1.4 that the policies are admissible. We also see from $\min \Delta \mathbf{a}$ that in some cases Manhattan and Axial Sum Policy make optimal choices, which is expected

Table 5.2: Summary of policy approximation performance. Minimum and maximum of the value gain and the distance from the optimal action.

Method	$\min \Delta Q$	$\max \Delta Q$	$\min \Delta u$	$\max \Delta u$
Manhattan	5.00	168.74	0.00	4.32
Axial Sum	3.40	163.76	0.00	4.37
Convex Sum	3.40	103.42	0.10	4.37

as well. The maximum distance from the optimal action column shows that the distance from the optimal action is bounded.

To further evaluate the policies' quality we measure the gain ratio between the policy's gain and maximum gain on the action-value function ($\mathbf{a}^*$ is optimal action):

$$g_{\pi^Q}(\mathbf{s}) = \frac{Q(\mathbf{s}, \pi^Q(\mathbf{s})) - Q(\mathbf{s}, \mathbf{0})}{Q(\mathbf{s}, \mathbf{a}^*) - Q(\mathbf{s}, \mathbf{0})}.$$

Non-admissible policies have negative or zero gain ratio for some states, while the gain ratio for admissible policies is strictly positive. The gain ratio of one signifies that policy π^Q is optimal, while a gain ratio of zero means that the selected action transitions the system to an equivalent state from the value function perspective. The elliptic paraboloids', $Q(\mathbf{s}, [a_1 a_2]^T) = c_1 a_1^2 + c_2 a_1 u_2 + c_3 a_2^2 + c_4 a_1 + c_5 u_2 + c_6$, isolines are ellipses, and the approximation error depends on the rotational angle of the ellipse's axes, and its eccentricity. Thus, a policy's quality is assessed as a function of these two parameters: the rotational angle α and range of the parameter c, while parameters c_1, c_4, c_5, and c_6 are fixed. Parameter c_2 is calculated such that $c_2 = (c_1 - c_3) \tan 2\alpha$. The eccentricity is depicted in Figure 5.2a, with zero eccentricity representing a circle, and an eccentricity of one representing the ellipse degenerating into a parabola. The white areas in the heat maps are areas where the function is either a hyperbolic paraboloid or a plane, rather than an elliptic paraboloid and has no maximum. Figure 5.2 displays the heat maps of the gain ratios for the Manhattan (Figure 5.2b), Axial Sum (Figure 5.2c), and

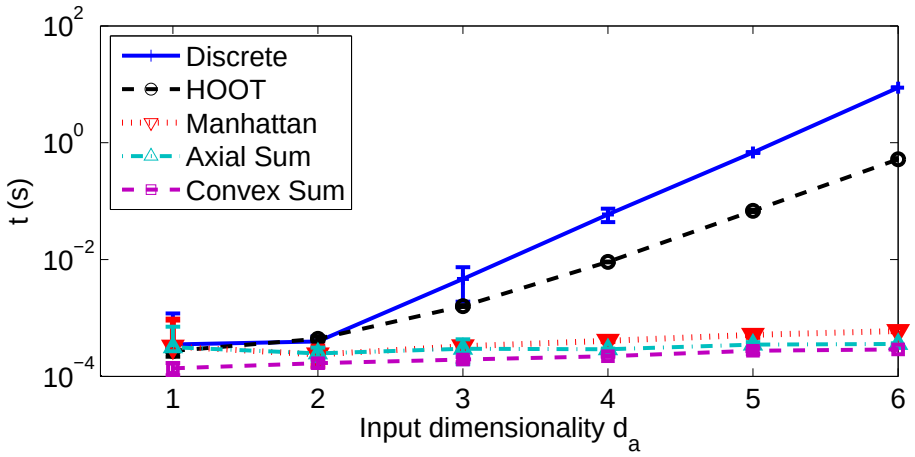


Figure 5.3: Policy approximation computational time per action dimensionality averaged over 10 trials. Vertical bars on (a) represent standard deviation. Comparison of discrete, HOOT, Manhattan, Axial Sum, and Convex Sum policies. The y-axis is logarithmic.

Convex Sum (Figure 5.2d) policies. All policies have strictly positive gain ratio, which gives additional empirical evidence to support the finding in Proposition 5.1.4. Manhattan and Axial Sum perform similarly, with the best results for near-circular paraboloids, and degrading as the eccentricity increases. In contrast, the Convex Sum policy performs best for highly elongated elliptical paraboloids.

Lastly, we consider the computational efficiency of the three policies, and compare the running time of a single decision-making with discrete and HOOT [81] policies. Figure 5.3 depicts the computational time for each of the policies as a function of the action dimensionality. Both discrete and HOOT policies' computational time grows exponentially with the dimensionality, while the three policies that are based on the axial maximums: Manhattan, Axial Sum, and Convex Sum are linear in the action dimensionality, although Manhattan is slightly slower.

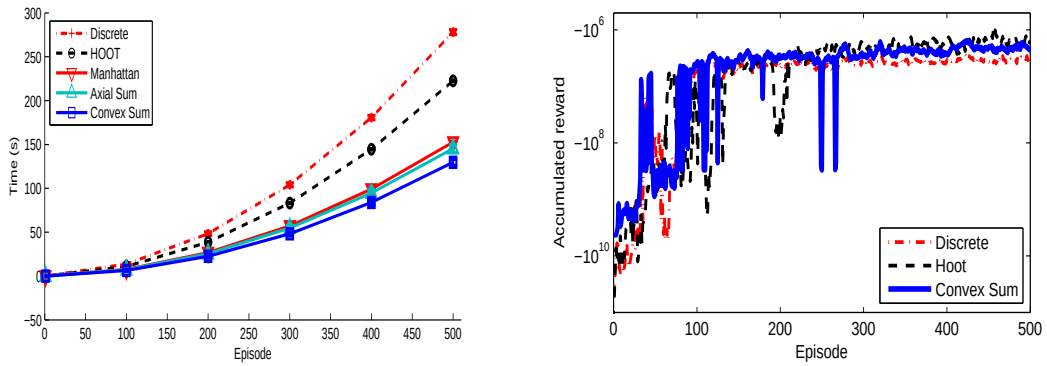
5.2.2 Minimal Residual Oscillations Task

This section applies the proposed methods to the aerial Minimal Residual Oscillations Task (Definition 4.1.4). This task is defined for a UAV carrying a suspended load, and seeks acceleration on the UAV's body, that transports the joint UAV-load system to a goal state with Minimal Residual Oscillations trajectory. We show that the system and its MDP satisfy conditions for Theorem 5.1.2, and will assess the methods though examining the learning quality, the resulting trajectory characteristics, and implementation on the physical system. We compare it to the discrete AVI (Chapter 4) and HOOT (see Chapter 2.3.5), and show that methods presented here solve the task with more precision. Here we use the same problem formulation, MDP setup, and feature vector, Equation (4.2), presented in Section 4.2.1. The only difference is that in this chapter we keep the action space continuous, rather than discretized.

The satisfies the form of Equation (4.2), and because the learning produces $\boldsymbol{\theta}$ with all negative components, all conditions for Theorem 5.1.2 are satisfied including the drift, Equation (5.7).

The time-to-learn is presented in Figure 5.4a. The standard deviation is between 1% and 10% of the mean. The time to learn increases polynomially with the iterations because the number of samples in each iteration grows linearly. The axial maximum policies perform an order of magnitude faster than the discrete and HOOT policies. To assess learning with Algorithm 5.7 using Manhattan, Axial Sum, and Convex Sum policies, we compare to learning using the greedy discrete policy and HOOT. Figure 5.4b shows the learning curve, over number of iterations. After 300 iterations all policies converge to a stable value. All converge to the same value, but discrete learning that converges to a lower value.

Finally, inspection of the learned parametrization vectors confirms that all the



(a) Time to learn

(b) Learning curve (logarithmic)

Figure 5.4: Learning results for Manhattan, and Axial Sum, and Convex Sum, compared to discrete greedy, and HOOT policies averaged over 10 trials. Vertical bars on (a) represent standard deviation. Learning curves for Manhattan and Axial Sum are similar to Convex Sum and are omitted from (b) for better visibility.

components are negative, meeting all needed criteria for Theorem 5.1.2. This means that the equilibrium is asymptotically stable, for admissible policies, and we can generate trajectories of an arbitrary length.

Next, we plan trajectories using the learned parametrization over the 100 trials for the three proposed policies and compare them to the discrete and HOOT policies. We consider Minimal Residual Oscillations Task complete when $\|\boldsymbol{p}\| \leq 0.010\,\mathrm{m}$, $\|\boldsymbol{v}\| \leq 0.025\,\mathrm{m\,s^{-1}}$, $\|\boldsymbol{\eta}\| \leq 1°$, and $\|\dot{\boldsymbol{\eta}}\| \leq 5\,°/\mathrm{s}$. This is a stricter terminal set than the one previously used in Chapter 4. The action limits are $-3\,\mathrm{m\,s^{-2}} \leq u_i \leq 3\,\mathrm{m\,s^{-2}}$, for $i \in 1,2,3$. The discrete and HOOT policies use the same setup described in Section 5.2. The planning occurs at 50 Hz. We compare the performance and trajectory characteristics of trajectories originating 3 m from the goal state. Table 5.3 presents results of the comparison. Manhattan, Axial Sum, and HOOT produce very similar trajectories, while Convex Sum generates slightly longer trajectories, but with the best load displacement characteristics. This is because the Convex Sum takes a different approach and selects smaller

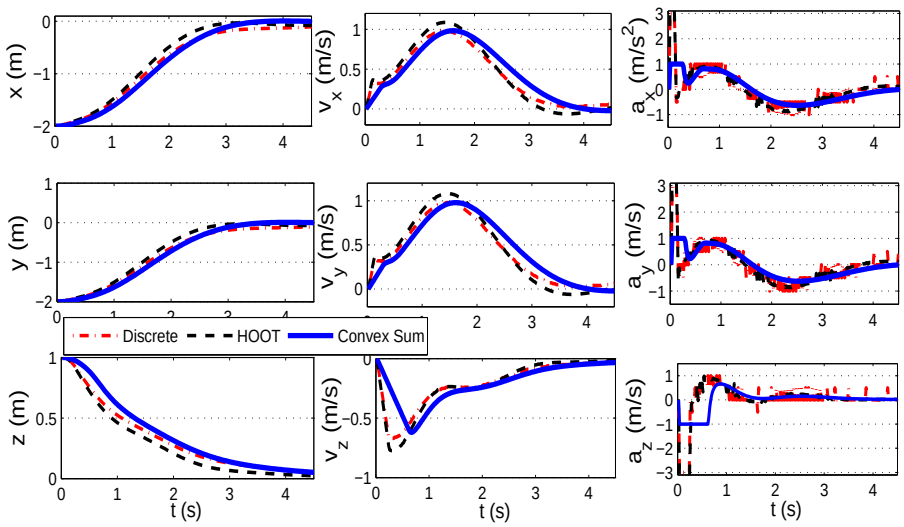
actions, resulting in smoother trajectories. The Convex Sum method plans the 9 s second trajectory in 0.14 s, over 5 times faster than the discrete planning, and over 3 times faster than HOOT. Finally, 30% of the discrete trajectories are never able to complete the task. This is because the terminal set is too small for the discretization. In other words, the discretized policy is not admissible. Examining the simulated trajectories in Figure 5.5 reveals that Convex Sum indeed selects a smaller action, resulting in a smoother trajectory (Figure 5.5a) and less swing (Figure 5.5b). HOOT, Manhattan, and Axial Sum, produce virtually identical trajectories, while the discrete trajectory has considerable jerk, absent from the other trajectories.

Lastly, we experimentally compare the learned policies using the same set up described in Section 4.3.2. HOOT and Axial Sum resulted in similar trajectories, while Manhattan's trajectory exhibited the most deviation from the planned trajectory (Figure 5.6). The Convex Sum trajectory is the smoothest. Table 5.4 quantifies the maximum load swing and the power required to produce the load's motion from the experimental data. Convex Sum policy generates experimental trajectories with the best load swing performance, and with load motion that requires close to three times less energy to generate[1].

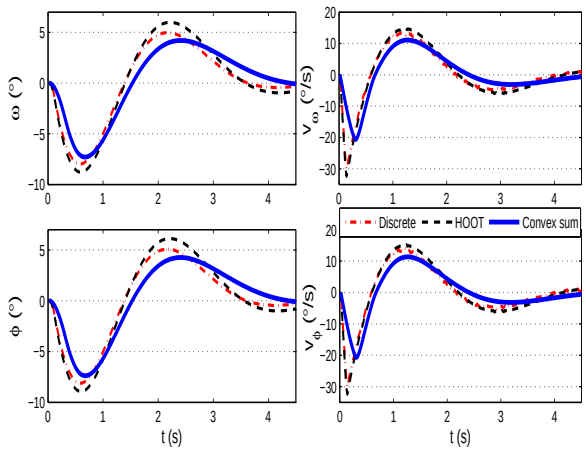
5.2.3 Rendezvous Task

The Rendezvous Cargo Delivery Task, or simply Rendezvous Task (Figure 5.7a), is a multi-agent variant of the time-sensitive Minimal Residual Oscillations Task. It requires an UAV carrying a suspended load to rendezvous with a ground-bound robot to hand over the cargo. The cargo might be a patient airlifted to a hospital and then taken by a moving ground robot for delivery to an operating room for

[1] A video of the experiments can be found at: `https://cs.unm.edu/amprg/People/afaust/afaustActa.mp4`.



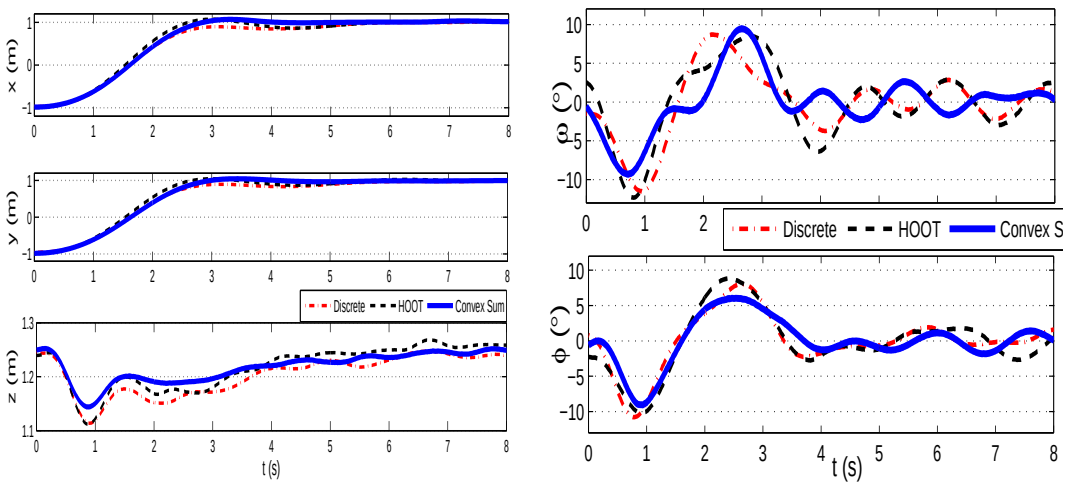
(a) Quadrotor trajectory



(b) Load trajectory

Figure 5.5: Comparison of simulated Minimal Residual Oscillations Trajectories created with Convex Sum versus trajectories created with discrete greedy and HOOT policies. (Trajectories for Manhattan and Axial Sum are similar to Convex Sum and are omitted for better visibility.)

surgery. Thus, the UAV trajectory must be with Minimum Residual Oscillations. The rendezvous location and time are not known a priori, and the two heterogeneous agents must plan jointly to coordinate their speeds and positions. The two



(a) Quadrotor trajectory

(b) Load trajectory

Figure 5.6: Comparison of experimental Minimal Residual Oscillations Task trajectories created with Convex Sum versus trajectories created with discrete greedy and HOOT policies. (Trajectories for Manhattan and Axial Sum are similar to Convex Sum and are omitted for better visibility.)

robots have no knowledge of the dynamics and each others' constraints. The task requires minimization of the distance between the load's and the ground robot's location, the load swing minimization, and minimization for the agents' velocities, while completing the task as fast as possible.

Table 5.3: Summary of trajectory characteristics over 100 trials. Means (μ) and standard deviations (σ) of time to reach the goal, final distance to goal, final swing, maximum swing, and time to compute the trajectory. Best results are highlighted.

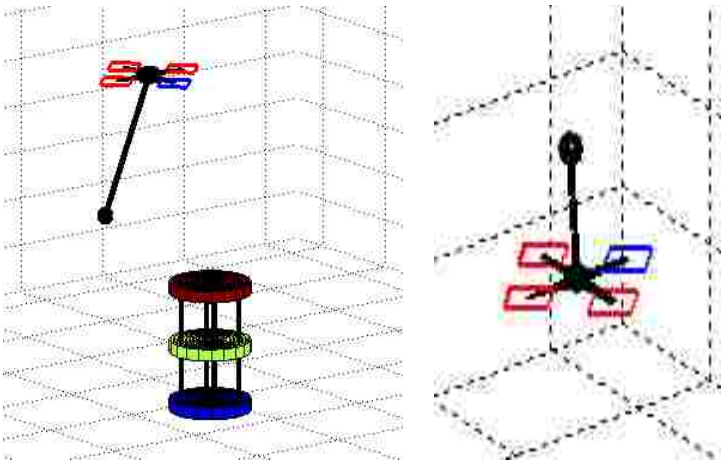
Method	Percent completed	t (s)		$\| p \|$ (cm)		$\| \eta \|$ (°)		max $\| \eta \|$ (°)		Comp. time (s)	
		μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
Discrete	70.00	10.81	3.12	0.98	0.33	0.16	0.14	11.96	1.63	0.81	0.23
HOOT	100.00	**8.49**	**1.33**	**0.83**	0.27	0.18	0.20	12.93	1.49	0.48	0.07
Manhattan	100.00	8.66	1.68	0.89	0.19	0.15	0.16	12.24	1.58	0.24	0.05
Axial Sum	100.00	8.55	1.56	0.85	0.22	0.20	0.18	12.61	1.55	0.17	0.03
Convex Sum	100.00	9.61	1.62	0.97	**0.07**	**0.03**	**0.06**	**9.52**	**1.29**	**0.14**	**0.02**

Table 5.4: Summary of experimental trajectory characteristics. Maximum swing and energy needed to produce load oscillations. Best results are highlighted.

Method	$max \parallel \eta \parallel (^\circ)$	Energy (J)
Discrete	15.21	0.0070
HOOT	15.61	0.0087
Manhattan	15.95	0.0105
Axial Sum	14.20	0.0086
Convex Sum	**12.36**	**0.0031**

The quadrotor with the suspended load is modeled as in Section 5.2.2, while a rigid body constrained to two DOF in a plane models the ground-based robot. The joint state space is a 16-dimensional vector: the 10-dimensional state space of the quadrotor (Section 5.2.2), and the position-velocity space of the ground robot. The action is 5-dimensional acceleration to the quadrotor's and ground robot's center of masses. The ground robot's acceleration constraints are lower than quadrotors.

Applying Algorithm 5.7 with Convex Sum Policy, the system learns the state-value function parametrization $\boldsymbol{\theta}$ that is negative definite. Figure 5.8 shows both



(a) Rendezvous Task (b) Flying Inverted Pendulum

Figure 5.7: PBT examples.

robots two seconds in the trajectory. The comparison of simulated trajectories created with the Convex Sum and HOOT policies is depicted in Figure 5.9. In 0.12 s Convex Sum Policy finds a 8.54 s trajectory that solves the task. HOOT policy fails to find a suitable trajectory before reaching the maximum trajectory duration, destabilizes the system, and terminates after 101.44 seconds. The discrete policy yields similar results as HOOT. This is because the action needed to solve the task is smaller than the HOOT's setup, and the system begins to oscillate. The rendezvous point produced with Convex Sum Policy is between the robots' initial positions, closer to the slower robot, as expected (Figure 5.9a). The quadrotor's load swing is minimal (Figure 5.9b). The absolute accumulated reward collected while performing the task is smooth and steadily making progress, while the accumulated reward along HOOT trajectory remains significantly lower (Figure 5.9c)[2]. The Rendezvous Task simulation shows that the proposed methods are able to solve tasks that previous methods are unable to because the Convex Sum Policy is admissible.

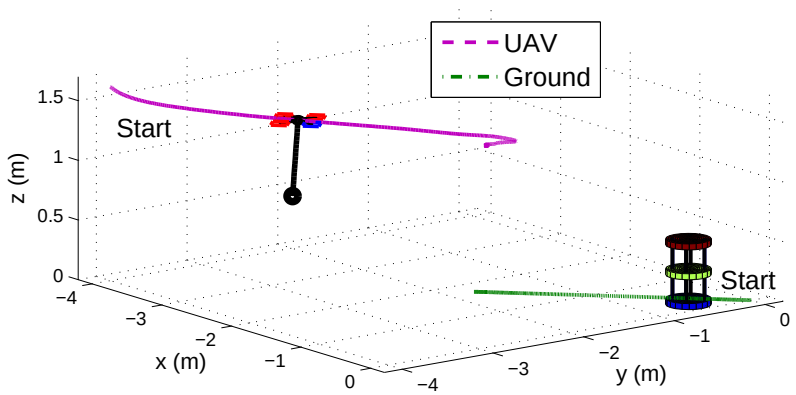


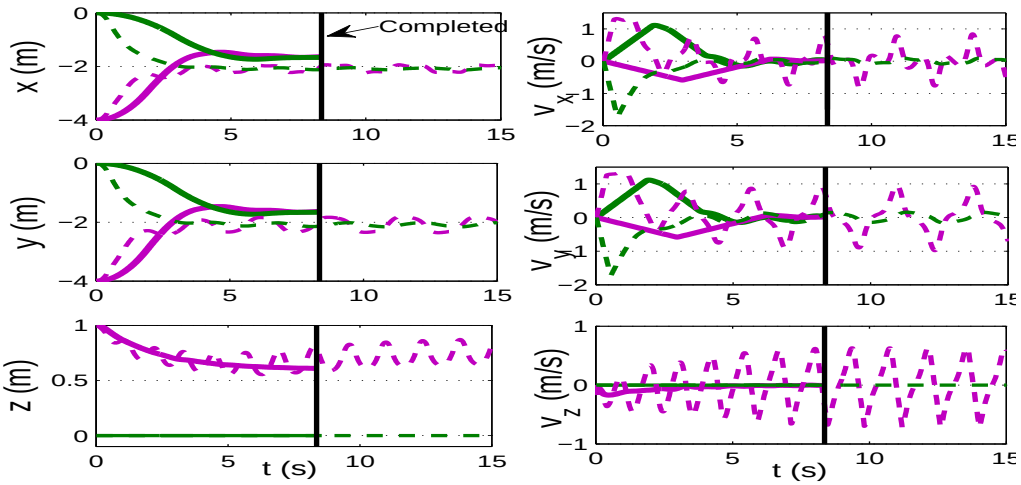
Figure 5.8: Cargo-bearing UAV and a ground-based robot rendezvous at 2 s.

[2]A video of the simulation can be found at: `https://cs.unm.edu/amprg/People/afaust/afaustActa.mp4`.

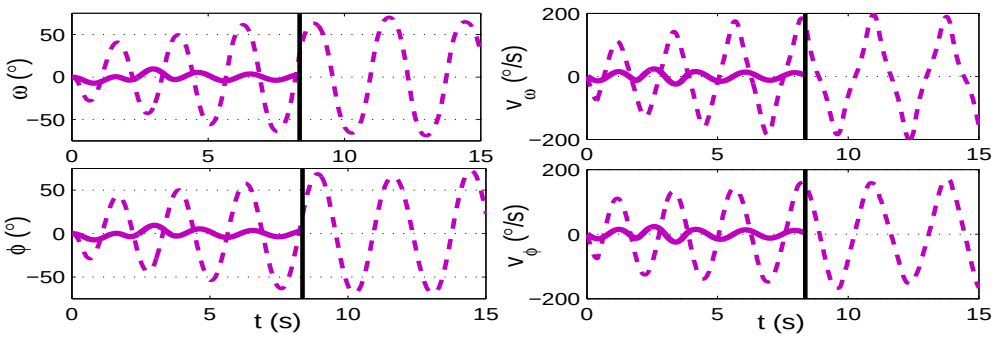
5.2.4 Flying Inverted Pendulum

The third task we consider is Flying Inverted Pendulum. It consists of a quadrotor-inverted pendulum system in a plane (see Figure 5.7b). The goal is to stabilize the pendulum and keep it balanced as the quadrotor hovers. The section [3] is based on portions of [42]. We decompose the Flying Inverted Pendulum task in two subtasks, Initial Balance and Balanced Hover. Initial Balance, developed by Faust, places pendulum very close to upright position. Balanced Hover, developed by Figueroa, slows down the quadrotor to a hover while maintaining the upright inverted pendulum position. We learn both tasks with CAFVI (Algorithm 5.7), and generate trajectories with the Convex Sum Policy. The joint controller, Flying Inverted Pendulum, developed by Faust, sequentially combines the two subtasks to solve the flying inverted pendulum. It produces a trajectory that starting for an arbitrary initial inverted pendulum displacement of up to 20° ends with the UAV hovering and maintaining the inverted pendulum minimally displaced from the upright position.

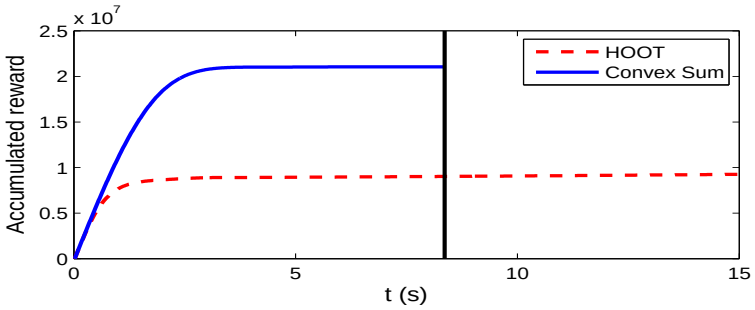
[3] © 2014 IEEE. Portions of this section are reprinted, with permission, from Rafael Figueroa, Aleksandra Faust, Patricio Cruz, Lydia Tapia, and Rafael Fierro, "Reinforcement Learning for Balancing a Flying Inverted Pendulum," The 11th World Congress on Intelligent Control and Automation, July 2014.



(a) Robot trajectories



(b) Load trajectory



(c) Accumulated reward

Figure 5.9: Comparison of simulated Rendezvous Task trajectories created with Convex Sum Policy to trajectories created with discrete greedy and HOOT policies. Green solid - Convex Sum ground; Purple solid - Convex Sum aerial; Green dashed - HOOT ground; Purple dashed - HOOT aerial.

The MDP's state space S is a eight-dimensional vector of Cartesian coordinates with the origin in the quadrotor's center of the mass belonging to the plane orthogonal to the gravity force, $\mathbf{s} = [x\ y\ a\ b\ \dot{x}\ \dot{y}\ \dot{a}\ \dot{b}]^T$; x and y, also combined into $\boldsymbol{r} = [a\ b]^T$, are the quadrotor's position in x and y directions, while $\dot{x}$ and $\dot{y}$ are its linear velocity. a and b, combined into $\boldsymbol{p} = [a\ b]^T$, are the projections of the pendulum's center of the mass onto the xy-plane. The action space is a two dimensional acceleration vector $\mathbf{a} = \begin{bmatrix} \ddot{x}_d & \ddot{y}_d \end{bmatrix}^T$. on the quadrotor's center of the mass in a plane horizontal to the ground, $\ddot{x}_d, \ddot{y}_d \in \mathbb{R}$. The maximum acceleration is $5\,\mathrm{m\,s^{-2}}$. The reward is one when the target zone is reached, and zero otherwise. The simulator used is a linearized model of a full dynamics of a planar flying inverted pendulum and can be found in [42]. This system is control-affine.

Given an initial displacement of the pendulum center of mass $\mathbf{p}_0 = [a_0\ b_0]^{\mathrm{T}}$, our goal is to find an optimal action $\mathbf{a}$ such that the pendulum is balanced on top of the quadrotor, i.e., $\mathbf{p}$ and $\dot{\mathbf{p}}$ tends to zero, and the quadrotor reaches a hovering state, i.e., $\dot{\mathbf{r}}$ also tends to zero. Thus, the Flying Inverted Pendulum is complete when the system comes to rest. This is when $\mathbf{p} = \dot{\mathbf{p}} = \dot{\mathbf{r}} = \mathbf{0}$. From the practical perspective, we consider the task done when the state norm is sufficiently small. Initial Balance Task raises the pendulum to an upright position without regard to the quadrotor state. The task is completed when

$$\mathbf{p} < \epsilon_p,\ \dot{\mathbf{p}} < \epsilon_{\dot{p}}, \tag{5.12}$$

for small positive constants ϵ_p and $\epsilon_{\dot{p}}$. The second subtask, Task assumes that Initial Balance was completed and the initial conditions satisfy Equation (5.12). The task requires quadrotor to reduce speed to hover while it maintains the minimal inverted pendulum displacement. It is completed when

$$\mathbf{p} < \epsilon_p,\ \dot{\mathbf{p}} < \epsilon_{\dot{p}}, \dot{\mathbf{r}} < \epsilon_{\dot{r}}, \tag{5.13}$$

for small positive constants ϵ_p, $\epsilon_{\dot{p}}$, and $\epsilon_{\dot{r}}$. Note although, that the completion

criteria for the Balanced Hover Task is the same as for Flying Inverted Pendulum Task, the latter has less restrictive initial conditions.

The features for the Initial Balance Task are squares of the pendulum's position and velocity relative to the goal upright position, $\mathbf{F}_B(\mathbf{s}) = [\|\mathbf{p}\|^2 \ \|\dot{\mathbf{p}}\|^2]^{\mathrm{T}}$. The Balanced Hover Task has an additional feature of a square of the quadrotor's velocity, $\mathbf{F}_BH(\mathbf{s}) = [\|\mathbf{p}\|^2 \ \|\dot{\mathbf{p}}\|^2 \ \|\dot{r}\|^2]^{\mathrm{T}}$. Here the subscript B, BH denote that just the Initial Balance amd Balanced Hover tasks are under consideration respectively.

Algorithm 5.8 depicts learning flying inverted pendulum task. We generate $\boldsymbol{\theta}_B$ and $\boldsymbol{\theta}_{BH}$ with Algorithm 5.7. We learn using Monte Carlo simulation for a fixed number of trials. Upon generating family of policies, the fittest one is selected for the inverted pendulum controller. The fittest policy reaches the completion state fastest. Algorithm 5.7 provides a satisfactory answer for a high initial displacement when no consideration is given to the condition that $\dot{\mathbf{r}}$ reaches zero. When the initial displacement is small, the requirement of $\dot{\mathbf{r}} \rightarrow \mathbf{0}$ can be added. For this reason, we used both $\boldsymbol{\theta}_B$ and $\boldsymbol{\theta}_{BH}$ to implement control for Flying Inverted Pendulum. Algorithm 5.9 summarizes the proposed control technique. The controller is initially configured for Initial Balance task. It selects a control action according to the Convex Sum Policy, Equation (5.10). When the completion condition for Initial Balance task is met, the controller switches to the parameters for Balanced Hover task. On the other hand, if the controller state is set to 'Balanced Hover', the controller invokes Convex Sum Policy with the parameters for the 'Balanced Hover' task. The output is the state of the controller and the action to perform.

To evaluate the Flying Inverted Pendulum we examine each of the subtasks, Initial Balance and Balanced Hover separately, as well as the joint Flying Inverted Pendulum controller from Algorithm 5.9. The evaluation goals are to determine the computational speed, region of attraction, and noise tolerance. All simulations

Algorithm 5.8 Learning to fly inverted pendulum.

Input: γ, features $\mathbf{F}_B$, $\mathbf{F}_{BH}$, R_B, R_{BH},action space dimensionality $d_a = 2$, number of Monte Carlo simulations mc

1: {Learning Initial balance with Monte Carlo simulation}
2: **for** i=1:mc **do**
3: $\quad \boldsymbol{\theta}_{B,i} \leftarrow \text{CAFVI}(\mathbf{F}_B, R_B, \gamma, d_a)$
4: **end for**
5: $\boldsymbol{\theta}_{B,i} \leftarrow$select fittest $\boldsymbol{\theta}_{B,i}$
6: {Learning Balanced Hover with Monte Carlo simulation}
7: **for** i=1:mc **do**
8: $\quad \boldsymbol{\theta}_{BH,i} \leftarrow \text{CAFVI}(\mathbf{F}_{BH}, R_{BH}, \gamma, d_a)$
9: **end for**
10: $\boldsymbol{\theta}_{BH,i} \leftarrow$select fittest $\boldsymbol{\theta}_{BH,i}$

Output: Vectors $\boldsymbol{\theta}_B, \boldsymbol{\theta}_{BH}$

are performed in Matlab 2012a running Windows 7 with Pentium Dual-Core CPU and 4Gbs of RAM. The pendulum point mass is located at $L = 0.5\,\text{m}$ from the center of mass of the quadrotor. The pendulum has a mass $m_p = 0.1\,\text{kg}$. An offset of the pendulum mass with respect to the quadrator equal to $a = 0.2\,\text{m}$ represents an angular displacement of approximately 20° with respect to the upright equilibrium position. The terminating condition for Initial Balance controller is $\|\boldsymbol{p}\| \leq 0.01\,\text{m}$ and $\|\dot{\boldsymbol{p}}\| \leq 0.01\,\text{m}\,\text{s}^{-1}$, while the terminating condition for the Balanced Hover and Flying Inverted Pendulum polices is $\|\boldsymbol{p}\| \leq 0.05\,\text{m}$, $\|\dot{\boldsymbol{p}}\| \leq 0.05\,\text{m}\,\text{s}^{-1}$, and $\|\dot{\boldsymbol{r}}\| \leq 0.05\,\text{m}\,\text{s}^{-1}$.

Algorithm 5.8 results in parametrization vectors $\boldsymbol{\theta}_B = [-86.6809 - 0.3345]^T$ and $\boldsymbol{\theta}_{BH} = 10^6[-1.6692, -0.0069, 0.0007]^T$. Both approximated V-functions, calculated with Equation (4.2), are much more sensitive to changes in pendulum's position than changes in velocity. The similar preference is seen in the Balanced Hover

Algorithm 5.9 Flying inverted pendulum controller.

Input: $\boldsymbol{\theta}_B$, $\boldsymbol{\theta}_{BH}$, features $\boldsymbol{F}_B$, $\boldsymbol{F}_{BH}$, small positive constants ϵ_p, $\epsilon_{\dot{p}}$, dimension of the action space $d_a = 2$, controller state

observe initial state $\mathbf{s}_0$
if controller state is 'Initial Balance' **then**
  {Initial balance}
  $\mathbf{a} \leftarrow$ convexSumPolicy$(\mathbf{F}_B, \boldsymbol{\theta}_B, \mathbf{s}_0)$ Equation (5.10)
  if $\|\mathbf{p}_0\| \leq \epsilon_p \wedge \|\dot{\mathbf{p}}_0\| \leq \epsilon_{\dot{p}}$ **then**
    {switch controller state}
    controller state $\leftarrow$ 'Balanced Hover'
  end if
else
  {Balanced Hover}
  $\mathbf{a} \leftarrow$ convexSumPolicy$(\boldsymbol{F}_{BH}, \boldsymbol{\theta}_{BH}, \mathbf{s}_0)$ Equation (5.10)
end if

Output: $\mathbf{a}$, controller state

with $\boldsymbol{\theta}_{BH}$. Here, only when the position and velocity of the inverted pendulum are close to the upright position, the controlled reduces the quadrotor's speed.

To examine the policies' characteristics, we randomly selected 100 initial system state conditions, $\mathbf{s}$, for each policy: Initial Balance with no noise, Initial Balance with 5% of randomly added noise, Balanced Hover with no noise, Balanced Hover with 2% of noise, Flying Inverted Pendulum with no noise, and Flying Inverted Pendulum with 5% of noise. The ranges for initial conditions per policy are presented in Table 5.5. For all policies, and all the initial conditions resulted in reaching the task terminal conditions. That means that the initial conditions in Table 5.5 are within the policies regions of attraction.

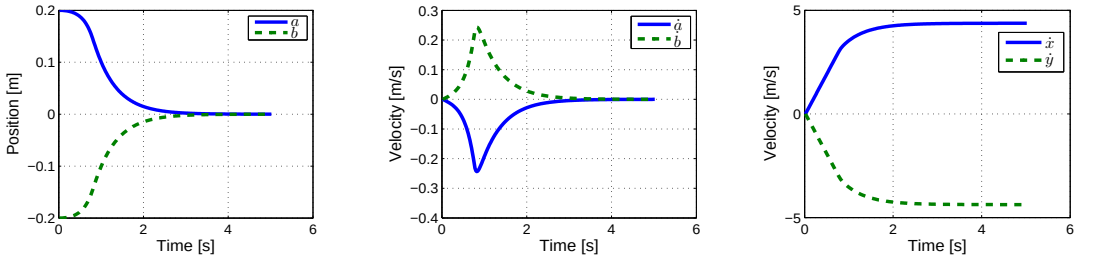
Table 5.6 shows the results trajectory characteristics upon completion. t_f is the

Table 5.5: Initial conditions for Flying Inverted Pendulum Task for the 100 sample selection.

Controller	$\dot{\mathbf{r}} = [\dot{x}\ \dot{y}]^{\mathrm{T}}$ (m s^{-1})	$\mathbf{p} = [a\ b]^{\mathrm{T}}$ (m)
Initial Balance	$\dot{x} = \dot{y} = 0$	$a, b \in [-0.2, 0.2]$
Balanced Hover	$\dot{x}, \dot{y} \in [-5, 5]$	$a, b \in [-0.01, 0.01]$
Flying Inverted Pendulum	$\dot{x} = \dot{y} = 0$	$a, b \in [-0.2, 0.2]$

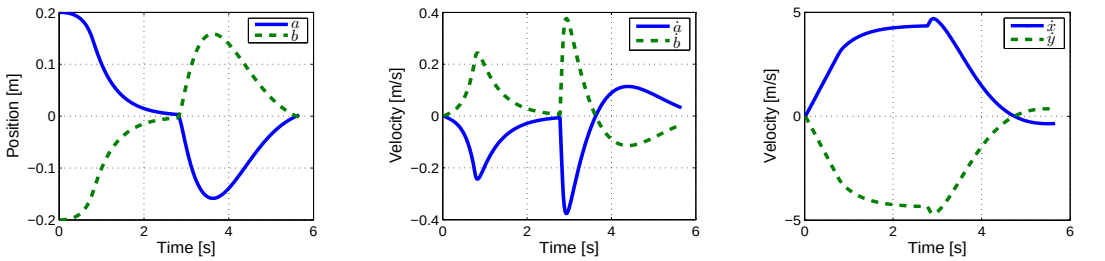
time to achieve the task measured in system time, t_c is the computation time, $\mathbf{p}_f$ is the final displacement of the pendulum, $\dot{\mathbf{p}}_f$ is the final velocity of the pendulum, and $\dot{\mathbf{r}}_f$ is the final velocity of the quadrotor. We note that the system time t_f is order of the magnitude higher than the computational time t_c, rendering our method real-time capable. Initial Balance policy brings the inverted pendulum to upright position efficiently for initial displacement of up to 20°, but the quadrotor's velocity remains constant at the completion (Table 5.6). This is expected due to the policy design. Balanced Hover, on the other hand, reduces quadrotor speed to zero, while maintaining the upright pole position and its small velocities. The downside is that it is capable in doing so only for the small initial pole displacements. Finally, the Flying Inverted Pendulum policy handles large initial pole displacements, and brings the pole to the upright position with small (less than 0.05 m s^{-1}) velocities, and reduces the quadrotor speed to less than 0.45 m s^{-1}. Lastly, the Table 5.6 shows that all three policies are capable of handling some level of random noise. This is important because it shows that the controllers have some tolerance to random disturbances and unmodeled system dynamics without impacting their performance significantly.

To visualize the trajectories Figures 5.10 and 5.11 show the trajectories created with Initial Balance and Flying Inverted Pendulum policies. Pendulum positions and velocity, and quadrotor speed are depicted. Figure 5.10c depicts the trajectory created with Initial Balance controller, starting at $\boldsymbol{p} = [0.2\ -0.2]^T$ m. Figure 5.10



(a) Pendulum position (b) Pendulum velocity (c) Quadrotor velocity sum

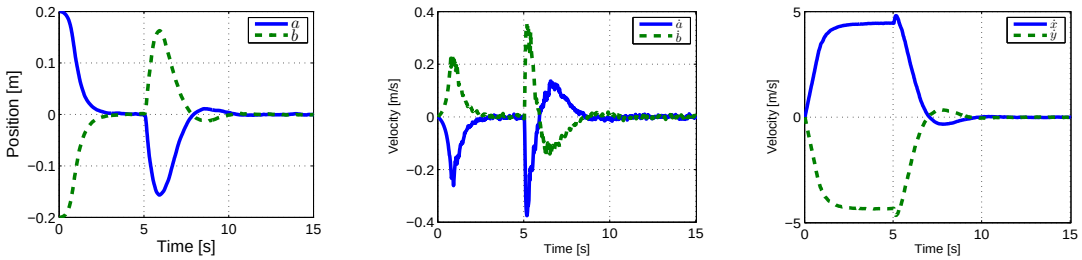
Figure 5.10: Trajectory created with Initial Balance policy.



(a) Pendulum position (b) Pendulum velocity (c) Quadrotor velocity sum

Figure 5.11: Trajectory created with Flying Inverted Pendulum policy..

shows that the Initial Balance policy brings the pole to the upright position with minimal velocity (Figures 5.10a and 5.10b), while the quadrotor velocity becomes



(a) Pendulum position (b) Pendulum velocity (c) Quadrotor velocity

Figure 5.12: Trajectory created with Flying Inverted Pendulum policy with 5% of noise.

constant (Figure 5.10c). The quadrotor's residual velocity is considered in the Flying Inverted Pendulum policy (Figure 5.11b). The Flying Inverted Pendulum starts with Initial Balance and reduces the position and velocity of the pendulum, once its terminal conditions are achieved, the Balanced Hover policy reduces the quadrotor velocity rapidly to zero while returning the pendulum to the upright position with minimal velocity. Figure 5.12 shows the results obtained using the policy Flying Inverted Pendulum while 5% of random noise is being added. We see that despite noise in the pendulum's velocity in Figure 5.12b, the pendulum stays upright and the quadrotor's velocity tends around zero (Figure 5.12c). Another view of the trajectory created with Flying Inverted Pendulum controller is shown in Figure 5.13. The lighter snapshots occur further in time. The quadrotor starts moving fast to perform initial balance, and then starts to slow down, as evidenced by more dense snapshots[4].

5.3 Conclusions

In this chapter we extended PEARL by developing a method for learning control of continuous non-linear motion systems through combined learning of state-value and action-value functions. Negative definite quadratic state-value functions imply quadratic, concave action-value functions. That allowed us to approximate policy as a combination of its action-value function maximums on the axes, which we found through interpolation between observed samples. These policies are admissible, consistent, and efficient. Lastly, we showed that a quadratic, negative definite state-value function, in conjunction with admissible policies, are sufficient conditions for the system to progress to the goal while minimizing given preferences.

[4]The simulation movie is available at `http://marhes.ece.unm.edu/index.php/Quadip-cafvi`

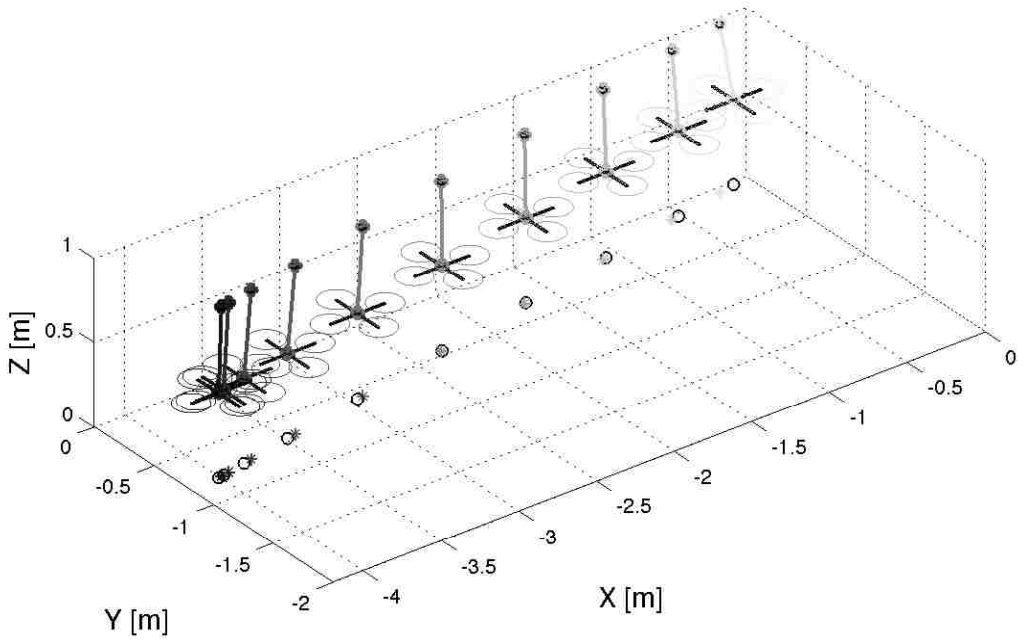


Figure 5.13: Flying Inverted Pendulum trajectory snapshots © 2014 IEEE. Less saturated snapshots occur further in time. The relative position of the planar position of the quadrotor and pendulum are tracked in the environment as a circle ($\circ$) and asterisk ($*$) respectively.

control-affine systems for PBTs with RL.

Table 5.6: Flying inverted pendulum simulation results for 100 random initial conditions for the three controllers with and without noise. Policy name, noise level, completion averages, minimums, and maximums for system time (t_f), computational time (t_c), pendulum displacement ($\|\mathbf{p}_f\|$), pendulum velocity magnitude ($\|\dot{\mathbf{p}}_f\|$), and quadrotor velocity magnitude ($\|\dot{\mathbf{r}}_f\|$)

Controller	Noise	t_f [s]			t_c (s)			$\|\mathbf{p}_f\|$ (mm)			$\|\dot{\mathbf{p}}_f\|$ (mm s^{-1})			$\|\dot{\mathbf{r}}_f\|$ (mm s^{-1})		
	%	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max
Initial	0	0.67	0.04	1.32	0.03	0.00	0.06	54	26	98	82	15	96	1622	25	4121
Balancing	5	0.68	0.04	1.54	0.03	0.00	0.09	53	27	99	85	15	105	1593	24	5545
Balanced	0	5.55	5.48	5.60	0.19	0.18	0.22	0	0	0	2	1	4	24	10	40
Hover	2	2.86	0.22	5.16	0.10	0.01	0.17	2	1	2	32	7	49	293	33	495
Flying	0	4.50	1.06	5.38	0.17	0.05	0.28	2	1	2	20	7	42	165	35	438
Inv. Pend.	5	4.51	0.76	5.50	0.17	0.04	0.32	2	1	2	21	4	46	173	25	446

Chapter 6

PEARL for Stochastic Dynamical Systems with Continuous Actions

Real-world conditions pose many challenges to physical robots. One such challenge is the introduction of stochastic disturbances that can cause robotic drift. For example, wind, imperfect models, or measurement inaccuracies are all possible disturbances [16]. These disturbances pose a particular challenge to modern robotics due to the expectation that robots automatically learn and perform challenging tasks. The disturbances, along with complicated, nonlinear system dynamics, make traditional solutions, e.g., adaptive and robust control modeling, which solves this problem using full knowledge of the system dynamics, difficult if not intractable [59].

This chapter presents a PEARL's extension for control-affine systems with stochastic disturbances based on [38]. The contributions are:

- LSAPA, an adaptive policy approximation that modifies of the Axial Sum Policy to adapt to variable disturbances (Section 6.1.2), and

- SCAFVI, an extension of the CAFVI algorithm to use adaptive policy approximation (Section 6.1.3).

The methods are verified on:

- Minimal Residual Oscillations Task (Section 6.2.2),
- Rendezvous Task (Section 6.2.3), and
- Flying Inverted Pendulum task (Section 6.2.4)

Recall from Section 2.3.7 that batch reinforcement learning used in PEARL separates learning and planning into two distinct phases. During the learning phase, the agent interacts with a simulator and learns feature vector weights that determine the policy. In the motion planning phase, the learning stops and the learned weights are used to generate trajectories for a physical system. The lack of learning in the planning phase guarantees the stability. The downside is that there is no adaptation during the planning. The methods presented in this chapter allow for adaptive MP.

The key extension from Chapter 5 is the use of least squares linear regression in lieu of interpolation, Equation (5.8), to estimate near-optimal action on an axis. This extension allows us to apply the method to non-zero mean disturbances with the only limit being imposed by the system's physical limits, and demonstrate it on three different problems.

6.1 Methods

Our goal is to learn to a perform and plan execution of a PBT on a kinematic (control-affine) system within PEARLframework (Figure 3.1) in the presence of an

external bias (stochastic disturbance), such as wind. A common stochastic noise model, a Gaussian process defined by mean and variance, is used [16]. We present the problem formulation in Section 6.1.1, develop LSAPA in Section 6.1.2. Finally, the policy is used for the task learning in Section 6.1.3.

6.1.1 Problem Formulation

The problem formulation is similar to the one used in Section 3.1.1, but instead uses a stochastic control-affine system. Thus, we model a robot as a discrete time, control-affine system with stochastic disturbance, $\mathbf{D} : S \times A \to S$,

$$\mathbf{D}_s : \quad \mathbf{s}_{k+1} = \boldsymbol{f}(\mathbf{s}_k) + \boldsymbol{g}(\mathbf{s}_k)(\mathbf{a}_k + \boldsymbol{\eta}_k). \tag{6.1}$$

States $\mathbf{s}_k \in S \subseteq \mathbb{R}^{d_s}$ belong to the position-velocity space and the control action is acceleration, $\mathbf{a}_k \in A \subseteq \mathbb{R}^{d_a}$. The action space is a compact set containing origin, $\mathbf{0} \in A$. Lipschitz continuous function $\boldsymbol{g} : S \to \mathbb{R}^{d_s} \times \mathbb{R}^{d_a}$ is regular outside the origin, $\mathbf{s}_k \in S \setminus \{\mathbf{0}\}$. The drift $\boldsymbol{f} : S \to \mathbb{R}^{d_s}$, is bounded and Lipschitz continuous. The non-deterministic term, $\boldsymbol{\eta}_k$, is a white Gaussian noise with a known distribution $\mathcal{N}(\mu_{\boldsymbol{\eta}_k}, \sigma_{\boldsymbol{\eta}_k})$; it acts as an additional and unpredictable external force on the system. Time step k is omitted from the notation when possible. We use the system and task assumptions from Chapter 5; the system is controllable [59] and our goal is to learn PBT that takes the system to its origin in a timely-manner while balancing, along the trajectory, constraints given by $\boldsymbol{C}\mathbf{s} = \{\boldsymbol{\alpha}_1(\mathbf{s}), ..., \boldsymbol{\alpha}_{d_g}(\mathbf{s})\}$.

To formulate the problem in RL terms, we define a discrete time, *stochastic*, first-order MDP with continuous state and action spaces,

$$\mathcal{M} : \ (S, A, \mathbf{D}_s, R) \tag{6.2}$$

Observed state reward is $R : X \to \mathbb{R}$; $\mathbf{D}_s$ is a an unknown system transition function with form given in Equation (6.1). Solution to a MDP is a policy $\pi : S \to$

Table 6.1: Summary of chapter-specific key symbols and notation.

Symbol	Description
$\boldsymbol{C}\mathbf{s}$	Preferences to minimize
$\boldsymbol{\eta}_k\, \mathcal{N}(\mu_{\boldsymbol{\eta}_k}, \sigma_{\boldsymbol{\eta}_k})$	External force excreted onto the
$\boldsymbol{p}_i = [p_{2,i}\ p_{1,i}\ p_{0,i}]^T \in \mathbb{R}^3$.	Coefficients of Q's axial restriction
$\boldsymbol{a_n} \in \mathbb{R}$	Set of vectors in direction of n$^{\text{th}}$ axis
$\hat{a}_n \in \mathbb{R}$	Estimate in direction of the n$^{\text{th}}$ axis
$\hat{\boldsymbol{a}}_{\boldsymbol{n}} = \sum_{i=1}^{n} \hat{a}_n \boldsymbol{e_i}$	Estimate over first n axes
$\hat{\boldsymbol{a}}$	Q's maximum estimate with a policy
$Q^{(\mathbf{0})}_{\mathbf{s},n}(a) = Q(\mathbf{s}, \boldsymbol{p} + u\boldsymbol{e_n})$	Univariate function in the direction of axis $\boldsymbol{e_n}$, passing through point $\boldsymbol{p}$

A that maximizes *expected* state-value function V,

$$V(\mathbf{s}) = E(R(\mathbf{s}) + \gamma \max_{\mathbf{a} \in A} V(\mathbf{D}_s(\mathbf{s}, \mathbf{a}))).$$

We use linear map approximation and represent V with Equation (4.2) with the feature vector given in Equation (5.2). We find the approximation to Equation (4.3).

6.1.2 Least Squares Axial Policy Approximation (LSAPA)

The *Least squares axial policy approximation* (LSAPA) policy extends Axial Sum Policy to handle larger disturbances. Axial Sum Policy is applicable to near zero mean disturbances due to the use of Lagrangian interpolation to find an approximation to the maximal Q value. The Lagrangian interpolation uses only three points to interpolate the underlying quadratic function and this compounds the error from the disturbances. In contrast, our new method, LSAPA, uses least squares regression with many sample points to compensate for the induced error.

Consider a fixed arbitrary state $\mathbf{s}$, in a control-affine system, Equation (2.8), with state-value approximation, Equation (4.2), action-value function, Q, is a

quadratic function of the action **a** Axial Sum Policy approximation finds an approximation for the Q function maximum locally for a fixed state **s**. It works in two steps, finding maximums on each axis independently, and then combining them together. To find a maximum on an axis, the method uses Lagrangian interpolation to find the coefficients of the quadratic polynomial representing the Q function Then, an action that maximizes the Q function on each axis is found by zeroing the derivative. The final policy is a piecewise maximum of a Convex and Non-Convex Sums of the action maximums found on the axes. The method is computationally-efficient, scaling linearly with the action space dimensionality $O(d_a)$. It is also consistent, as the maximum selections do not depend on the selected samples.

Because deterministic axial policies are sample independent, they do not adapt to the changing conditions and influence of the external force. We extend the deterministic axial policies for the presence of disturbances. The LSAPA uses least squares regression, rather than Lagrangian interpolation, to select the maximum on a single axis. This change, allows the LSAPA method to compensate for the error induced by non-zero mean disturbances. We now present finding the maximum on ith axis using the least squares linear regression with polynomial features. Q-axial restriction, $Q^{(\mathbf{0})}_{\mathbf{s},i}(a) = Q(\mathbf{s}, a\boldsymbol{e_i})$ on i^{th} axis is a quadratic function,

$$Q^{(\mathbf{0})}_{\mathbf{s},i}(a) = \boldsymbol{p}_i^T [a^2 \ a \ 1]^T,$$

for some vector $\boldsymbol{p}_i = [p_{2,i} \ p_{1,i} \ p_{0,i}]^T \in \mathbb{R}^3$ based on Proposition 5.1.1. Our goal is to find $\boldsymbol{p}_i$ by sampling the action space A at fixed state.

Suppose, we collect d_n action samples in the i^{th} axis,

$$A_i = [a_{1,i} \ ... \ a_{d_n,i}]^T.$$

The simulator returns state outcomes when the action samples are applied to the

fixed state $\mathbf{s}$,

$$S_i = [\mathbf{s}'_{1,i} \ldots \mathbf{s}'_{d_n,i}]^T,$$

where

$$\mathbf{s}'_{j,i} \leftarrow \mathbf{D}(\mathbf{s}, a_{j,i}), \quad j = 1, ..., d_n.$$

Next, Q-estimates are calculated with Equation (2.6),

$$\boldsymbol{Q_i} = [Q_{\mathbf{s},i}(a_{ii}) \ldots Q_{\mathbf{s},i}(a_{ii})]^T,$$

where $Q_{\mathbf{s},j}(a_{ij}) = \boldsymbol{\theta}^T \boldsymbol{F}(\mathbf{s}'_{j,i}), \quad j = 1, ..., d_n$. Using the supervised learning terminology the Q estimates, $\boldsymbol{Q_i}$, are the labels that match the training samples A_i. Matrix,

$$G_i = \begin{bmatrix} (a_{1,i})^2 & a_{1,i} & 1 \\ (a_{2,i})^2 & a_{2,i} & 1 \\ & \ldots & \\ (a_{d_n,i})^2 & a_{d_n,i} & 1 \end{bmatrix},$$

contains the training data projected onto the quadratic polynomial space. The solution to the supervised machine learning problem,

$$G_i \boldsymbol{p}_i = \boldsymbol{Q_i} \tag{6.3}$$

fits $\boldsymbol{p}_i$ into the training data G_i and labels $\boldsymbol{Q_i}$. The solution to Equation (6.3),

$$\hat{\boldsymbol{p}}_i = \underset{\boldsymbol{p}_i}{\operatorname{argmin}} \sum_{j=1}^{d_n} (G_{j,i} \boldsymbol{p}_i - Q_{\mathbf{s},j}(a_{ij}))^2 \tag{6.4}$$

are the coefficient estimates of the Q-axial restriction. Because Q is quadratic, we obtain its critical point by zeroing the first derivative,

$$\hat{a}_i^* = -\frac{p_{1,i}}{2p_{2,i}}.$$

Lastly, we ensure that the selection action falls within the allowed action limits,

$$\hat{a}_i = \min(\max(\hat{a}_i^*, a_i^l), a_i^u). \tag{6.5}$$

Repeating the process of estimating the maximums on all axes and obtaining $\hat{\boldsymbol{a}}_{\boldsymbol{i}} = [\hat{a}_1, ..., \hat{a}_{d_u}]$, we calculate the final policy with

$$\hat{\boldsymbol{\pi}}(\mathbf{s}) = \begin{cases} \boldsymbol{\pi}_{\boldsymbol{c}}^Q(\mathbf{s}), & Q(\mathbf{s}, \boldsymbol{\pi}_{\boldsymbol{c}}^Q(\mathbf{s})) \geq Q(\mathbf{s}, \boldsymbol{\pi}_{\boldsymbol{n}}^Q(\mathbf{s})) \\ \boldsymbol{\pi}_{\boldsymbol{n}}^Q(\mathbf{s}), & \text{otherwise} \end{cases} \tag{6.6}$$

where

$$\boldsymbol{\pi}_{\boldsymbol{n}}^Q(\mathbf{s}) = \sum_{i=1}^{d_u} \hat{a}_i \boldsymbol{e_i}, \qquad \text{(Stohastic Non-Convex Policy)}$$

$$\boldsymbol{\pi}_{\boldsymbol{c}}^Q(\mathbf{s}) = d_u^{-1} \boldsymbol{\pi}_{\boldsymbol{n}}^Q, (\mathbf{s}) \qquad \text{(Stohastic Convex Policy)}$$

The policy approximation, Equation (6.6), combines the simple vector sum of the Stochastic Non-Convex Policy with Stochastic Convex Sum Policy using Equation (6.5). The Convex Sum Policy guarantees the system's monotonic progression towards the goal, but the Non-Convex Sum does not. If, however, the Non-Convex Sum performs better than the Convex Sum policy, then Equation (6.6) allows us to use the better result.

Algorithm 6.10 summarizes the process of calculating the policy approximation. With no external disturbances Algorithm 6.10 results in the Axial Sum Policy, Equation (5.11). On the other hand, in the presence of the disturbances, the regression fits Q into the observed data, and the algorithm adapts to the changes in the disturbance.

Algorithm 6.10 Least squares axial policy approximation (LSAPA)

Input: current state $\mathbf{s}$, parametrization estimate $\boldsymbol{\theta}$

Input: basis function vector $\boldsymbol{F}$, simulator $\mathbf{D}_s$

Output: $\hat{\boldsymbol{a}}$

1: **for** $i = 1, .., d_u$ **do**
2: sample action $A_i = [a_{1,i} \; ... \; a_{d_n,i}]^T$
3: **for** $j = 1, ..., d_n$ **do**
4: $\mathbf{s}'_{j,i} \leftarrow \mathbf{D}(\mathbf{s}, a_{j,i})$
5: $Q_{\mathbf{s},j}(a_{ij}) \leftarrow \boldsymbol{\theta}^T \boldsymbol{F}(\mathbf{s}'_{j,i})$
6: **end for**
7: $\hat{\boldsymbol{p}}_i \leftarrow \text{argmin}_{\boldsymbol{p}_i} \sum_{j=1}^{d_n} (G_{j,i}\boldsymbol{p}_i - Q_{\mathbf{s},j}(a_{ij}))^2$
8: $\hat{a}_i^* \leftarrow -\frac{p_{1,i}}{2p_{2,i}}$
9: $\hat{a}_i = \min(\max(\hat{a}_i^*, a_i^l), a_i^u)$
10: **end for**
11: $\hat{\boldsymbol{a}} \leftarrow$ calculated with Equation (6.6)
12: return $\hat{\boldsymbol{a}}$

6.1.3 Stochastic Continuous Action Fitted Value Iteration

Algorithm 6.11 is applied during the learning phase. It leverages Algorithm 6.10 to derive a policy and iteratively updates the expected state-value function. To adapt the CAFVI to systems with disturbances, we use LSAPA rather than a deterministic admissible policy. The algorithm learns state-value function, V, globally, like the standard approximate value iteration [36]. It uniformly randomly samples the state space, and using the provided feature vectors and its current weight $\boldsymbol{\theta}$, calculates the new estimate of the state-values. Instead of using the greedy policy for estimating state-values, it uses policy approximation in Line 6 to locally learn action-value function Q, and find near-optimal action. After all state samples are processed, the algorithm updates the feature weights $\boldsymbol{\theta}$ with least squares linear

regression. Policy approximation, LSAPA in Line 6 uses least squares minimization, and thus requires the stochastic disturbance data points to be independent and identically distributed (i.i.d.). However, the same distribution does not need to hold between different runs of the LSAPA, thus both learning and planning independently adapt to the changes in the disturbance.

Algorithm 6.11 Stochastic continuous action fitted value iteration (SCAFVI)

Input: S, A, discount factor γ

Input: basis function vector $\boldsymbol{F}$, simulator D

Input: $max\_iterations$, ϵ

Output: θ

1: $\theta_0 \leftarrow$ zero vector
2: $k \leftarrow 0$
3: **while** ($k \leq max\_iterations$) or $\|\theta_{k-1} - \theta_k\| \geq \epsilon$ **do**
4: **for** $k_s = 1, .., n_s$ **do**
5: select a random state and observe its reward $(\mathbf{s}_{k_s}, R_{k_s})$
6: $\hat{\boldsymbol{a}}_{\boldsymbol{k_s}} \leftarrow \text{LSAPA}(\mathbf{s}_{k_s}, \boldsymbol{\theta_{k+1}}, \boldsymbol{F}, \boldsymbol{D})$
7: obtain $\{\mathbf{s}_{k_s}, \hat{\boldsymbol{a}}_{\boldsymbol{k_s}}, \mathbf{s}'_{k_s}\}$
8: $v_{k_s} = R_{k_s} + \gamma \boldsymbol{\theta}_k^T \boldsymbol{F}(\mathbf{s}'_{k_s})$ {estimate state-value}
9: **end for**
10: $\boldsymbol{\theta_{k+1}} \leftarrow \text{argmin}_{\boldsymbol{\theta}} \sum_{l_s=1}^{n_s} (v_{k_s} - \boldsymbol{\theta}^T \boldsymbol{F}(\mathbf{s}_{k_s}))^2$ {least-squares regression}
11: $k \leftarrow k + 1$
12: **end while**
13: return $\boldsymbol{\theta_k}$

We showed in Theorem 5.1.4, that when the deterministic CAFVI returns negative weights, the task goal is asymptotically stable and the agent progresses to the goal. In the stochastic case, we do not have this guarantee. Instead, the probability of reaching the goal depends on the combination of the maximum action

magnitude and probability distribution of the disturbances. The convergence of general AVI depends on the nature of the feature vectors and the underlying system dynamics [27, 12]. Also, because of the sampling nature of the algorithm, it is susceptible to a possibility of divergence. For that reason, we learn in Monte Carlo fashion, repeating the learning for several trials and testing fitness of resulting policies.

6.2 Results

To evaluate SCAFVI (Section 6.1.3) and motion planning using LSAPA (Section 6.1.2), we use three tasks: Minimal Residual Oscillations Task, Rendezvous Task, and Flying Inverted Pendulum. For all tasks we show their definition and the feature derivation from the definition. Learning evaluation is done on the Minimal Residual Oscillations Task in Section 6.2.2, the planning performance for varying initial conditions is evaluated on the Rendezvous Task in Section 6.2.3. Flying Inverted Pendulum in Section 6.2.4 explores the number of needed samples.

6.2.1 Setup

To evaluate the stochastic continuous action selection described in Algorithm 5.7, we compare it to the baseline deterministic Axial Sum Policy. All learning and planning occurs at 50 Hz. All trajectory planning was performed on a single core of Intel Core i7 system with 8GB of RAM, running the Linux operating system using Matlab 2011.

6.2.2 Minimal Residual Oscillations Task

We first consider a Minimal Residual Oscillations Task. We use the same MDP setup as in Section 5.2.2 with the only difference that here we rely on a stochastic simulator $\mathbf{D}_s$.

We ran Algorithm 6.11 for 100 trials with 300 iterations with varying disturbance noise parameters. During the learning phase, we generated a 15 s trajectory starting at (-2 , 2, 1)m from the origin. Table 6.2 shows the trajectory characteristic ot the learning trials. Due to the constant presence of the disturbance, we consider the average position of the quadrotor and the load over the last second, rather than simply expecting to reach the goal region. The results in Table 6.2 show that both learning and planning using stochastic policy are slower than using the deterministic policy. This is expected because the deterministic policy uses 3 samples per action dimension, while the stochastic policy in this case uses 300 samples. Nevertheless, the planning time is still an order of magnitude smaller than the 15 s trajectory duration, allowing ample time to plan the trajectory in a real-time closed feed-back loop. Next in Table 6.2, we see that the stochastic policy produces consistent residual distance to the goal. The larger the variance of the disturbance, the larger the error. When the mean is $2\,\mathrm{m\,s^{-2}}$ and the standard deviation is 1, the stochastic policy results start degrading. This is because the upper limit on the action space is $3\,\mathrm{m\,s^{-2}}$, and the noisy conditions start overwhelming the system. The deterministic policy, learning and acting on the same data, fails to bring the system near the goal. The two policies show similar behavior only for the zero-mean noise with small (0.5) variance.

For LSAPA and deterministic Axial Sum Policy, Figure 6.1a shows the learning rate in logarithmic scale. Both policies converge to their peak performance around 200 iterations, but the LSAPA policy collects more reward. Figure 6.1 shows the

trajectories planned with LSAPA and a deterministic axial sum in an environment with $\mathcal{N}(2, 0.5)$ noise. Although both quadrotor's and the load's speeds are noisy (Figures 6.1b and 6.1c), the position moves smoothly and arrives near the goal position where it remains. This is in contrast to trajectories planned with a deterministic axial sum that never reaches the origin.

6.2.3 Rendezvous Task

The Rendezvous Task involves two heterogeneous robots, an aerial vehicle with the suspended load, and a ground robot. The two robots work together for the cargo to be delivered on top of the ground robot (Figure 5.7), thus the load must have minimum oscillations when they meet. We use the same setup as in Section 5.2.3. Supplying this information to the Algorithm 6.11, we get the parametrization for the state-value approximation.

To evaluate how the resulting policy behaves for different initial conditions starting from 5 m apart, we run it for 100 different initial positions of the aerial and ground robots for a disturbance with a $\mathcal{N}(1, 1)$ distribution. Table 6.3 presents the results. Rendezvous Task policy reaches a rendezvous less that 1 cm apart on average, while deterministic Axial Sum Policy diverges from the goal. LSAPA spends 4.36 s on average to plan the 15 s trajectory.

Figure 6.2 shows a trajectory of the Rendezvous Task learned with Algorithm 6.11 and created with the LSAPA (Algorithm 6.10). The two robots rendezvoused in 4 s, after the initial failed coordinated slow down at 1 s. Note, that the targeted position for the quadrotor is 0.6 s in order for the load to be positioned above the ground robot.

6.2.4 Flying Inverted Pendulum

The last task we consider is Flying Inverted Pendulum. With the exception of the maximum acceleration, which is $5\,\mathrm{m\,s^{-2}}$ here, the we use the same setup as in Section 5.2.4. To access the adaptive planning, we learn the policy with the deterministic CAFVI. In the planning phase, we use a disturbance probability density function $\mathcal{N}(1,1)$ and a pole initial displacement of 23°. While the deterministic sum solves this problem and balances the inverted pendulum in the absence of the disturbances and small zero-mean disturbances ($\mathcal{N}(0,0.5)$ in Table 5.6), it fails to balance the inverted pendulum for non-zero mean noise. In contrast, LSAPA policy solves the task (Figure 6.3). Figure 6.3a shows the quadrotor's trajectory and Figure 6.3b displays pendulum position in Cartesian coordinates relative to the target position above the quadrotor. During the first 5 s, the first subtask brings the pole upright, followed by the second task that slows down the quadrotor. The pole is slightly disturbed during the initial slowdown but returns to upright shortly.

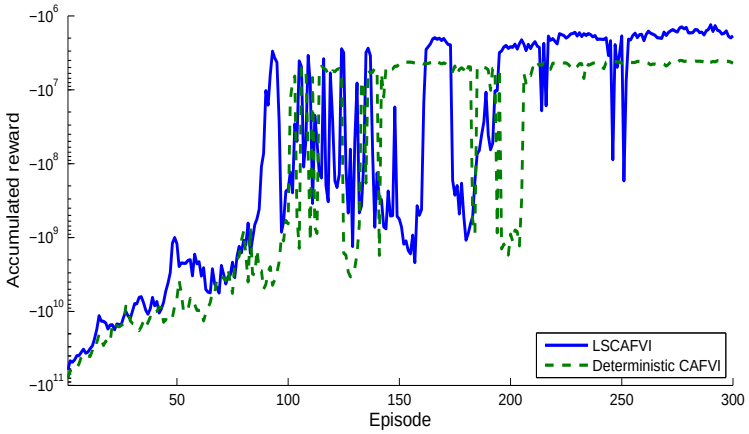
Figure 6.4 depicts the results of the trajectory characteristics as the number of samples d_n in Algorithm 6.10 increases. The smallest number of samples is three. The accumulated reward (Figure 6.4a) increases exponentially below 10 samples. Between 10 and 20 samples the gain decreases, and the peak performance is reached after 20 samples. Increased sampling beyond that point brings no gain. We see the same trend with the pole displacement (Figure 6.4b) and speed magnitude (Figure 6.4c).

6.3 Conclusion

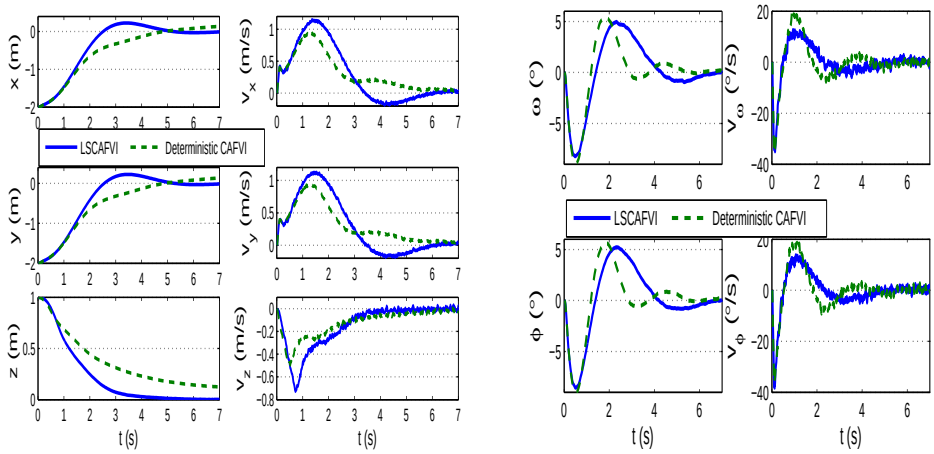
This chapter presented a method for policy approximation and RL for robots learning PBTs in environments with external stochastic disturbances. The method uses least squares linear regression to find an optimal action on each axis. The resulting action is a combination of the axial maximums. Placed in the AVI setting, we presented a learning algorithm that learns to perform the task. The work is a stochastic extension of methods in Chapter 5, where we showed that in deterministic kinematic systems with a bounded drift learning a PBTs with quadratic features with negative weights, the goal state becomes asymptotically stable and the policy is safe for the physical system. This paper takes an empirical approach to access the safety of the policy. We showed that the method is applicable for a range of practical problems, and offered a discussion on feature vector selection from a task description.

Table 6.2: Summary of learning results for Minimal Residual Oscillations Task task averaged over 100 trials. Policy, injected noise distribution, time needed to learn the policy, time to plan a trajectory, average distance from the goal and load displacement during the last 1 s of the flight, and maximum load displacement.

Policy	Noise		Computational Time (s)		Distance (cm)		Swing (°)		Max. Swing (°)	
	μ	σ^2	Learning	Planning	μ	σ	μ	σ	μ	σ
	0.00	0.50	56.13	0.87	2.20	10.72	0.21	0.51	12.47	2.09
	1.00	0.50	55.59	0.86	1.58	3.94	0.18	0.29	12.16	1.84
LSCAVFI	2.00	0.50	56.02	0.87	1.68	8.68	0.18	0.50	12.60	2.02
(Stochastic)	0.00	1.00	55.74	0.87	1.42	1.43	0.30	0.29	12.51	2.30
	1.00	1.00	55.71	0.86	3.84	2.46	0.31	0.19	11.85	1.91
	2.00	1.00	55.61	0.86	15.06	25.08	0.56	0.52	12.56	2.06
	0.00	0.50	21.41	0.30	0.69	1.06	0.13	0.15	13.03	1.94
	1.00	0.50	18.47	0.30	15.59	2.57	0.12	0.08	12.87	2.00
CAFVI	2.00	0.50	19.20	0.30	30.95	6.83	0.19	0.58	13.65	1.91
(Deterministic)	0.00	1.00	18.92	0.30	2.28	7.22	0.30	0.48	12.49	2.05
	1.00	1.00	18.76	0.30	15.91	3.45	0.27	0.29	12.89	2.11
	2.00	1.00	18.70	0.30	32.95	6.03	0.35	0.30	13.60	2.10



(a) Learning rate



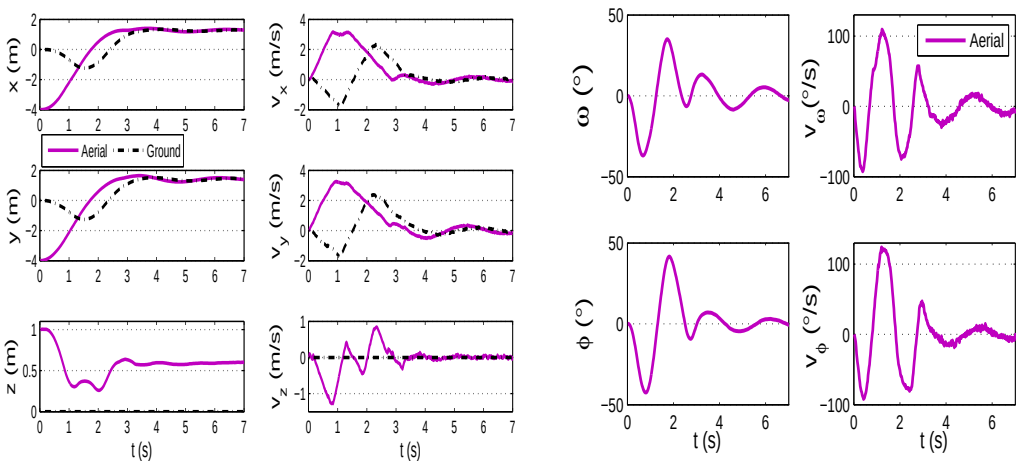
(b) Quadrotor trajectory

(c) Load trajectory

Figure 6.1: Stochastic Minimal Residual Oscillations Task - learning curve and trajectory created with SCAFVI compared to a trajectory created with deterministic Axial Sum Policy with disturbance of $\mathcal{N}(2, 0.5)$

Table 6.3: Summary of planning results created with LSAPA (Algorithm 6.10) and deterministic Axial Sum Policy for Rendezvous Task averaged over 100 different inital conditions. Disturbance: $\mathcal{N}(1,1)$ Policy, injected noise distribution, time needed to learn the policy, time to plan a trajectory, average distance from the goal and load displacement during the last 1 second of the flight, and maximum load displacement.

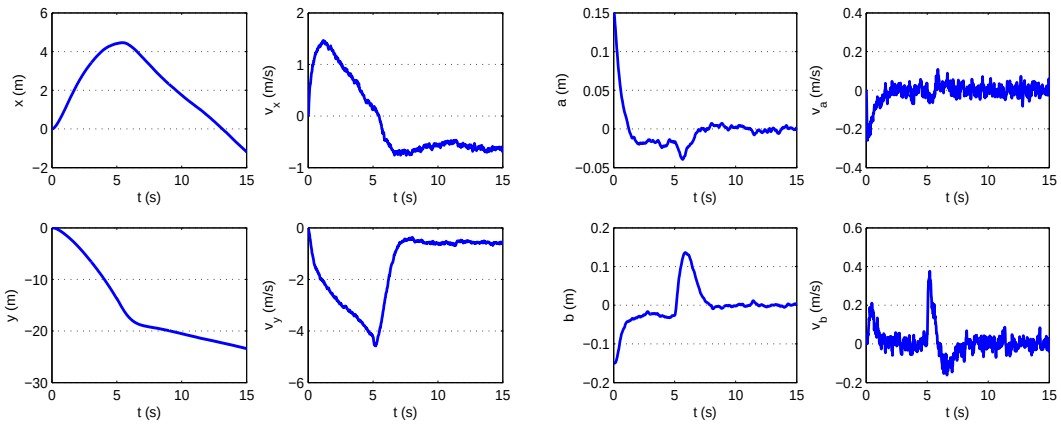
		Policy	
		LSAPA	Deterministic Axial Sum
Computational Time (s)	Learning	91.99	23.37
	Planning	4.36	1.21
Distance (cm)	μ	0.75	4459.46
	σ^2	0.29	317.69
Swing (°)	μ	0.34	17.21
	σ^2	0.14	2.22
Max. Swing (°)	μ	33.14	22.46
	σ^2	11.94	3.19



(a) Quadrotor trajectory

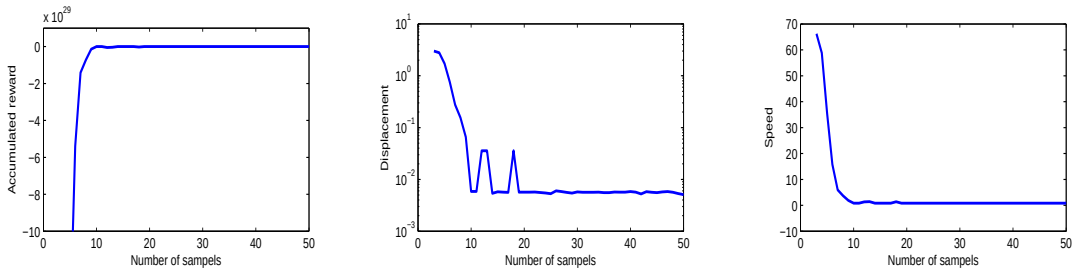
(b) Load trajectory

Figure 6.2: Rendezvous Task trajectories with $\mathcal{N}(1,1)$.



(a) Quadrotor trajectory (b) Inverted trajectory

Figure 6.3: Flying inverted pendulum trajectory created with stochastic Axial Sum Policy with disturbance of $\mathcal{N}(1,1)$.



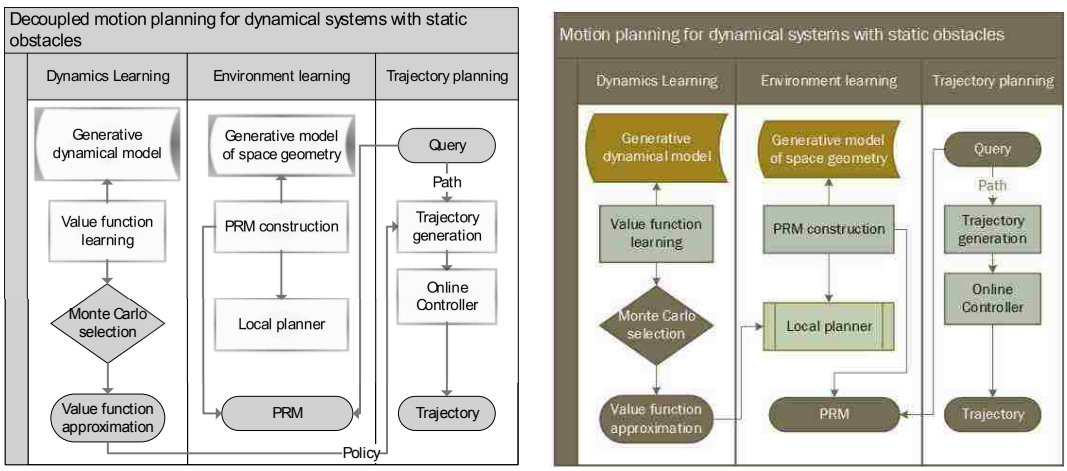
(a) Accumulated reward (b) Displacement (logarithmic) (c) Speed

Figure 6.4: Trajectory characteristics per number of samples in stochastic flying pendulum with disturbance $\mathcal{N}(1,1)$, calculated stochastic sum and averaged over 100 trials.

Chapter 7

PEARL as a Local Planner

This chapter places CAFVI in the context of a local planner to offer another method for solving Cargo Delivery Task (Definition 4.1.5). This solution integrates path planning with the control of the system.



(a) Decoupled (b) Integrated

Figure 7.1: Decoupled (a) and integrated (b) architecture frameworks.

7.1 Integrated Path and Trajectory Planning

In Chapter 4, we developed a motion planning method for creating trajectories for holonomic UAVs that constrain the load displacement while avoiding obstacles in a static environment. At the heart of the method was decoupling of learning the physical environment and learning the system dynamics constraints. This method combined the advantages of being highly concurrent and flexible with the ability to change among a wide-variety of algorithms. Although this method solves Minimal Residual Oscillations Task, it required path bisecting to solve the Cargo Delivery Task, which slows down the task (Chapter 4). Here we integrate the trajectory generation with path planning. We propose an integrated framework (see Figure 7.1b), where an efficient motion planner is used as a local planner to reject paths that violate task-constraints, i.e., predicted load displacement.

The motion planner is an agent that executes Axial Sum Policy. We assume start and goal states in *C-space*. Without loss of generality, we can also assume that they are valid (*C-free*) states: free of collision, and with both task and dynamics constraints satisfied at zero velocities. Otherwise, we can make a quick check and return without further processing. The algorithm takes a swing-constant ϵ (task constant), task preferences, and dynamical constraints as inputs. The dynamical constraints put limits on the actuators, i.e., acceleration. When either of the limits is violated the system's safety cannot be guaranteed, and the task cannot be performed.

Algorithm 7.12 describes the motion planner. It works simultaneously in both the *C-space* and MDP space. It starts with the initial state and repeatedly finds the best action to perform, validates that the action and resulting state satisfy both dynamical and task-mandated constraints, and that it is free of collision. If all the conditions are met, the algorithm continues on to the next state. It stops when the

resulting state is the goal state, when a maximum number of steps is reached, or if any of the constraints are violated.

Algorithm 7.12 Task-based motion planner.

Input: $\mathbf{s}_s$, $\mathbf{s}_g$: start and goal
Input: V: approximation of MDP value function
Input: W: workspace generative model
Input: Vol robot's geometric bounding volume,
Input: $\mathbf{D}$ generative model of system dynamics
Input: $dynLimits$, $taskLimits$, $maxSteps$
Output: $success$

$\mathbf{s}' \leftarrow toMDPSpace(\mathbf{s}_s)$
$success \leftarrow true, steps \leftarrow 0$
while $success \wedge steps < maxSteps \wedge \|\mathbf{s}' - \mathbf{s}_g\| > \epsilon$ **do**
  $\mathbf{s} \leftarrow \mathbf{s}'$
  $\mathbf{a} \leftarrow selectAction(\mathbf{s}, dynLimits, \mathbf{D}, V)$ per Equation (5.11)
  $\mathbf{s}' \leftarrow \mathbf{D}.predictState(\mathbf{s}, \mathbf{a})$
  $taskOK \leftarrow checkTaskConstraints(\mathbf{s}', taskLimits)$
  $cs' \leftarrow toCSpace(\mathbf{s}')$
  $configOK \leftarrow checkConfig(cs', W, Vol)$
  $success \leftarrow taskOK \wedge configOK, step \leftarrow step + 1$
end while
$success \leftarrow success \wedge \|\mathbf{s}' - \mathbf{s}_g\| < \epsilon$
return $success$

To allow the agent to navigate in environments with static obstacles, we integrate the motion planner described in Algorithm 7.12 with PRMs. Under this setup the motion planner acts as a local planner, and only edges that conform to task and dynamics limits are added to the roadmap. Since the planner ensures that the constraints are met, we can set the weights on the edges to a metric related to a

quality of a trajectory. For example, instead of using the shortest distance metric, we might be looking for the fastest trajectories, and setting the edge weight to the trajectory duration. PRMs then find a path through the roadmap that minimizes the edge weight.

Figure 7.1b shows the overview of the integration of the RL-based planner with PRMs. The key to this framework is that we learn the system dynamics and environment separately. First, we use RL to learn the dynamics of the system. We learn using a Monte Carlo selection: multiple policies are learned simultaneously and the fittest one is selected. The fitness metric can be the duration of the trajectory. Then the value function is passed to the PRMs, which learn a particular environment using the RL-based motion planner as a local planner. Once the roadmap is constructed it can be used for any number of queries in the environment. The value function is robot and task dependent, and the same function can be used for a multitude of environments, as we will show in the results.

The computational efficiency of the motion planner allows it to be used in this manner. A roadmap with n nodes requires $O(n^2)$ edge calculations, thus the time required to compute the roadmap still scales linearly with with product of state and action space dimensionality. We use the geometric robot model developed in Section 4.2.4 and depicted in Figure 4.1c. The quadrotor with the suspended load is modeled as a combination of two rigid bodies: a cylinder for the quadrotor, and a cone for the suspended load. The cone's aperture corresponds to the maximum allowed swing, and ensures that the load does not come in contact with the environment.

7.2 Results

We evaluate the local planner integrated with PRMs in obstacle-laden environments. A model of a city, our simulation environment, allows us to test the algorithm over longer distances; while two experimental environments allow us to compare to previous methods and to experimentally verify the algorithm on a Hummingbird quadrotor UAV equipped with a suspended load. We use the same set described in Section 4.3.4.

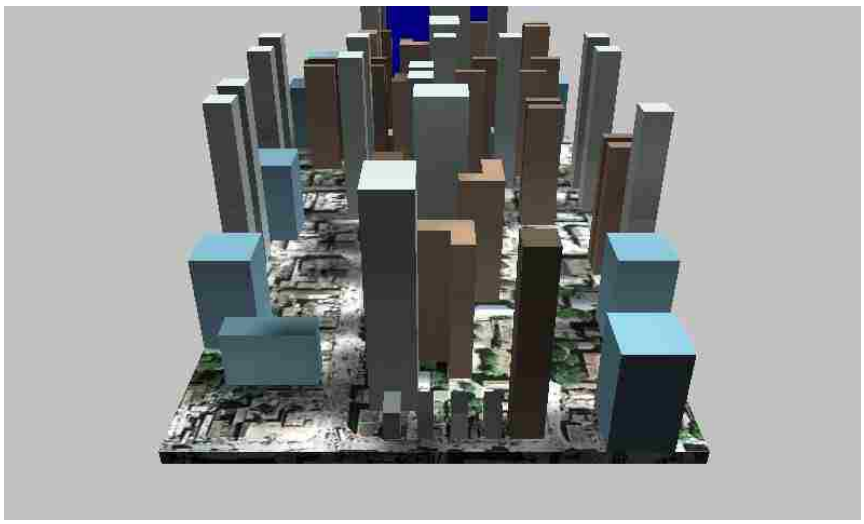


Figure 7.2: City simulation envrionment.

7.2.1 Simulations

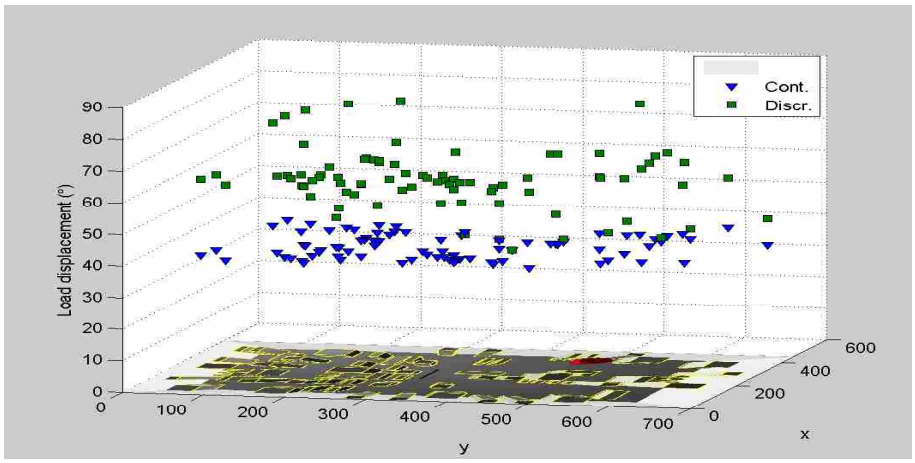
To verify our planner in large spaces we use the city model depicted in Figure 7.2. This virtual benchmark represents an urban environment. In cities, UAVs perform Cargo Delivery Tasks such as carrying injured people to hospitals and delivering packages. Both need to be done in the timely manner, avoiding obstacles, and controlling the load displacement. There are numerous obstacles in this environ-

ment, and this benchmark tests the methodology in a cluttered environment. The city block is 450 m, 700 m long, and 200 m tall. Its bounding box is 250 million times larger than the quadotor's model shown in Figure 4.1c.

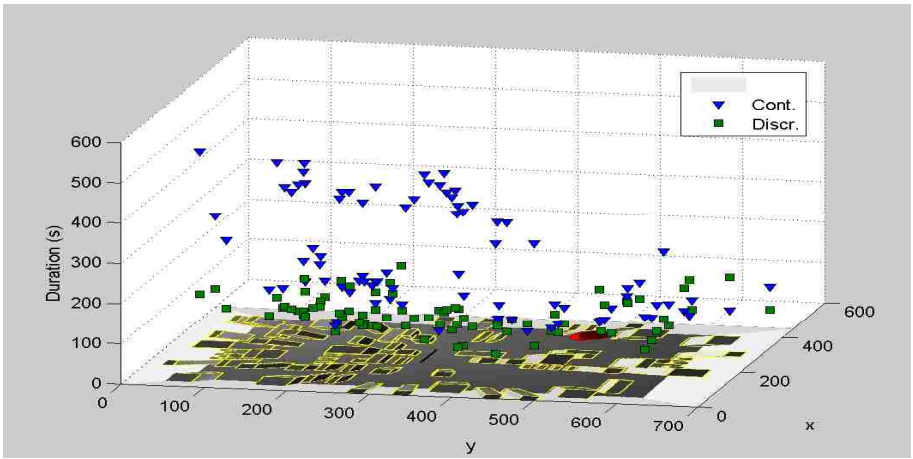
We randomly select 100 locations in *C-free* within the city. Then we generate trajectories from a fixed origin to all 100 destinations. The Axial Sum planner trajectories are compared to the discrete greedy trajectories. To generate the discrete trajectories, we first created a collision-free path using PRMs with a straight line planner and the simplified robot model from Figure 4.1c. Recall that the geometry of the model ensured that if the load displacement during the trajectory was below 45°, the trajectory will be free of collision. Then, in the second step, we used the nodes from the path and generated the Minimal Residual Oscillations Trajectories with discrete actions between the nodes in the path. To generate trajectories in the continuous case, the planner was used as described in Section 7.1, requiring the load displacement to be under 45°. We measure the load displacement, trajectory duration, and number of waypoints.

Figure 7.3 shows the load displacement and trajectory duration results. The xy-plane contains the city projection. The red area marks the origin location. The data points' x and y coordinates are the queries' goal location in the city projected onto the xy-plane. The data points' z-coordinates in Figure 7.3a are the maximum load displacement throughout the trajectory, and in Figure 7.3b the time it takes to complete the trajectory. Points represented as squares (green) are the result of the discrete planner, and the triangle points (blue) are generated with the continuous action planner.

Figure 7.3a shows that the maximum load displacement in the continuous case is consistently below the load displacement exhibited with the discrete planner. Table 7.1 shows that the load displacement in the continuous case stays under the required 45°, while in the discrete case the load's suspension cable almost gets



(a) Load displacement



(b) Duration

Figure 7.3: Trajectory statistics in the City Environment. Load displacement (a) and trajectory duration (b) for continuous action planner (triangle), compared to discrete planner (square) in the city environment. City image is projected on the xy-plane.

parallel to the UAV's body for some trajectories.

To offset the load displacement control, the trajectories with the continuous action planner take a longer time to complete the task (see Figure 7.3b and Table 7.1). They generally take twice as long to complete and contain on average 5

Table 7.1: Summary of trajectory characteristics for continuous and discrete action planners over 100 queries in the city environment. Mean (μ), standard deviation (σ), minimum and maximum are shown. Best results are highlighted.

Method		Continuous	Discrete
Load displacement(°)	μ	**41.44**	62.58
	σ	**1.77**	10.36
	min	**36.22**	36.31
	max	**44.76**	89.86
Duration (s)	μ	252.76	**116.35**
	σ	171.34	**56.72**
	min	**33.36**	48.16
	max	565.84	**262.84**
Waypoints	μ	12.88	1.74
	σ	9.74	1.00
	min	1.00	1.00
	max	31.00	6.00

times more waypoints. Figure 7.3b shows that in the vicinity of the origin, both continuous and discrete trajectories have similar durations. As the goal's distance from the origin increases, the discrepancy becomes more significant. And even when the trajectories are over 1 km in length, the flights complete within 10 min.

7.2.2 Experiments

The goal of the experiments is to evaluate the motion planner and its integration with PRMs for a physical robot and validate the simulation results. Using the setup described in Section 4.3.4, we compare simulation and experimental trajectories and look for discrepancies in the length, duration, and load displacement over the entire trajectory.

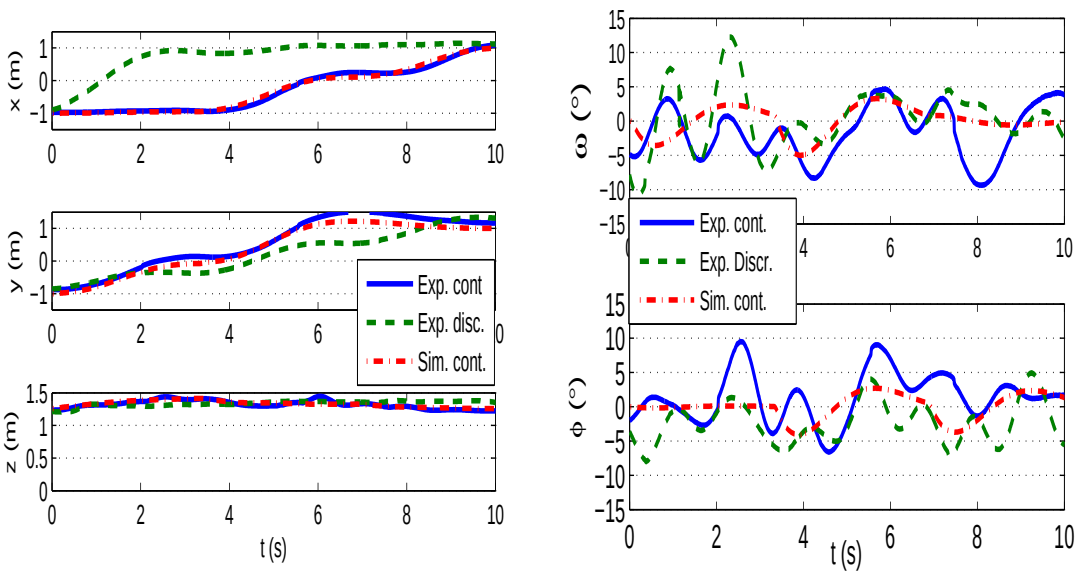
In Environment 1 from Figure 4.10b, we created a roadmap using the continuous action local planner. We require that load displacement not exceed 10° with respect to L_∞ norm. The resulting planned trajectory is flown in the testbed. Figure 7.4 depicts the resulting vehicle and load trajectories. We can see that the vehi-

cle trajectories match closely between simulation and experiment. The results also suggest that the load displacement stays under 10°, even in experiments. Further, the discrepancy between simulation and experiments of 10° remains consistent with our previous results from Section 4.3.2. This is due to the unmodeled load turbulence caused by the propellers' wind.

In the next set of experiments, we create a trajectory in the same environment using the discrete action trajectory generation with PRMs method described in 7.2.1. The resulting trajectory was flown in the testbed and its results are in Figure 7.4. Note that the vehicle trajectory differs in this case. This is because the roadmaps in the two cases differ. The roadmap with straight line planner does not directly control the load displacement, while the roadmap created with the continuous action planner rejects edges that exceed the maximum allowed load displacement. The load trajectory (Figure 7.4 (b)) indicates that the load displacement exceeds the 10° limit 2.5 s into the flight.

7.3 Conclusion

This chapter presented another method for applying the policy approximation developed in Chapter 5. The two solutions for Cargo Delivery Task, one presented here and the other one developed in Chapter 4 have different strengths. The integrated path and trajectory planning solution develops faster trajectories, while the decoupled solution (Chapter 4) offers more flexibility in the choice of the local planner and trajectory generation algorithms.



(a) (b)

Figure 7.4: Results of experimental quadrotor (a) and load (b) trajectories planned with continuous actions compared to the experimental trajectory with discrete actions and simulated trajectory with continuous actions.

Chapter 8

PEARL for Non-Motion Tasks

To evaluate PEARL's generality, we apply it to software development. This chapter contributes:

- A controlled dynamical system formulation for algorithmic computing (Section 8.1), and
- A PEARL sorting agent (Section 8.2).

Software development is time consuming and error prone [46]. Often the software is intended to run under unforeseen conditions. Nowadays, software interacts with other software components that may be unreliable due to malware or bugs [3]. Yet, unlike robotics, software engineering handles reliability through development process best practices such as code reviews, pair-programming [46], and design patterns [19]. In high consequence applications where human life is at risk, the correctness of specific routines is proven mathematically, a difficult method requiring highly specialized knowledge [46]. Given the similarities in the operating uncertainties of the software and robots, we apply PEARL to software engineering for developing RL computing agents. This method takes require-

ments as input, and produces an agent that performs the desired computation. We consider the computation to be trajectory planning in the program's variable space [91]. After establishing a dynamical system for computing, a case study applies PEARL to array sorting in the presence of the unreliable comparison components.

8.1 PEARL for Computing Agents

To develop computing agents with PEARL framework, we need to formulate its MDP $\mathcal{M}\ (S, A, \mathbf{D}, R)$, and define task-preference features. The control problem definition, (S,A,$\mathbf{D}$) part of MDP, requires determination of the state space, control input (action space), rules for state transformations, and the time interval at what the changes occur. We consider deterministic programs, where a program's outcome is determined by two factors: initial state of its variables, and the sequence of steps (*algorithm*) that solves the program. We do not consider global variables that can change without program's knowledge.

At run-time the program's in-scope variables and the location of the instruction counter uniquely determine the program's state (*configuration*) [91]. Regardless of how a program got into a certain configuration, the computation unfolds the same from that point. Thus, a program's state space is the space of all variable values. Without loss of generality, we assume all variables to be real numbers. Thus, a state space for a program with d_s variables is $S = \mathbb{R}^{d_s}$, and a state is simply an d_s-dimensional vector. Operations that change the variables are the action space. These are programming constructs such as *assignment*, *arithmetic operations*, and *changing instruction pointer*, etc.The control-flow constructs, if-then-else, and loops, are controller switches. Instructions are performed at discrete time steps per internal clock tick. The state transitions are determined by executing instruc-

tions in the instruction register. A program seen this way is a controlled dynamical system that changes the state vector over time until the computation stops. Under this paradigm, a computation is a trajectory in the variable space starting with an initial state of variables (initial vector) and terminating at the goal state (goal vector). We now show that a program with assignment, addition (subtraction), and swap operations is a nonlinear dynamical system. Because the states are vectors, the operations are vector transformations and can be represented with the transformation matrices. Proposition 8.1.1 formalizes that.

Proposition 8.1.1. *A program P with d_a local variables and assignment, summation, and swap operations is a nonlinear discrete time and input system.*

Proof. The proof is by construction. The variable manipulations are changes in the vector space, transforming one vector to another. Finding a transformation matrix between the vectors proves the proposition.

Let $\mathbf{s} = [x_1, ..., x_m]^T \in \mathbb{R}^{na}$ be a vector representing the current state of variables, and the assignment operation $x_i \leftarrow x_j$ assigns the value of the j^{th} variable to the i^{th} variable. Consider a square d_a-by-d_a matrix $T^a_{i,j}$, where its elements $t_{k,l}, 1 \leq k, l \leq d_a$ are defined as follows:

$$t_{k,l} = \begin{cases} 1 & k = i,\ l = j, \\ 1 & k \neq i,\ k = l\ . \\ 0 & otherwise \end{cases}$$

This matrix differs from the identity matrix only in the i^{th} row, where the i^{th} element is zeroed out, and j^{th} element is set to one. Then, vector $\mathbf{s}' = T^a_{i,j}\mathbf{s}$, is a vector with i^{th} component equal to x_j, and other components unchanged from $\mathbf{s}$. Similarly, matrix T^c_i, where $t_{i,i} = c, t_{k,k} = 1, k \neq i$ and zero otherwise, assigns constant c to the i^{th} variable.

We show construction of the two-variable summation action matrix, although the general case can be shown with induction. Consider action matrix T^a_{i,j_1,j_2} defined with

$$t_{k,l} = \begin{cases} 1 & k = l,\ k \neq i, \\ 1 & k = i,\ l \in \{j_1, j_2\} \\ 0 & otherwise \end{cases}.$$

As previously, this matrix differs from the identity matrix only in the i^{th} row, where only elements j_1 and j_2 are non zero.

Lastly, we construct a transformation matrix $T^s_{i,j}$ that swaps x_i and x_j. Consider

$$t_{k,l} = \begin{cases} 1 & k = i,\ l = j \\ 1 & k = j,\ l = i \\ 1 & k = l,\ k \neq i, j \\ 0 & otherwise \end{cases}.$$

Finally, when the action space is set of transformation matrices,

$$T^a_{i,j}, T^c_i, T^a_{i,j_1,j_2}, T^s_{i,j},\ i, j = 1, \ldots, d_s,$$

the variable space manipulation with an action T is a nonlinaer dynamical system $\boldsymbol{f}(\mathbf{s}) = T\mathbf{s}$. □

All programs are nonlinear control systems per Proposition 8.1.1, thus the control theory tools can be applied for program analysis.

Having formulated the control problem $(S, A, \boldsymbol{D})$, we need the reward for the MDP formulation. The reward is a scalar feedback on state quality. Typically, the reward is given only when the goal is reached [120], or using a simple metric of a state when available. Lastly, preference-features are the often opposing algorithm's properties that the software architect requires of the system. For example,

we desire both a fast algorithm and high-precision computation. We are looking for the right combination of the trade-offs that express themselves as some combination of features. Once the learning is set up, applying PEARL to the problem results in a computing agent. During planning, given an initial state, the agent transitions to the reachable state with the highest value, thus progressing the system to the solution.

8.2 Case Study in Software: PEARL Sorting

After demonstrating the PEARL's use on a robotic tasks, we use the same toolset to solve array sorting, a primitive computer science problem. Specifically, we develop a sorting agent that uniformly progresses to produce a sorted array.

The array sorting state space is the d_s-element array itself; $S = \mathbb{R}^{d_s}$. The action space is the discrete set of possible element repositions, $A = \{(i,j)|i,j = 1..d_s\}$. Action (i,j), acts as a list insert, it removes array's i^{th} element and inserts it into the j^{th} position. Considering the arrays as vectors, the actions are permutations of the vector's coordinates, and are reversible. Indeed, transformation matrix $T_{i,j}$, repositions i^{th} element to the j^{th} position when $i < j$, when its elements are defined as

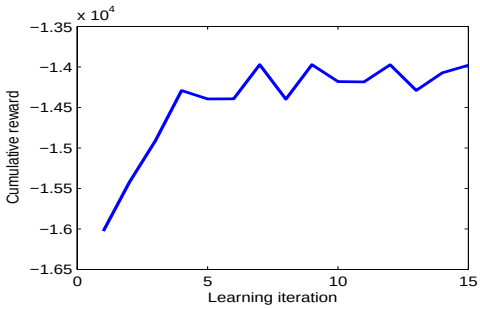
$$a_{k,l} = \begin{cases} 1 & k = l, k < i \\ 1 & k = l, k > j \\ 1 & k = j, l = i \\ 1 & i \leq k < j, l = k - 1 \\ 0 & otherwise \end{cases}, \; i \leq j \text{ and}$$

$$a_{k,l} = \begin{cases} 1 & k = l, k < j \\ 1 & k = l, k > i \\ 1 & k = j, l = i \qquad , \; i \geq j. \\ 1 & j < k \leq i, l = k + 1 \\ 0 & otherwise \end{cases}$$

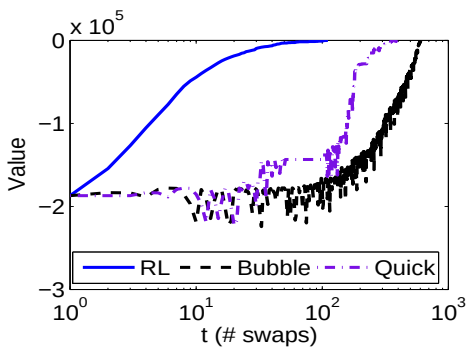
The reward is binary: positive when the array is sorted, and zero otherwise.

The PEARL sort uses the number and the sum of adjacent out-of-order elements as distance-preferences. The second preference ranks arrays with similar elements close together as more sorted than arrays with large variations between adjacent elements. The first feature is defined as $\boldsymbol{F}_1(\mathbf{s}, d_s) = \sum_{i=2}^{d_s}(\text{id}(\mathbf{s}(i) - \mathbf{s}(i-1) < 0))$, and the second is $\boldsymbol{F}_2(\mathbf{s}, d_s) = \sum_{i=2}^{d_s}((\mathbf{s}(i) - \mathbf{s}(i-1))^2 \text{id}(\mathbf{s}(i) - \mathbf{s}(i-1) < 0))$, where $\text{id}(cond)$ is an identity function, equal to one when the condition is true, and zero otherwise. Value function approximation, $V(\mathbf{s}) = \boldsymbol{\theta}^T \boldsymbol{F}(\mathbf{s}, d_s)$, for $\boldsymbol{F}(\mathbf{s}, d_s) = [\boldsymbol{F}_1(\mathbf{s}, d_s) \quad \boldsymbol{F}_2(\mathbf{s}, d_s)]^T$, has a global maximum if both elements of the $\boldsymbol{\theta}$ are negative per Proposition 4.2.1. Note, that we use the discrete PEARL variant here because array element swap actions is a finite set, and the dynamics has no smoothness guaranties.

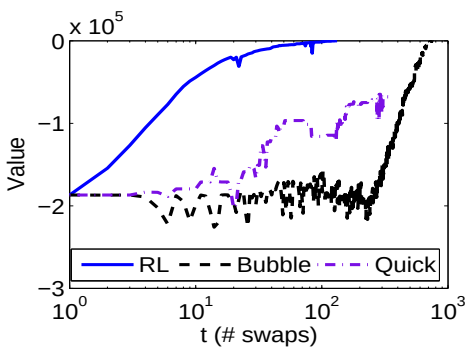
To learn the parametrization $\boldsymbol{\theta}$ we run the AVI with discrete actions. The samples are drawn uniformly from the space of 6-element arrays with values between zero and one, $\mathbf{s}_s \in (0, 1)^6$. The 6-element arrays provide a good balance of sorted and unsorted examples for the learning, determined empirically. We train the agent for 15 iterations. The resulting parametrization, $\boldsymbol{\theta} = [-1.4298 - 0.4216]^T$, has all negative components and meets the progression-to-goal condition in Proposition 4.2.1. The learning curve (Figure 8.1a) shows the averaged cumulative reward when sorting a 25-element array, over the learning iterations. The cumulative reward increases with the iterations. After 7 iterations, the accumu-



(a) Learning curve



(b) Reliable



(c) 5% unreliable

Figure 8.1: Learning curve (a) and value progression over time for an array sorted with RL, bubble, and quick sorts with a reliable (b) and 5% unreliable (c). x-axis is logarithmic in (b) and (c).

lated reward converges, and the agent reaches optimum performance.

We compare the PEARL sort performance to bubblesort, and quicksort methods. The bubblesort repeatedly scans the list and swaps adjacent out-of-order elements. Quicksort selects a pivot element, and creates three sublists that are smaller, equal, and larger then the pivot. It then performs a recursive sort on the lesser and greater elements and merges the three lists together. The two algorithms represent two sorting extremes [3]. Quicksort is a single-pass algorithm making large changes in element placement. On the other hand, bubblesort makes small changes repeatedly until it completes. The dataset consists of 100 arrays with 10, 50, and 100 uniformly randomly drawn elements. We consider sorted,

sorted in reverse, and unsorted elements. In addition, we consider reliable and unreliable comparison routines because modern software increasingly depends on potentially unreliable components [3]. While the reliable routine always compares two numbers correctly, the unreliable one returns an arbitrary answer with a 5% probability.

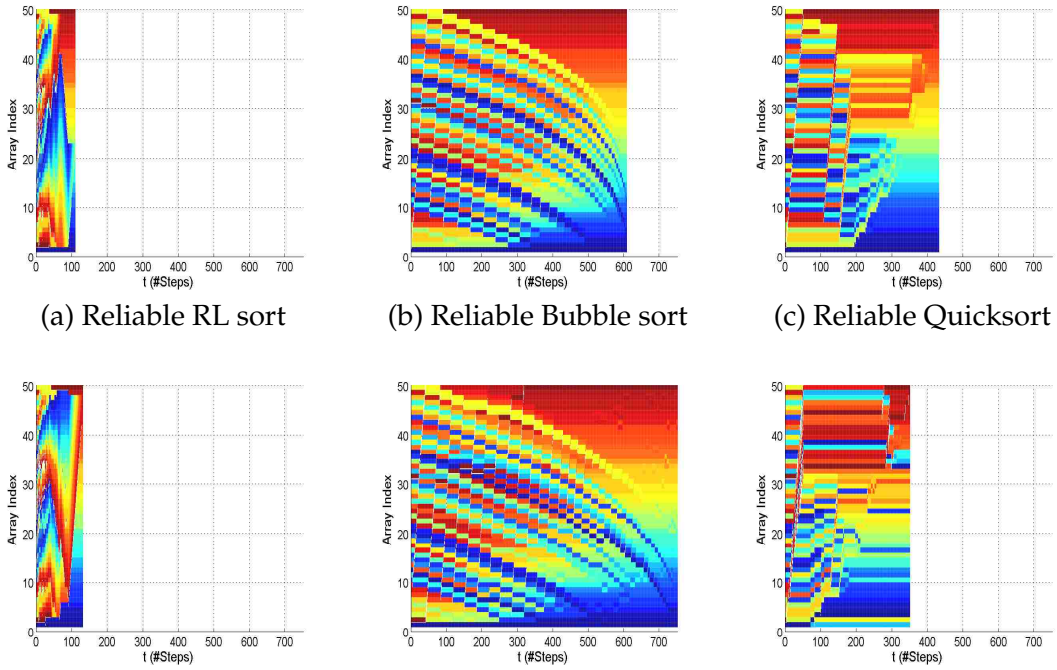
Table 8.1 summarizes the sorting performance. PEARL sort finds a solution with the least changes to the array. This is because the PEARL sort does not make the comparisons in the same way traditional sorting algorithms do. Most of its time is spent calculating features that are quadratic. PEARL sort performs the best on the sorted, and worst on unsorted lists. Bubblesort, on the other hand, performs worst for arrays in reversed order. In contrast, quicksort changes the array a consistent number of times regardless of the array's order.

In the presence of an unreliable comparison (Table 8.1), the number of changes to the array that PEARL sort and quicksort perform do not change significantly (less than two standard deviations). The bubblesort, however, changes the array twice as much. Next, we look into the array error. The error is a Euclidean distance, $d(\mathbf{s}^o, \mathbf{s}^s) = \|\mathbf{s}^o - \mathbf{s}^s\|$, between an outcome of sorting with an unreliable component, $\mathbf{s}_o \in \mathbb{R}^{d_s}$, and the reliably sorted array, $\mathbf{s}_s \in \mathbb{R}^{d_s}$. No error means that the algorithm returned a sorted array, while high error indicates big discrepancies from the sorted array. Note that this similarity metric would have been an ideal feature vector, but it is impossible to calculate it without knowing the sorted array. With 5% noise, the PEARL sort's error remains consistent and small across the datasets and array sizes, although with a relatively high standard deviation. Bubblesort has a comparable error level but makes an order of magnitude more array changes. The quicksort completes with an order of magnitude higher error, for all datasets and array sizes. It is clear that PEARL sort is robust to noise and changes the array the least.

Figure 8.2 visualizes sorting progression of the same array with the three methods, with a reliable component. Runs end at 111, 433, and 608 steps for PEARL sort, quicksort, and bubblesort respectively. Bubblesort makes small, local changes and quicksort's movements around pivot make large steps movement. The PEARL sort (Figure 8.2a) makes a hybrid approach: the array is sorted into progressively larger sorted subgroups. This is because the agent reduces the number of adjacent out-of-order elements at each step. Figures 8.2d-8.2c depict the same array sorted with a faulty comparison. Although PEARL sort takes a different trajectory as the result of the faulty information, it arrives at the goal state, sorted array. The inaccurate information affects bubblesort locally, because its decisions are local, and it too produces a sorted array. On the other hand, quicksort, does not revisit sections of the array it previously sorted. Without the redundancy, it fails to sort the array, explaining the high error rates in Table 8.1. Visualizing the intermediate array values, $V(\mathbf{s}) = \boldsymbol{\theta}^T \boldsymbol{F}(\mathbf{s})$, Figs. 8.1b and 8.1c offer another view into the algorithms' progression. The PEARL sort with the reliable comparison monotonously progresses to the sorted array, empirically showing asymptotic stability based on findings in [41]. Bubblesort and quicksort have setbacks and do not progress monotonically. When the comparison is unreliable, (Figure 8.1c) quicksort fails to reach the same value level as PEARL sort because it stops computation after the first pass. Bubblesort, revisits decisions and corrects the faulty decisions, so it eventually reaches the same value level as the PEARL sort.

8.3 Conclusion

This chapter used the same framework as previously used for robotic tasks to solve array sorting with unreliable comparison components. This required formulation of computing as a nonlinear control problem. The PEARL sorting agent was



(a) Reliable RL sort (b) Reliable Bubble sort (c) Reliable Quicksort

(d) Unreliable RL sort (e) Unreliable Bubble sort (f) Unreliable Quicksort

Figure 8.2: Sorting progression. A 50-element random array sorted with PEARL, bubblesort and quicksort with a reliable comparison (a-c) and 5% unreliable (d-f) comparison components. Time steps are on x-axis, and the array element heatmap is on y-axis. Blue colored are the smallest, and red colored are the largest array elements. Runs end when the array is fully sorted.

compared to two traditional sorting algorithms. The results showed that PEARL sorting agent completes the task with less array element swaps even than the two traditional algorithms.

Table 8.1: Sorting characteristics demonstrating the impact of random initial distance, the array length, and noise in the comparison routine. Measures the number of swaps between the elements and computational time. The results are averaged over 100 trials.

			Reliable comparison		5% Unreliable comparison			
Algorithm	Dataset	Array length	# Swaps μ	σ	# Swaps μ	σ	Error μ	σ
PEARL sort	Sorted	10	1.00	0.00	1.00	0.00	2.12	15.20
		50	1.00	0.00	1.00	0.00	6.22	26.57
		100	1.00	0.00	1.00	0.00	4.82	23.27
	Reversed	10	10.00	0.00	10.83	1.33	4.99	20.70
		50	50.00	0.00	55.88	3.46	0.36	3.57
		100	100.00	0.00	110.31	7.00	3.62	25.32
	Random	10	10.66	2.69	11.34	3.05	5.31	23.16
		50	112.79	7.33	123.59	9.21	6.11	26.02
		100	284.02	9.42	311.38	13.00	0.57	3.97
Bubble	Sorted	10	0.00	0.00	1.96	3.40	0.50	3.53
		50	0.00	0.00	714.94	824.94	0.66	2.68
		100	0.00	0.00	9331.80	2369.31	8.03	4.39
	Reversed	10	45.00	0.00	50.93	4.85	3.42	14.57
		50	1225.00	0.00	2058.29	605.55	1.06	4.03
		100	4950.00	0.00	9980.28	195.54	9.97	4.71
	Random	10	23.19	4.95	28.08	6.73	3.13	19.21
		50	609.82	58.48	1517.21	815.87	0.91	3.34
		100	2466.44	153.20	9836.22	992.49	8.96	4.84
Quicksort	Sorted	10	42.05	4.67	43.19	5.29	53.83	60.25
		50	358.57	25.43	343.96	21.16	176.65	61.70
		100	843.57	63.64	810.67	44.09	245.37	78.74
	Reversed	10	42.45	4.83	43.11	5.44	60.17	53.58
		50	356.89	25.17	350.91	23.44	173.83	66.61
		100	846.15	50.54	826.82	42.72	261.82	77.31
	Random	10	43.83	5.32	42.77	4.92	50.13	52.02
		50	358.22	25.25	348.83	19.99	181.51	60.64
		100	846.99	63.46	816.34	42.29	255.82	74.84

Chapter 9

Conclusions

The research presents a solution to solving the motion-based PBTs for robotic systems with unknown dynamics. The solution, PEARL framework, is founded on RL and MP. The framework works under two-phase batch RL paradigm. Unlike traditional batch RL solutions, the PEARL offers a method for automated feature selection, learning generalization (learning transfer) between its training and acting phases, and computationally efficient policy approximations. The framework takes a PBT problem, and extracts features. It learns to perform a task using Monte Carlo AVI-based learning on small problems. Once the fittest policy is selected PEARL uses it in the acting phase. The acting phase is an on-line or off-line decision maker that plans the tasks execution. The planner is capable of solving problems of the larger scale than used in learning. The method is appropriate for systems with continuous state-space and either discrete or continuous action spaces with unknown system dynamics.

PEARL addresses three major challenges of MP, efficiency, safety, and adaptability. First, the framework addresses efficiency though the feature creation method, learning generalization, and policy approximations. The selected fea-

tures are metric based and project the high-dimensional problems state space into low-dimensional task space. Because they are metric-based, the same features work in spaces of variable dimensions and physical sizes. That allows the framework to learn on small problems, and generalize the learned policies to larger problems during the acting phase. The policy approximations solve a high-dimensional optimization problem efficiently with sampling, enabling tractable decision-making. Second, PEARL addresses safety by selecting a batch RL setting, and performing a formal analysis. The batch setting ensures that there is no exploration in the acting phase. The formal analysis derives conditions on the system dynamics in relation to the task that guarantee an agent's progression towards a goal. Lastly, the polymorphic nature of the features and adaptable policy approximations make the PEARL adaptable to unpredictable disturbances.

To evaluate the PEARL we placed it several different contexts, both with discrete and continuous actions, as a basis planning including for path following and local planning, and even, outside of robotics, to array sorting. PEARL was also shown successful in solving several difficult robotic problems such as Aerial Cargo Delivery, Multi-Agent Pursuit, Rendezvous, and Flying Inverted Pendulum tasks. The method was successful both in simulation and experimentally.

Together, the presented research proposed, analyzed, and evaluated PEARL framework for solving high-dimensional motion-based PBT tasks with theoretical convergence guarantees and when the system dynamic is unknown. The method's limitations are cases when the system dynamics and tasks are incompatible with each other. Namely, there are two conditions when the method is not appropriate. First, PEARL will not produce the policy that solves the task when the combined system task-dynamics landsape is too corrugated. We have seen that on the example of the Flying Inverted Pendulum Task, which we had to solve with two tasks. Second, PEARL is not appropriate when the system does not

contain enough power to accomplish the task, e.g., the acceleration is too weak. Examples are large non-zero mean disturbances such as wind, and tasks such as pendulum swing up that require strong power. Lastly, the method's convergence to the goal criteria for tasks with multiple intensity-reducing features is not well-understood, and should be a topic of further study. Despite the applied method's somewhat restrictive conditions, the results demonstrated high accuracy and fast learning times on the practical applications. The future research can address task classes and formalization of the success conditions for stochastic systems. To extend PEARL to a class of related tasks, the agent would learn several examples and adapt its behavior to any tasks within the class. Another extension would be developing formal analysis of the probability of convergence in for the stochastic systems in order to give an estimate of likelihood of achieving the goal given the relationship between the system dynamics and external disturbances.

Appendix A

Proof for Lemma 5.1.3

Proof. First, to show that there is $\exists a_0 \in [a_l^i, a_u^i]$ such that $Q_{\mathbf{s},i}^{(\boldsymbol{p})}(a) \geq Q(\mathbf{s}, \boldsymbol{p})$, we pick $a = 0$, and directly from the definition, we get $Q_{\mathbf{s},i}^{(\boldsymbol{p})}(0) = Q(\mathbf{s}, \boldsymbol{p})$. As a consequence

$$Q_{\mathbf{s},i}^{(\boldsymbol{p})}(0) \leq Q_{\mathbf{s},i}^{(\boldsymbol{p})}(\hat{a}_i) \tag{A.1}$$

Second, to show that $Q_{\mathbf{s},i}^{(\mathbf{0})}(\hat{a}_i) - V(\mathbf{s}) \geq 0$,

$$\begin{aligned} Q_{\mathbf{s},i}^{(\mathbf{0})}(\hat{a}_i) &\geq Q_{\mathbf{s},i}^{(\mathbf{0})}(0), \text{ from Equation (A.1)} \\ &= \boldsymbol{f}(\mathbf{s})^T \boldsymbol{\Lambda} \boldsymbol{f}(\mathbf{s}) \geq \mathbf{s}\boldsymbol{\Lambda}\mathbf{s}, \text{ due to Equation (5.7)} \\ &= V(\mathbf{s}) \end{aligned}$$

Third, we show $Q(0, \hat{a}_i \boldsymbol{e_i}) - V(0) = 0$. Since, the origin is equilibrium, the dynamics is $\mathbf{D}(\mathbf{0}, \hat{a}_i \boldsymbol{e_i}) = 0$. Thus, $Q(0, \hat{a}_i \boldsymbol{e_i}) - V(0) = 0$. □

Appendix B

Proof for Theorem 5.1.4

Proof. In all three cases, it is sufficient to show that the policy approximations are admissible.

Manhattan Policy: To show that the policy approximation in Equation (5.9) is admissible, for $\mathbf{s} \neq \mathbf{0}$ we use induction by n, $1 \leq n \leq d_a$, with induction hypothesis,

$$\begin{aligned} &\Delta Q(\mathbf{s}, \hat{\boldsymbol{a}}_{\boldsymbol{n}}) \geq 0, \text{ where } \hat{\boldsymbol{a}}_{\boldsymbol{n}} = \sum_{i=1}^{n} \hat{a}_i \boldsymbol{e_i}, \text{ and} \\ &\Delta Q(\mathbf{s}, \hat{\boldsymbol{a}}_{\boldsymbol{n}}) = 0 \Leftrightarrow \\ &\boldsymbol{f}(\mathbf{s})^T \boldsymbol{\Lambda} \boldsymbol{g}_i(\mathbf{s}) \\ &\qquad = 0, \forall i \leq n, \boldsymbol{f}(\mathbf{s})^T \boldsymbol{\Lambda} \boldsymbol{f}(\mathbf{s}) = \mathbf{s}^T \boldsymbol{\Lambda} \mathbf{s} \end{aligned} \tag{B.1}$$

First note that at iteration $1 < n \leq d_a$,

$$\begin{aligned} &\mathbf{D}(\mathbf{s}, \hat{\boldsymbol{a}}_{\boldsymbol{n-1}} + a\boldsymbol{e_n}) = \boldsymbol{f}(\mathbf{s}) + \boldsymbol{g}(\mathbf{s}) \left(\hat{\boldsymbol{a}}_{\boldsymbol{n-1}} + a\boldsymbol{e_n}\right) \\ &= \boldsymbol{f}(\mathbf{s}) + \boldsymbol{g}(\mathbf{s}) \hat{\boldsymbol{a}}_{\boldsymbol{n-1}} + \boldsymbol{g}(\mathbf{s}) a \boldsymbol{e_n} \\ &= \boldsymbol{f}_n(\mathbf{s}) + \boldsymbol{g}_n(\mathbf{s}) a \end{aligned}$$

and

$$\begin{aligned} Q(\mathbf{s}, \boldsymbol{a_n}) &= (\boldsymbol{f}_n(\mathbf{s}) + \boldsymbol{g}_n(\mathbf{s})a)^T \boldsymbol{\Lambda} (\boldsymbol{f}_n(\mathbf{s}) + \boldsymbol{g}_n(\mathbf{s})a) \\ &= \boldsymbol{g}_n(\mathbf{s})^T \boldsymbol{\Lambda} \boldsymbol{g}_n(\mathbf{s})a^2 + 2\boldsymbol{f}_n(\mathbf{s})^T \boldsymbol{\Lambda} \boldsymbol{g}_n(\mathbf{s})a + \boldsymbol{f}_n(\mathbf{s})^T \boldsymbol{\Lambda} \boldsymbol{f}_n(\mathbf{s}) \\ &= p_n a^2 + q_n a + r_n, \quad p_n, q_n, r_n \in \mathbb{R}. \end{aligned} \tag{B.2}$$

Because $\boldsymbol{\Lambda} < 0$, $Q(\mathbf{s}, \boldsymbol{a_n})$ is a quadratic function of one variable with a maximum in

$$\hat{a}_n^* = -\frac{\boldsymbol{g}_n(\mathbf{s})^T \boldsymbol{\Lambda} \boldsymbol{f}_n(\mathbf{s})}{\boldsymbol{g}_n(\mathbf{s})^T \boldsymbol{\Lambda} \boldsymbol{g}_n(\mathbf{s})} \tag{B.3}$$

Applying the induction for $n = 1$, and using Lemma 5.1.3,

$$\begin{aligned} \Delta Q(\mathbf{s}, \hat{\boldsymbol{a}}_\mathbf{1}) &= Q(\mathbf{s}, \hat{a}_1 \boldsymbol{e_1}) - V(\mathbf{s}) \\ &\geq Q(\mathbf{s}, \mathbf{0}) - V(\mathbf{s}) \\ &= \boldsymbol{f}(\mathbf{s})^T \boldsymbol{\Lambda} \boldsymbol{f}(\mathbf{s}) - \mathbf{s}^T \boldsymbol{\Lambda} \mathbf{s} \\ &> 0, \text{ when } \boldsymbol{f}(\mathbf{s})^T \boldsymbol{\Lambda} \boldsymbol{f}(\mathbf{s}) > \mathbf{s}^T \boldsymbol{\Lambda} \mathbf{s}. \end{aligned} \tag{B.4}$$

Given that, $\hat{\boldsymbol{a}}_\mathbf{1} \neq 0 \Leftrightarrow \Delta Q(\mathbf{s}, \hat{\boldsymbol{a}}_\boldsymbol{n}) > \Delta Q(\mathbf{s}, \mathbf{0})$, and assuming $\boldsymbol{f}(\mathbf{s})^T \boldsymbol{\Lambda} \boldsymbol{f}(\mathbf{s}) = \mathbf{s}^T \boldsymbol{\Lambda} \mathbf{s}$, we evaluate $\hat{\boldsymbol{a}}_\mathbf{1} = 0$. From Equation (B.3),

$$\hat{a}_1 = -\frac{\boldsymbol{g}_1(\mathbf{s})^T \boldsymbol{\Lambda} \boldsymbol{f}(\mathbf{s})}{\boldsymbol{g}_1(\mathbf{s})^T \boldsymbol{\Lambda} \boldsymbol{g}_1(\mathbf{s})} = 0 \Leftrightarrow \boldsymbol{g}_1(\mathbf{s})^T \boldsymbol{\Lambda} \boldsymbol{f}(\mathbf{s}) = 0 \tag{B.5}$$

So, the induction hypothesis, Equation (B.1), for $n = 1$ holds. Assuming that Equation (B.1) holds for $1, .., n-1$, and using Lemma 5.1.3,

$$\begin{aligned} \Delta Q(\mathbf{s}, \hat{\boldsymbol{a}}_\boldsymbol{n}) &= Q(\mathbf{s}, \hat{\boldsymbol{a}}_{\boldsymbol{n-1}} + \hat{a}_n \boldsymbol{e_n}) - V(\mathbf{s}) \\ &\geq Q(\mathbf{s}, \hat{\boldsymbol{a}}_{\boldsymbol{n-1}} + \mathbf{0}) - V(\mathbf{s}) \\ &= \Delta Q(\mathbf{s}, \hat{\boldsymbol{a}}_{\boldsymbol{n-1}}) \text{ from ind. hyp. Equation (B.1)} \\ &> 0 \text{ when } \boldsymbol{f}(\mathbf{s})^T \boldsymbol{\Lambda} \boldsymbol{f}(\mathbf{s}) > \mathbf{s}^T \boldsymbol{\Lambda} \mathbf{s}. \end{aligned}$$

Similarly, assuming $\boldsymbol{f}(\mathbf{s})^T \boldsymbol{\Lambda} \boldsymbol{f}(\mathbf{s}) = \mathbf{s}^T \boldsymbol{\Lambda} \mathbf{s}$,

$$\begin{aligned}
&\Delta Q(\mathbf{s}, \hat{\boldsymbol{a}}_{\boldsymbol{n}}) = 0 \Leftrightarrow \\
&\hat{a}_n = -\frac{\boldsymbol{g}_n(\mathbf{s})^T \boldsymbol{\Lambda} \boldsymbol{f}_n(\mathbf{s})}{\boldsymbol{g}_n(\mathbf{s})^T \boldsymbol{\Lambda} \boldsymbol{g}_n(\mathbf{s})} = 0, \text{ and } \Delta Q(\mathbf{s}, \hat{\boldsymbol{a}}_{\boldsymbol{n-1}}) = 0
\end{aligned}$$

Since $\Delta Q(\mathbf{s}, \hat{\boldsymbol{a}}_{\boldsymbol{n-1}}) = 0 \Leftrightarrow \hat{\boldsymbol{a}}_{\boldsymbol{n-1}} = \mathbf{0}$, means that $\boldsymbol{f}_{\boldsymbol{n}}(\mathbf{s}) = \boldsymbol{f}(\mathbf{s}) + \boldsymbol{g}(\mathbf{s}) \hat{\boldsymbol{a}}_{\boldsymbol{n-1}} = \boldsymbol{f}(\mathbf{s})$,

$$\begin{aligned}
&\Delta Q(\mathbf{s}, \hat{\boldsymbol{a}}_{\boldsymbol{n}}) = 0 \Leftrightarrow \\
&\boldsymbol{g}_n(\mathbf{s})^T \boldsymbol{\Lambda} \boldsymbol{f}(\mathbf{s}) = 0, \text{ and } \Delta Q(\mathbf{s}, \hat{\boldsymbol{a}}_{\boldsymbol{n-1}}) = 0 \Leftrightarrow \\
&\boldsymbol{g}_i(\mathbf{s})^T \boldsymbol{\Lambda} \boldsymbol{f}(\mathbf{s}) = 0, \text{ for } 1 \leq i \leq n
\end{aligned}$$

For $n = d_a$, the policy gain $\Delta Q(\mathbf{s}, \hat{\boldsymbol{a}}_{\boldsymbol{d_a}}) = 0 \Leftrightarrow \boldsymbol{f}(\mathbf{s})^T \boldsymbol{\Lambda} \boldsymbol{f}(\mathbf{s}) = \mathbf{s}^T \boldsymbol{\Lambda} \mathbf{s}$, and $\boldsymbol{g}_i(\mathbf{s})^T \boldsymbol{\Lambda} \boldsymbol{f}(\mathbf{s}) = 0$, for $1 \leq i \leq d_u$. But, that is contradiction with the controllability assumption Equation (5.4), thus $\Delta Q(\mathbf{s}, \hat{\boldsymbol{a}}_{\boldsymbol{d_u}}) > 0$, when $\mathbf{s} \neq \mathbf{0}$.

When $\mathbf{s} = \mathbf{0}$, we get directly from Lemma 5.1.3, $\Delta Q(\mathbf{0}, \hat{\boldsymbol{a}}_{\boldsymbol{d_a}}) = 0$. This completes the proof that Manhattan Policy Equation (5.9) is admissible, and therefore the equilibrium is asymptotically stable.

Convex Sum Policy, Equation (5.10)*:* Following the same reasoning as for the first step of the Manhattan Policy, Equations (B.4) and (B.5), we get that for all $1 \leq n \leq d_a$, $\Delta Q(\mathbf{s}, \hat{a}_n \boldsymbol{e_n}) \geq 0$, where $\hat{a}_n \boldsymbol{e_n} = \text{argmax}_{a_l^n \leq a \leq a_u^n} Q_{\mathbf{s},n}^{(\mathbf{0})}(a)$ and the equality holds only when

$$\Delta Q(\mathbf{s}, \hat{a}_n \boldsymbol{e_n}) = 0 \Leftrightarrow \boldsymbol{f}(\mathbf{s})^T \boldsymbol{\Lambda} \boldsymbol{g}_n(\mathbf{s}) = 0, \boldsymbol{f}(\mathbf{s})^T \boldsymbol{\Lambda} \boldsymbol{f}(\mathbf{s}) = \mathbf{s}^T \boldsymbol{\Lambda} \mathbf{s} \tag{B.6}$$

To simplify the notation, let $Q_i = \Delta Q(\mathbf{s}, \hat{a}_n \boldsymbol{e_n})$, and $Q_0 = 0$. Without loss of generality, assume that

$$Q_0 \leq Q_1 \leq ... \leq Q_{d_u},\ n = 1, ..., d_u.$$

The equality only holds when Equation (B.6) holds for all $n = 1, ..., d_a$ which is contradiction with the Equation (5.4). Thus, there must be at least one $1 \leq n_0 \leq d_a$, such that $Q_{n_0-1} < Q_{n_0}$, and consequently $0 < Q_{d_u}$.

Lastly, we need to show that the combined action $\hat{\mathbf{a}}$ calculated with Equation (5.10) is admissible, i.e., $\Delta Q(\mathbf{s}, \hat{\mathbf{a}}) > 0$. It suffices to show that $\hat{\mathbf{a}}$ is inside the ellipsoid $\check{Q}_0 = \{\mathbf{a} | Q(\mathbf{s}, \mathbf{a}) \geq Q_0\}$. Similarly, $Q_1, ..., Q_{d_u}$ define a set of concentric ellipsoids

$$\check{Q}_i = \{\mathbf{a} | Q(\mathbf{s}, \mathbf{a}) \geq Q_i\}, \;\; i = 1, ..., d_u.$$

Since, $\check{Q}_0 \supseteq \check{Q}_1 \supseteq ... \supseteq \check{Q}_{d_u}$, and $\forall i, \hat{\boldsymbol{u}}_{\boldsymbol{i}} \in \check{Q}_i \implies \hat{\boldsymbol{u}}_{\boldsymbol{i}} \in \check{Q}_0$. Because ellipsoid $\check{Q}_0$ is convex, the convex combination of points inside it, Equation (5.10) belongs to it as well. Since, at least one ellipsoid must be a true subset of $\check{Q}_0$, which completes the asymptotic stability proof.

Axial Sum Policy approximation, Equation (5.11): is admissible because Convex Sum Policy, Equation (5.10), is admissible. Formally, $\Delta Q(\mathbf{s}, \boldsymbol{\pi}_{\boldsymbol{s}}^{Q}(\mathbf{s})) \geq \Delta Q(\mathbf{s}, \boldsymbol{\pi}_{\boldsymbol{c}}^{Q}(\mathbf{s})) \geq 0$.

□

Appendix C

Axial Sum Policy Optimality

Proposition C.0.1. *When $\boldsymbol{g}(\mathbf{s})$ is an independent input matrix, $\boldsymbol{C} = \boldsymbol{I}$, and state-value function parameterization $\boldsymbol{\Theta}$ is negative definite, then Axial Sum Policy, Equation* (5.11)*, is optimal with respect to the state-value function, Equation* (4.2)*.*

Proof. The optimal input $\boldsymbol{u}^*$ is a solution to $\frac{\partial Q(\mathbf{s},u_i)}{\partial u_i} = 0$, and $\hat{\boldsymbol{a}}$ is a solution to $\frac{\mathrm{d}Q^{(\mathbf{0})}_{\mathbf{s},i}(a)}{\mathrm{d}a}$ at state $\mathbf{s}$ with respect to the state-value function, Equation (4.2). To show that the Axial Sum Policy is optimal, $\boldsymbol{u}^* = \hat{\boldsymbol{a}}$, it is enough to show that

$$\frac{\mathrm{d}Q(\mathbf{s}, u_i)}{\mathrm{d}u_i} = \frac{\mathrm{d}Q^{(\mathbf{0})}_{\mathbf{s},i}(a)}{\mathrm{d}a}$$

. This is the case when Q has the form of

$$Q(\mathbf{s}, \mathbf{a}) = \sum_{i=1}^{d_x} (p_{x_i} u_i^2 + q_{x_i} u_i + r_{x_i}),$$

for some $p_{x_i}, q_{x_i}, r_{x_i} \in \mathbb{R}$ that depend on the current state $\mathbf{s}$. In the Proposition

5.1.1 we showed that

$$\begin{aligned} Q(\mathbf{s}, \mathbf{a}) &= (\boldsymbol{f}(\mathbf{s}) + \boldsymbol{g}(\mathbf{s})\mathbf{a}))^T \boldsymbol{\Theta} (\boldsymbol{f}(\mathbf{s}) + \boldsymbol{g}(\mathbf{s})\mathbf{a}) \\ &= \sum_{i=1}^{d_x} \theta_i \left(\sum_{j=1}^{d_u} g_{ij}(\mathbf{s}) u_j + f_i(\mathbf{s}) \right)^2 . \end{aligned}$$

Since there is a single nonzero element j_i in row i of matrix $\boldsymbol{g}$,

$$\begin{aligned} Q(\mathbf{s}, \mathbf{a}) &= \sum_{i=1}^{d_x} \left(\theta_i (g_{j_i}(\mathbf{s}) u_{j_i} + f_{j_i}(\mathbf{s}) \right)^2 \\ &= \sum_{i=1}^{d_x} (\theta_i g_{j_i}^2(\mathbf{s}) u_{j_i}^2 + 2\theta_i f_{j_i}(\mathbf{s}) g_{j_i}(\mathbf{s}) u_{j_i} + f_{j_i}^2(\mathbf{s})) \end{aligned}$$

After rearranging, $Q(\mathbf{s}, \mathbf{a}) = \sum_{i=1}^{d_x} (p_{x_i} u_i^2 + q_{x_i} u_i + r_{x_i})$.

□

Acronyms

AVI	Approximate Value Iteration
CAFVI	Continuous Action Fitted Value Iteration
DOF	Degrees Of Freedom
HOOT	Hierarchical Optimistic Optimization Applied To Trees
LSAPA	Least Squares Axial Policy Approximation
MDP	Markov Decision Process
MP	Motion Planning
PBT	Preference Balancing Task
PEARL	PrEference Appraisal Reinforcement Learning
PRM	Probabilistic Roadmap
PSD	Power Spectral Density
RL	Reinforcement Learning
RRT	Rapidly Exploring Random Trees
SCAFVI	Stochastic Continuous Action Fitted Value Iteration
UAV	Unmanned Aerial Vehicle

Glossary

action	A signal that changes state of the system
action-value function	Value of performing an action from a given state, $Q(\mathbf{s}, \mathbf{a}) = V(\mathbf{s}')$, $\mathbf{s}'$ is the resulting state of applying action $\mathbf{a}$ to state $\mathbf{s}$
Array Sorting Task	A taks requiring an arbitrary array to be sorted in ascending order
AVI	Reinforcement learning algorithm for MDPs with continuous states and discrete actions. The algorithm works with state-value function V
Axial Sum Policy	A policy approximation that selects a better action between Convex Sum Policy and standard vector sum
Balanced Hover	A task brings a quadrotor to a hover while balancing the inverted pendulum

bootstrapping	A method for estimating new values using old estimates
Bounded Load Displacement	A trajectory, or a flight, in which the swing is bounded for the duration of the entire trajectory
CAFVI	Adaptation of Fitted Value Iteration algorithm for MDPs with continuous both states and actions. The algorithm works with both state-value and action-value functions
Cargo Delivery Task	A task that requires Bounded load displacement trajectory
C-free	Collision-free portion of a configuration space
Coffee Delivery Task	A Coffee Shop variant of the Aerail Cargo Delivery Task
continuous set	Set with continuum ($\mathfrak{c}$) cardinality
control-affine	Nonlinear system, linear in input
control Lyapunov function	A positive definite function of state that is sufficient for showing Lyapunov stability of origin

Convex Sum Policy	A policy approximation that selects actions on each axis concurrently, and then takes a convex sum of the axes selections
C-space	Configuration space, space of all poses a robot can take
decision-making	Determining an action to perform
deterministic process	Process where action's effect on a state is uniquely determined
dynamic programming	A method for solving finite, completely known discrete-time Markov decision processes
Flying Inverted Pendulum	A task that balances the inverted pendulum on a flying hovering quadrotor
greedy	Greedy policy w.r.t. a state-value function V, $\pi^V(S) = \text{argmax}_{\mathbf{a}\in A} V(\mathbf{s}')$, where $\mathbf{s}' = T(\mathbf{s}, \mathbf{a})$
Initial Balance	A task that balances the inverted pendulum on a flying quadrotor without slowing down the quadrotor
Load Swing	Same as Load Displacement
Load Displacement	Displacement of the suspended load from a vertical axis

LSAPA	An adaptive policy approximation for stochastic MDPs with continuous both states and actions.
Manhattan Policy	A policy approximation that selects actions sequentially on each axis
Markov property	Effect of an action depends only on current state
Minimal Residual Oscillations Task	A task that requires Minimal Residual Oscillations trajectory
Minimal Residual Oscillations trajectory	A trajectory, or a flight, with minimal swing at the end
MP	A method for discovering a feasible collision-free sequence of robot's positions
Multi-Agent Pursuit Task	A taks requiring multi-agent system to pursure a prey
Package Delivery Task in Cities	Aerail Cargo Delivery Task applied for package delivery in cities
path	Sequence of robot's positions without regard to time
PRM	A sampling-based motion planning method

probability transition function	Probability distribution over states when one action is applied to state
Rendezvous Task	A task requiring a quadrotor and a ground robot to meet as soon as possible without distrubign the quadrotor's suspended load
return	Cumulative reward over agent's life-time
reward	Immediate obervable, numeric feedback
RRT	A sampling-based motion planning method
SCAFVI	Adaptation of Continuous Action Fitted Value Iteration algorithm for problems in presence of stochastic noise
state	Information that describes the system
state-value function	Cummulative, discounted reward, $V(\mathbf{s}) = \sum_{t=0}^{\infty} \gamma^t R(t)$
Swing-free Path-following	A task that requires minimal residual oscillations while the trajectory closely follows a path
Swing-free trajectory	Same as Bounded load displacement
time step	Time between two state observations

trajectory	Sequence of time and robot's position pairs
Value Iteration	A dynamic programming method that iterates over state-value functions to solve a finite, completely known discrete-time Markov Decision Processes

Symbols

a	Univariate action variable
A	Action set
α	Learning constant $0 < \alpha \leq 1$
$\mathbf{a}$	Action $\mathbf{a} \in A$
$\mathbf{D}$	System dynamics
d_a	Dimensionality of action space i.e., $A \subset \mathbb{R}^{d_a}$
δ	Proximity constant
Δt	Time step
d_f	Number of features
d_n	Number of action samples on an axis
d_r	Number of agents
d_s	Dimensionality of state space i.e., $S \subset \mathbb{R}^{d_s}$
$\boldsymbol{e_i}$	Unit vector of i^th axes
ϵ	Swing constant
$\boldsymbol{\eta}$	$\boldsymbol{\eta}(t) = [\phi(t)\ \omega(t)]^T$ position of the suspended load in polar coordinates
$\dot{\boldsymbol{\eta}}$	$\dot{\boldsymbol{\eta}}(t) = [\dot{\phi}(t)\ \dot{\omega}(t)]^T$ velocity of the suspended load in polar coordinates
$\boldsymbol{f}$	Controlled dynamics

$\boldsymbol{F}$	Feature vector
$\boldsymbol{g}$	System drift
γ	Discount constant $0 < \gamma \leq 1$
k	Number of actions to select
$\mathcal{M}$	Markov decision process
$(S, A, \mathbf{D}, R)$	Markov decision process tuple
n_o	Number of objectives
$\boldsymbol{o}$	Task objective point
$\boldsymbol{p}$	projection - $\boldsymbol{p}_j^{\boldsymbol{o}_i}(\mathbf{s})$ is a projection of j^{th} robot's state onto minimal subspace that contains $\boldsymbol{o}_i$
π	Policy $\pi : S \rightarrow A$
π^*	Optimal policy $\pi^* = \max \pi$
Q	Action-value function $V : S \times A \rightarrow \mathbb{R}$
R	Reward $R : S \rightarrow \mathbb{R}$
$\mathbf{s}$	State $\mathbf{s} \in S$
S	State set
$\dot{\boldsymbol{s}}$	State $\dot{\mathbf{s}} \in S$
$\ddot{\boldsymbol{s}}$	State $\ddot{\mathbf{s}} \in S$
$\mathbf{s}_g$	Goal state $\mathbf{s} \in S$
$\mathbf{s}_s$	Start state $\mathbf{s} \in S$
T	Trajectory duration
$\boldsymbol{\theta}$	Feature weights
V	State-value function $V : S \rightarrow \mathbb{R}$
V^*	Optimal state-value function $V^* = \max V$
$\hat{V}$	Approximate state-value function

References

[1] Pieter Abbeel. *Apprenticeship learning and reinforcement learning with application to robotic control*. PhD thesis, Stanford University, Stanford, CA, USA, 2008.

[2] Pieter Abbeel, Adam Coates, and Andrew Y. Ng. Autonomous helicopter aerobatics through apprenticeship learning. *International Journal of Robotics Research (IJRR)*, 29(13):1608–1639, November 2010.

[3] David H. Ackley. Beyond efficiency. *Commun. ACM*, 56(10):38–40, October 2013.

[4] A. Agha-mohammadi, Suman Chakravorty, and Nancy Amato. FIRM: Sampling-based feedback motion planning under motion uncertainty and imperfect measurements. *Int. J. Robot. Res.*, 33(2):268–304, 2014.

[5] M.J. Agostini, G.G. Parker, H. Schaub, K. Groom, and III Robinett, R.D. Generating swing-suppressed maneuvers for crane systems with rate saturation. *Control Systems Technology, IEEE Transactions on*, 11(4):471 – 481, July 2003.

[6] Ibrahim Al-Bluwi, Thierry Simon, and Juan Corts. Motion planning algorithms for molecular simulations: A survey. *Computer Science Review*, 6(4):125 – 143, 2012.

[7] A. Al-Tamimi, F.L. Lewis, and M. Abu-Khalaf. Discrete-time nonlinear hjb solution using approximate dynamic programming: Convergence proof. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 38(4):943–949, 2008.

[8] Kostas Alexis, George Nikolakopoulos, and Anthony Tzes. Constrained-control of a quadrotor helicopter for trajectory tracking under wind-gust

disturbances. In *MELECON 2010-2010 15th IEEE Mediterranean Electrotechnical Conference*, pages 1411–1416. IEEE, 2010.

[9] R. Alterovitz, T. Simeon, and K. Goldberg. The stochastic motion roadmap: A sampling framework for planning with markov motion uncertainty. In *Proc. Robotics: Sci. Sys. (RSS)*, page 246 253, Atlanta, GA, USA, June 2007.

[10] Nancy M. Amato, O. Burchan Bayazit, Lucia K. Dale, Christopher Jones, and Daniel Vallejo. Obprm: An obstacle-based prm for 3d workspaces. In *Proc. Int. Wkshp. on Alg. Found. of Rob. (WAFR)*, pages 155–168, March 1998.

[11] C.W. Anderson, P.M. Young, M.R. Buehner, J.N. Knight, K.A. Bush, and D.C. Hittle. Robust reinforcement learning control using integral quadratic constraints for recurrent neural networks. *Neural Networks, IEEE Transactions on*, 18(4):993–1002, 2007.

[12] A. Antos, Cs. Szepesvari, and R. Munos. Fitted Q-iteration in continuous action-space MDPs. In J.C. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems 20*, pages 9–16, Cambridge, MA, 2007. MIT Press.

[13] Brenna D. Argall, Brett Browning, and Manuela M. Veloso. Teacher feedback to scaffold and refine demonstrated motion primitives on a mobile robot. *Robotics and Autonomous Systems*, 59(34):243 – 255, 2011.

[14] Brenna D Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robotics and Autonomous Systems*, 57(5):469–483, 2009.

[15] AscTec. Ascending Technologies GmbH, 2013. http://www.asctec.de/.

[16] Karl J. Astrom. *Introduction to Stochastic Control Theory (Dover Books on Electrical Engineering)*. Dover Publications, 2006.

[17] Karl J. Astrom and Richard M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, April 2008.

[18] Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. In James A. Anderson and Edward Rosenfeld, editors, *Neurocomputing: foundations of research*, chapter Neuronlike adaptive elements that can solve difficult learning control problems, pages 535–549. MIT Press, Cambridge, MA, USA, 1988.

[19] Len Bass, Paul Clements, and Rick Kazman. *Software Architecure in Practice, 2nd ed.* Addison Wesley, 2003.

[20] Richard Ernest Bellman. *Dynamic Programming*. Dover Publications, Incorporated, 1957.

[21] M. Bernard and K. Kondak. Generic slung load transportation system using small size helicopters. In *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, pages 3258 –3264, may 2009.

[22] D. P. Bertsekas. *Dynamic Programming and Optimal Control, Vol. II, 3rd Ed.* Athena Scientific, Belmont, MA, 2007.

[23] Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, 1st edition, 1996.

[24] Shubhendu Bhasin, Nitin Sharma, Parag Patre, and Warren Dixon. Asymptotic tracking by a reinforcement learning-based adaptive critic controller. *J of Control Theory and Appl*, 9(3):400–409, 2011.

[25] Adam Bry and Nicholas Roy. Rapidly-exploring random belief trees for motion planning under uncertainty. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 723–730. IEEE, 2011.

[26] Sébastien Bubeck, Rémi Munos, Gilles Stoltz, and Csaba Szepesvári. X-armed bandits. *J. Mach. Learn. Res.*, 12:1655–1695, July 2011.

[27] L. Buşoniu, R. Babuška, B. De Schutter, and D. Ernst. *Reinforcement Learning and Dynamic Programming Using Function Approximators*. CRC Press, Boca Raton, Florida, 2010.

[28] L. Busoniu, A. Daniels, R. Munos, and R. Babuska. Optimistic planning for continuous-action deterministic systems. In *Adaptive Dynamic Programming And Reinforcement Learning (ADPRL), 2013 IEEE Symposium on*, pages 69–76, April 2013.

[29] Zheng Chen and Sarangapani Jagannathan. Generalized hamilton–jacobi–bellman formulation-based neural network control of affine nonlinear discrete-time systems. *Neural Networks, IEEE Transactions on*, 19(1):90–106, 2008.

[30] Tao Cheng, Frank L. Lewis, and Murad Abu-Khalaf. A neural network solution for fixed-final time optimal control of nonlinear systems. *Automatica*, 43(3):482–490, 2007.

[31] Jean claude Latombe. Motion planning: A journey of robots, molecules, digital actors, and other artifacts. *International Journal of Robotics Research*, 18:1119–1128, 1999.

[32] P. Corke. *Robotics, Vision and Control: Fundamental Algorithms in MATLAB*. Springer Tracts in Advanced Robotics. Springer, 2011.

[33] Jonathan A DeCastro and Hadas Kress-Gazit. Guaranteeing reactive high-level behaviors for robots with complex dynamics. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 749–756. IEEE, 2013.

[34] Vishnu R Desaraju and Jonathan P How. Decentralized path planning for multi-agent teams in complex environments using rapidly-exploring random trees. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 4956–4961. IEEE, 2011.

[35] T. Dierks and S. Jagannathan. Online optimal control of affine nonlinear discrete-time systems with unknown internal dynamics by using time-based policy update. *IEEE Transactions on Neural Networks and Learning Systems*, 23(7):1118–1129, 2012.

[36] Damien Ernst, Mevludin Glavic, Pierre Geurts, and Louis Wehenkel. Approximate value iteration in the reinforcement learning context. application to electrical power system control. *International Journal of Emerging Electric Power Systems*, 3(1):1066.1–1066.37, 2005.

[37] A. M. Farahmand, R. Munos, and Cs. Szepesvári. Error propagation for approximate policy and value iteration. In *Advances in Neural Information Processing Systems*, pages 568–576, Dec. 2010.

[38] Aleksandra Faust, Nicolas Malone, and Lydia Tapia. Planning constraint-balancing motions with stochastic disturbances. In *under submission*.

[39] Aleksandra Faust, Ivana Palunko, Patricio Cruz, Rafael Fierro, and Lydia Tapia. Automated aerial suspended cargo delivery through reinforcement learning. *Adaptive Motion Planning Research Group Technical Report TR13-001*, 2013. under submission.

[40] Aleksandra Faust, Ivana Palunko, Patricio Cruz, Rafael Fierro, and Lydia Tapia. Learning swing-free trajectories for UAVs with a suspended load. In *IEEE International Conference on Robotics and Automation (ICRA), Karlsruhe, Germany*, pages 4887–4894, May 2013.

[41] Aleksandra Faust, Peter Ruymgaart, Molly Salman, Rafael Fierro, and Lydia Tapia. Continuous action reinforcement learning for underactuated dynamical system control. *Acta Automatica Sinica, in press*, 2014.

[42] Rafael Figueroa, Aleksandra Faust, Patricio Cruz, Lydia Tapia, and Rafael Fierro. Reinforcement learning for balancing a flying inverted pendulum. In *Proc. The 11th World Congress on Intelligent Control and Automation*, July 2014.

[43] M. Glavic, D. Ernst, and L. Wehenkel. Combining a stability and a performance-oriented control in power systems. *Power Systems, IEEE Transactions on*, 20(1):525–526, 2005.

[44] I. Grondman, L. Busoniu, G. A D Lopes, and R. Babuska. A survey of actor-critic reinforcement learning: Standard and natural policy gradients. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 42(6):1291–1307, Nov 2012.

[45] Wassim M. Haddad and VijaySekhar Chellaboina. *Nonlinear Dynamical Systems and Control*. Princeton University Press, 2008.

[46] Dick Hamlet and Joe Maybee. *The Engineering of Software*. Addison Wesley, 2001.

[47] Hado Hasselt. Reinforcement learning in continuous state and action spaces. In Marco Wiering and Martijn Otterlo, editors, *Reinforcement Learning*, volume 12 of *Adaptation, Learning, and Optimization*, pages 207–251. Springer Berlin Heidelberg, 2012.

[48] Kris Hauser, Timothy Bretl, Jean Claude Latombe, and Brian Wilcox. Motion planning for a sixlegged lunar robot. In *The Seventh International Workshop on the Algorithmic Foundations of Robotics*, pages 16–18, 2006.

[49] M. Hehn and R. D'Andrea. A flying inverted pendulum. In *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, pages 763–770. IEEE, 2011.

[50] Todd Hester and Peter Stone. Learning and using models. In Marco Wiering and Martijn van Otterlo, editors, *Reinforcement Learning: State of the Art*. Springer Verlag, Berlin, Germany, 2011.

[51] Todd Hester and Peter Stone. TEXPLORE: real-time sample-efficient reinforcement learning for robots. *Machine Learning*, 90(3):385–429, 2013.

[52] David Hsu, Robert Kindel, Jean-Claude Latombe, and Stephen M. Rock. Randomized kinodynamic motion planning with moving obstacles. *Int. J. Robot. Res.*, 21(3):233–256, 2002.

[53] A. Isidori. *Nonlinear Control Systems, 2nd Ed.* Springer-Verlag, Berlin, 1989.

[54] Yu Jiang and Zhong-Ping Jiang. Computational adaptive optimal control for continuous-time linear systems with completely unknown dynamics. *Automatica*, 48(10):2699–2704, October 2012.

[55] Jeff Johnson and Kris Hauser. Optimal acceleration-bounded trajectory planning in dynamic environments along a specified path. In *ICRA*, pages 2035–2041, 2012.

[56] L. E. Kavraki, P. Švestka, J. C. Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Trans. Robot. Automat.*, 12(4):566–580, August 1996.

[57] H. Kawano. Study of path planning method for under-actuated blimp-type uav in stochastic wind disturbance via augmented-mdp. In *Advanced Intelligent Mechatronics (AIM), 2011 IEEE/ASME International Conference on*, pages 180–185, July 2011.

[58] Hiroshi Kawano. Three dimensional obstacle avoidance of autonomous blimp flying in unknown disturbance. In *Proc. IEEE Int. Conf. Intel. Rob. Syst. (IROS)*, pages 123–130. IEEE, 2006.

[59] H.K. Khalil. *Nonlinear Systems*. Prentice Hall, 1996.

[60] H. Kimura. Reinforcement learning in multi-dimensional state-action space using random rectangular coarse coding and gibbs sampling. In *Proc. IEEE Int. Conf. Intel. Rob. Syst. (IROS)*, pages 88–95, 2007.

[61] J Kober, D. Bagnell, and J. Peters. Reinforcement learning in robotics: A survey. *International Journal of Robotics Research*, 32(11):1236–1272, 2013.

[62] Mangal Kothari and Ian Postlethwaite. A probabilistically robust path planning algorithm for UAVs using rapidly-exploring random trees. *Journal of Intelligent & Robotic Systems*, pages 1–23, 2012.

[63] J.J. Kuffner and S.M. LaValle. RRT-connect: An efficient approach to single-query path planning. In *IEEE International Conference on Robotics and Automation*, volume 2, pages 995 –1001 vol.2, 2000.

[64] Vijay Kumar and Nathan Michael. Opportunities and challenges with autonomous micro aerial vehicles. *The International Journal of Robotics Research*, 31(11):1279–1291, September 2012.

[65] Tobias Kunz and Mike Stilman. Manipulation planning with soft task constraints. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1937–1942, October 2012.

[66] A. Lambert and D. Gruyer. Safe path planning in an uncertain-configuration space. In *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, volume 3, pages 4185–4190 vol.3, 2003.

[67] Sascha Lange, Thomas Gabel, and Martin Riedmiller. Batch reinforcement learning. In Marco Wiering and Martijn van Otterlo, editors, *Reinforcement Learning: State of the Art*. Springer Verlag, Berlin, Germany, 2011.

[68] Jean-Claude Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.

[69] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006.

[70] Steven M. Lavalle and James J. Kuffner. Rapidly-exploring random trees: Progress and prospects. In *Algorithmic and Computational Robotics: New Directions*, pages 293–308, 2000.

[71] Alessandro Lazaric, Marcello Restelli, and Andrea Bonarini. Reinforcement learning in continuous action spaces through sequential monte carlo methods. *Advances in neural information processing systems*, 20:833–840, 2008.

[72] J. Levine. *Analysis and Control of Nonlinear Systems: A Flatness-based Approach*. Mathematical Engineering. Springer, 2010.

[73] S. Lupashin and R. DAndrea. Adaptive open-loop aerobatic maneuvers for quadrocopters. In *World Congress*, volume 18, pages 2600–2606, 2011.

[74] S. Lupashin, A. Schöllig, M. Sherback, and R. D'Andrea. A simple learning strategy for high-speed quadrocopter multi-flips. In *IEEE International Conference on Robotics and Automation*, pages 1642–1648, May 2010.

[75] Anirudha Majumdar and Russ Tedrake. Robust online motion planning with regions of finite time invariance. In *Algorithmic Foundations of Robotics X*, pages 543–558. Springer, 2013.

[76] Nicholas Malone, Kasra Manavi, John Wood, and Lydia Tapia. Construction and use of roadmaps that incoroporate worksapce modeling errors. In *Proc. IEEE Int. Conf. Intel. Rob. Syst. (IROS)*, pages 1264–1271, Nov 2013.

[77] Nicholas Malone, Brandon Rohrer, Lydia Tapia, Ron Lumia, and John Wood. Implementation of an embodied general reinforcement learner on a serial link manipulator. In *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, pages 862–869, May 2012.

[78] Nick Malone, Aleksandra Faust, Brandon Rohrer, Ron Lumia, John Wood, and Lydia Tapia. Efficient motion-based task learning for a serial link manipulator. *Transactions on Control and Mechanical Systems*, 3(1), 2014.

[79] Nick Malone, Aleksandra Faust, Brandon Rohrer, John Wood, and Lydia Tapia. Efficient motion-based task learning. In *Robot Motion Planning Workshop, IEEE/RSJ International Conference on Intelligent Robots and Systems*, October 2012.

[80] Nick Malone, Kendra Lesser, Meeko Oishi, and Lydia Tapia. Stochastic reachability based motion planning for multiple moving obstacle avoidance. In *International Conference on Hybrid Systems: Computation and Control*, pages 51–60, apr 2014.

[81] Chris Mansley, Ari Weinstein, and Michael Littman. Sample-based planning for continuous action markov decision processes. In *Proc. of Int. Conference on Automated Planning and Scheduling*, 2011.

[82] MARHES. Multi-Agent, Robotics, Hybrid, and Embedded Systems Laboratory, Department of Computer and Electrical Engineering, University of New Mexico. http://marhes.ece.unm.edu, February 2013.

[83] Ahmad A. Masoud. A harmonic potential field approach for planning motion of a uav in a cluttered environment with a drift field, Orlando, FL, USA. In *50th IEEE Conference on Decision and Control and European Control Conference*, pages 7665–7671, dec 2011.

[84] H. Brendan McMahan, Maxim Likhachev, and Geoffrey J. Gordon. Bounded real-time dynamic programming: Rtdp with monotone upper bounds and performance guarantees. In *ICML05*, pages 569–576, 2005.

[85] S. Mehraeen and S. Jagannathan. Decentralized nearly optimal control of a class of interconnected nonlinear discrete-time systems by using online Hamilton-Bellman-Jacobi formulation. In *International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2010.

[86] S. Mehraeen and S. Jagannathan. Decentralized optimal control of a class of interconnected nonlinear discrete-time systems by using online Hamilton-Jacobi-Bellman formulation. *IEEE Transactions on Neural Networks*, 22(11):1757–1769, 2011.

[87] Daniel Mellinger, Quentin Lindsey, Michael Shomin, and Vijay Kumar. Design, modeling, estimation and control for aerial grasping and manipulation. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2668 –2673, sept. 2011.

[88] Daniel Mellinger, Nathan Michael, and Vijay Kumar. Trajectory generation and control for precise aggressive maneuvers with quadrotors. *The International Journal of Robotics Research*, 31(5):664–674, 2012.

[89] N. Michael, D. Mellinger, Q. Lindsey, and V. Kumar. The grasp multiple micro-UAV testbed. *Robotics Automation Magazine, IEEE*, 17(3):56 –65, sept. 2010.

[90] Hamidreza Modares, Mohammad-Bagher Naghibi Sistani, and Frank L. Lewis. A policy iteration approach to online optimal control of continuous-time constrained-input systems. *ISA Transactions*, 52(5):611–621, 2013.

[91] Cristopher Moore and Stephan Mertens. *The Nature of Computation*. Oxford University Press, 2011.

[92] K. Mülling, J. Kober, and J. Peters. A biomimetic approach to robot table tennis. *Adaptive Behavior*, 19(5):359–376, 2011.

[93] R. Munos and Cs. Szepesvári. Finite time bounds for sampling based fitted value iteration. *Journal of Machine Learning Research*, 9:815–857, 2008.

[94] R. M. Murray, Z. Li, and S. Sastry. *A Mathematical Introduction to Robotic Manipulation*. CRC Press, Boca Raton, FL, 1994.

[95] Chikatoyo Nagata, Eri Sakamoto, Masato Suzuki, and Seiji Aoyagi. Path generation and collision avoidance of robot manipulator for unknown moving obstacle using real-time rapidly-exploring random trees (RRT) method. In *Service Robotics and Mechatronics*, pages 335–340. Springer, 2010.

[96] Andrew Y. Ng and Stuart J. Russell. Algorithms for inverse reinforcement learning. In *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, pages 663–670, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[97] Ann Now, Peter Vrancx, and Yann-Michal Hauwere. Game theory and multi-agent reinforcement learning. In Marco Wiering and Martijn Otterlo, editors, *Reinforcement Learning*, volume 12 of *Adaptation, Learning, and Optimization*, pages 441–470. Springer Berlin Heidelberg, 2012.

[98] P. Ogren, M. Egerstedt, and X. Hu. A control Lyapunov function approach to multi-agent coordination. *IEEE Transactions on Robotics and Automation*, pages 847–852, Oct 2002.

[99] I. Palunko, P. Cruz, and R. Fierro. Agile load transportation : Safe and efficient load manipulation with aerial robots. *IEEE Robotics Automation Magazine*, 19(3):69 –79, sept. 2012.

[100] I. Palunko, R. Fierro, and P. Cruz. Trajectory generation for swing-free maneuvers of a quadrotor with suspended payload: A dynamic programming approach. In *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, pages 2691 –2697, May 2012.

[101] Ivana Palunko, Aleksandra Faust, Patricio Cruz, Lydia Tapia, and Rafael Fierro. A reinforcement learning approach towards autonomous suspended load manipulation using aerial robots. In *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, pages 4881–4886, May 2013.

[102] Parasol. Parasol Lab, Department of Computer Science and Engineering, Texas A&M University. https://parasol.tamu.edu/, December 2012.

[103] Jung-Jun Park, Ji-Hun Kim, and Jae-Bok Song. Path planning for a robot manipulator based on probabilistic roadmap and reinforcement learning. In *International Journal of Control, Automation, and Systems*, pages 674–680, 2008.

[104] Theodore J. Perkins and Andrew G. Barto. Lyapunov design for safe reinforcement learning. *Journal of Machine Learning Research*, 3:803–832, 2002.

[105] Jan Peters and Stefan Schaal. Natural actor-critic. *Neurocomputing*, 71(79):1180 – 1190, 2008.

[106] P.E.I. Pounds, D.R. Bersak, and A.M. Dollar. Grasping from the air: Hovering capture and load stability. In *2011 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2491 –2498, may 2011.

[107] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. *SIGGRAPH Comput. Graph.*, 21(4):25–34, August 1987.

[108] R. Ritz, M.W. Muller, M. Hehn, and R. D'Andrea. Cooperative quadrocopter ball throwing and catching. In *Proc. IEEE/RSJ Intl Conf. on Intelligent Robots and Systems (IROS)*, pages 4972 –4978, oct. 2012.

[109] Samuel Rodríguez, Jyh-Ming Lien, and Nancy M. Amato. A framework for planning motion in environments with moving obstacles. In *Proc. IEEE Int. Conf. Intel. Rob. Syst. (IROS)*, pages 3309–3314, 2007.

[110] B. Rohrer. Biologically inspired feature creation for multi-sensory perception. In *BICA*, 2011.

[111] B. Rohrer. A developmental agent for learning features, environment models, and general robotics tasks. In *ICDL*, 2011.

[112] G. A. Rummery and M. Niranjan. On-line q-learning using connectionist systems, a lyapunov-based approach. Technical report, 1994.

[113] Angela P. Schoellig, Fabian L. Mueller, and Raffaello D'Andrea. Optimization-based iterative learning for precise quadrocopter trajectory tracking. *Autonomous Robots*, 33:103–127, 2012.

[114] J. Schultz and T. Murphey. Trajectory generation for underactuated control of a suspended mass. In *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, pages 123 –129, may 2012.

[115] Alexander A. Sherstov and Peter Stone. Improving action selection in MDP's via knowledge transfer. In *AAAI Conference on Artificial Intelligence*, pages 1024–1030, July 2005.

[116] Alexander C. Shkolnik and Russ Tedrake. Path planning in 1000+ dimensions using a task-space voronoi bias. In *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, pages 2061–2067. IEEE, may 2010.

[117] Koushil Sreenath, Nathan Michael, and Vijay Kumar. Trajectory generation and control of a quadrotor with a cable-suspended load – a differentially-flat hybrid system. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 4873–4880, 2013.

[118] G. Starr, J. Wood, and R. Lumia. Rapid transport of suspended payloads. In *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pages 1394 – 1399, april 2005.

[119] Mike Stilman. Global manipulation planing in robot joint space with task constraints. *IEEE/RAS Transactions on Robotics*, 26(3):576–584, 2010.

[120] R. Sutton and A. Barto. *A Reinforcement Learning: an Introduction*. MIT Press, MIT, 1998.

[121] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

[122] Csaba Szepesvári. *Algorithms for Reinforcement Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2010.

[123] Lydia Tapia, Shawna Thomas, and Nancy M. Amato. A motion planning approach to studying molecular motions. *Communications in Information and Systems*, 10(1):53–68, 2010.

[124] Camillo Taylor and Anthony Cowley. Parsing indoor scenes using rgb-d imagery. In *Proc. Robotics: Sci. Sys. (RSS)*, Sydney, Australia, July 2012.

[125] Matthew E. Taylor and Peter Stone. Behavior transfer for value-function-based reinforcement learning. In *The Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 53–59, New York, NY, July 2005. ACM Press.

[126] Matthew E. Taylor, Peter Stone, and Yaxin Liu. Transfer learning via intertask mappings for temporal difference learning. *Journal of Machine Learning Research*, 8:2125–2167, 2007.

[127] Kyriakos G. Vamvoudakis, Draguna Vrabie, and Frank L. Lewis. Online adaptive algorithm for optimal control with integral reinforcement learning. *International Journal of Robust and Nonlinear Control*, 2013.

[128] Thomas J. Walsh, Sergiu Goschin, and Michael L. Littman. Integrating sample-based planning and model-based reinforcement learning. In Maria Fox and David Poole, editors, *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*, pages 612–617. AAAI Press, 2010.

[129] C. Watkins. *Learning from Delayed Rewards*. PhD thesis, University of Cambridge, England, 1989.

[130] C.J.C.H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992.

[131] Spencer White, Tony Martinez, and George Rudolph. Automatic algorithm development using new reinforcement programming techniques. *Computational Intelligence*, 28(2):176–208, 2012.

[132] Marco Wiering and Martijn Otterlo, editors. *Reinforcement Learning*, volume 12 of *Adaptation, Learning, and Optimization*. Springer Berlin Heidelberg, 2012.

[133] Jan Willmann, Federico Augugliaro, Thomas Cadalbert, Raffaello D'Andrea, Fabio Gramazio, and Matthias Kohler. Aerial robotic construction towards a new field of architectural research. *International Journal of Architectural Computing*, 10(3):439 – 460, 2012.

[134] Steven A. Wilmarth, Nancy M. Amato, and Peter F. Stiller. Maprm: A probabilistic roadmap planner with sampling on the medial axis of the free space. In *In Proc. IEEE Int. Conf. Robot. Autom. (ICRA),*, pages 1024–1031, May 1999.

[135] Cheng Wu. *Novel function approximation techniques for large-scale reinforcement learning*. PhD thesis, Northeastern University, Apr 2010.

[136] Jung Yang and Shih Shen. Novel approach for adaptive tracking control of a 3-d overhead crane system. *Journal of Intelligent and Robotic Systems*, 62:59–80, 2011.

[137] Tansel Yucelen, Bong-Jun Yang, and Anthony J. Calise. Derivative-free decentralized adaptive control of large-scale interconnected uncertain systems. In *IEEE Conference on Decision and Control and European Control Conference (CDC-ECC)*, pages 1104–1109, 2011.

[138] D. Zameroski, G. Starr, J. Wood, and R. Lumia. Rapid swing-free transport of nonlinear payloads using dynamic programming. *ASME Journal of Dynamic Systems, Measurement, and Control*, 130(4), July 2008.

[139] Yajia Zhang, Jingru Luo, and Kris Hauser. Sampling-based motion planning with dynamic intermediate state objectives: Application to throwing. In *ICRA*, pages 2551–2556, 2012.