# REINFORCEMENT LEARNING CONTROL WITH APPROXIMATION OF TIME-DEPENDENT AGENT DYNAMICS

A Dissertation

by

KENTON CONRAD KIRKPATRICK

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Approved by:

| | |
|---|---|
| Chair of Committee, | John Valasek |
| Committee Members, | Raktim Bhattacharya |
| | Suman Chakravorty |
| | Thomas Ioerger |
| Department Head, | Rodney Bowersox |

May 2013

Major Subject: Aerospace Engineering

Copyright 2013 Kenton Conrad Kirkpatrick

ABSTRACT

Reinforcement Learning has received a lot of attention over the years for systems ranging from static game playing to dynamic system control. Using Reinforcement Learning for control of dynamical systems provides the benefit of learning a control policy without needing a model of the dynamics. This opens the possibility of controlling systems for which the dynamics are unknown, but Reinforcement Learning methods like $Q$-learning do not explicitly account for time. In dynamical systems, time-dependent characteristics can have a significant effect on the control of the system, so it is necessary to account for system time dynamics while not having to rely on a predetermined model for the system.

In this dissertation, algorithms are investigated for expanding the $Q$-learning algorithm to account for the learning of sampling rates and dynamics approximations. For determining a proper sampling rate, it is desired to find the largest sample time that still allows the learning agent to control the system to goal achievement. An algorithm called *Sampled-Data Q-learning* is introduced for determining both this sample time and the control policy associated with that sampling rate. Results show that the algorithm is capable of achieving a desired sampling rate that allows for system control while not sampling "as fast as possible".

Determining an approximation of an agent's dynamics can be beneficial for the control of hierarchical multiagent systems by allowing a high-level supervisor to use the dynamics approximations for task allocation decisions. To this end, algorithms are investigated for learning first- and second-order dynamics approximations. These algorithms are respectively called *First-Order Dynamics Learning* and *Second-Order Dynamics Learning*. The dynamics learning algorithms are evaluated on several

examples that show their capability to learn accurate approximations of state dynamics. All of these algorithms are then evaluated on hierarchical multiagent systems for determining task allocation. The results show that the algorithms successfully determine appropriated sample times and accurate dynamics approximations for the agents investigated.

# DEDICATION

This dissertation is dedicated to my loving family. Without their patience, understanding, and encouragement the completion of this research and writing of this dissertation would not have been possible.

# ACKNOWLEDGEMENTS

I would like to acknowledge and express gratitude to my committee for their help in the execution of this research. Dr. Thomas Ioerger, Dr. Suman Chakravorty, and Dr. Raktim Bhattacharya provided valuable knowledge and expert advice in the fields of machine learning, dynamics, and control. I am especially thankful for the help of my committee chair, Dr. John Valasek, for his exceptional guidance and instrumental help in both the determination of this thesis and the execution of the research. His continuing support made this dissertation possible.

I would also like to acknowledge the sponsors who provided funding for this research. This work was sponsored (in part) by the National Science Foundation Graduate Research Fellowship Program, and the Air Force Office of Scientific Research, USAF, under grant/contract number FA9550-08-1-0038. The technical monitor is Dr. Fariba Fahroo. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the National Science Foundation, Air Force Office of Scientific Research, or the U.S. Government.

TABLE OF CONTENTS

Page

ABSTRACT . . . ii

DEDICATION . . . iv

ACKNOWLEDGEMENTS . . . v

TABLE OF CONTENTS . . . vi

LIST OF FIGURES . . . ix

LIST OF TABLES . . . xiv

LIST OF ALGORITHMS . . . xvi

NOMENCLATURE . . . xvii

1. INTRODUCTION . . . 1

   1.1 Motivation . . . 2
   1.2 Scope and Contribution . . . 4

2. REINFORCEMENT LEARNING . . . 6

   2.1 $Q$-learning . . . 6
   2.2 $\varepsilon$-greedy . . . 8
   2.3 Function Approximation . . . 10
      2.3.1 Artificial Neural Networks . . . 11
      2.3.2 Genetic Algorithms . . . 13
      2.3.3 $k$-Nearest Neighbor . . . 14
   2.4 $Q$-learning Control Example . . . 15

3. SAMPLED-DATA $Q$-LEARNING . . . 22

   3.1 Sampled-Data $Q$-learning Algorithm . . . 22
   3.2 Markov Property . . . 24
   3.3 Reward Shaping . . . 27
   3.4 Sampled-Data $Q$-learning Examples . . . 34

3.4.1 Inverted Pendulum . . . . . . . . . . . . . . . . . . . . . . . . 35
3.4.2 Rotating Robot . . . . . . . . . . . . . . . . . . . . . . . . . 45

4. DYNAMICS APPROXIMATION LEARNING . . . . . . . . . . . . . . . . 56

4.1 First-Order Dynamics Learning . . . . . . . . . . . . . . . . . . . . 58
4.2 Second-Order Dynamics Learning . . . . . . . . . . . . . . . . . . . 65
4.3 Dynamics Learning Examples . . . . . . . . . . . . . . . . . . . . . 75
4.3.1 First-Order Examples . . . . . . . . . . . . . . . . . . . . . . 78
4.3.2 Second-Order Examples . . . . . . . . . . . . . . . . . . . . . 88
4.3.3 Multiple States . . . . . . . . . . . . . . . . . . . . . . . . 105
4.4 Sample Time Ranges . . . . . . . . . . . . . . . . . . . . . . . . . 110

5. MULTIAGENT SYSTEMS . . . . . . . . . . . . . . . . . . . . . . . . . 116

5.1 Learning in Multiagent Systems . . . . . . . . . . . . . . . . . . . 116
5.1.1 Single-Level Multiagent Learning . . . . . . . . . . . . . . . . 116
5.1.2 Hierarchical Multiagent Learning . . . . . . . . . . . . . . . . 120
5.2 Homogeneous Agents Examples . . . . . . . . . . . . . . . . . . . . 122
5.2.1 Equal Number of Agents and Goals . . . . . . . . . . . . . . . 124
5.2.2 Fewer Goals than Agents . . . . . . . . . . . . . . . . . . . . 133
5.3 Heterogeneous Agents Examples . . . . . . . . . . . . . . . . . . . . 140
5.3.1 Equal Number of Agents and Goals . . . . . . . . . . . . . . . 143
5.3.2 Fewer Goals than Agents . . . . . . . . . . . . . . . . . . . . 155

6. CONCLUSIONS AND RECOMMENDATIONS . . . . . . . . . . . . . . . . 162

6.1 Sampled-Data $Q$-learning . . . . . . . . . . . . . . . . . . . . . . 162
6.2 Dynamics Learning . . . . . . . . . . . . . . . . . . . . . . . . . . 163
6.3 Multiagent Systems . . . . . . . . . . . . . . . . . . . . . . . . . 163
6.4 Recommendations . . . . . . . . . . . . . . . . . . . . . . . . . . 164

REFERENCES . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 167

APPENDIX A. DERIVATION OF INVERTED PENDULUM EQUATIONS OF MOTION . . . . . . . . . . . . . . . . . . . . . . . . . . . . 174

APPENDIX B. PARTIAL FRACTION EXPANSION SOLUTIONS . . . . . 181

B.1 No Damping . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 181

B.2 Underdamped . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 183
B.3 Critically Damped . . . . . . . . . . . . . . . . . . . . . . . . . . . 186
B.4 Overdamped . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 188

LIST OF FIGURES


FIGURE Page

2.1 Perceptron . . . 11

2.2 $Q$-learning Simulation of Robot - Goal = [0,0] . . . 18

2.3 Time History of Robot - Goal = [0,0] . . . 19

2.4 Command History of Robot - Goal = [0,0] . . . 19

2.5 $Q$-learning Simulation of Robot - Goal = [3,6] . . . 20

2.6 Time History of Robot - Goal = [3,6] . . . 20

2.7 Command History of Robot - Goal = [3,6] . . . 21

3.1 Sampled-Data $Q$-learning Diagram . . . 24

3.2 Inverted Pendulum . . . 35

3.3 Inverted Pendulum State Time History: Constant Reward . . . 40

3.4 Inverted Pendulum Force Time History: Constant Reward . . . 41

3.5 Inverted Pendulum Phase Diagram: Constant Reward . . . 41

3.6 Inverted Pendulum State Time History: Quadratic Reward . . . 43

3.7 Inverted Pendulum Force Time History: Quadratic Reward . . . 44

3.8 Inverted Pendulum Phase Diagram: Quadratic Reward . . . 45

3.9 Robot Simulation - Quadrant 1 . . . 48

3.10 State History of Robot - Quadrant 1 . . . 49

3.11 Command History of Robot - Quadrant 1 . . . 49

3.12 Robot Simulation - Quadrant 2 . . . 50

3.13 State History of Robot - Quadrant 2 . . . 50

3.14 Command History of Robot - Quadrant 2 . . . 51

3.15 Robot Simulation - Quadrant 3 . . . 51

3.16 State History of Robot - Quadrant 3 . . . 52

3.17 Command History of Robot - Quadrant 3 . . . . . . . . . . . . . . . . . 52
3.18 Robot Simulation - Quadrant 4 . . . . . . . . . . . . . . . . . . . . . . 53
3.19 State History of Robot - Quadrant 4 . . . . . . . . . . . . . . . . . . . 53
3.20 Command History of Robot - Quadrant 4 . . . . . . . . . . . . . . . . . 54
4.1 FODL Diagram . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
4.2 Complex Plane - $\zeta = 0$ . . . . . . . . . . . . . . . . . . . . . . . . . 69
4.3 Complex Plane - $0 < \zeta < 1$ . . . . . . . . . . . . . . . . . . . . . . . 69
4.4 Complex Plane - $\zeta = 1$ . . . . . . . . . . . . . . . . . . . . . . . . . 70
4.5 Complex Plane - $\zeta > 1$ . . . . . . . . . . . . . . . . . . . . . . . . . 70
4.6 SODL Diagram . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
4.7 Multiagent System . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
4.8 High-Level Agent Commands . . . . . . . . . . . . . . . . . . . . . . 76
4.9 High-Level Agent Decision Making . . . . . . . . . . . . . . . . . . . 77
4.10 $1^{st}$ Order Value Function for $\tau = 0.2$: 0 Episodes . . . . . . . . . . . . 79
4.11 $1^{st}$ Order Value Function for $\tau = 0.2$: 200 Episodes . . . . . . . . . . . 79
4.12 $1^{st}$ Order Value Function for $\tau = 0.2$: 1,000 Episodes . . . . . . . . . . 80
4.13 $1^{st}$ Order Value Function for $\tau = 0.2$: 5,000 Episodes . . . . . . . . . . 80
4.14 $1^{st}$ Order Value Function for $\tau = 0.2$: 10,000 Episodes . . . . . . . . . 81
4.15 $1^{st}$ Order Value Function History for $\tau = 0.2$ . . . . . . . . . . . . . . 81
4.16 $1^{st}$ Order Value Function for $\tau = 1$: 0 Episodes . . . . . . . . . . . . . 83
4.17 $1^{st}$ Order Value Function for $\tau = 1$: 200 Episodes . . . . . . . . . . . . 84
4.18 $1^{st}$ Order Value Function for $\tau = 1$: 1,000 Episodes . . . . . . . . . . . 84
4.19 $1^{st}$ Order Value Function for $\tau = 1$: 5,000 Episodes . . . . . . . . . . . 85
4.20 $1^{st}$ Order Value Function for $\tau = 1$: 10,000 Episodes . . . . . . . . . . 85
4.21 $1^{st}$ Order Value Function History for $\tau = 1$ . . . . . . . . . . . . . . . 86
4.22 $2^{nd}$ Order Value Function for $(\omega_n, \zeta) = (6, 0.8)$: 0 Episodes . . . . . . 90

4.23 $2^{nd}$ Order Value Function for $(\omega_n, \zeta) = (6, 0.8)$: 200 Episodes . . . . . 90
4.24 $2^{nd}$ Order Value Function for $(\omega_n, \zeta) = (6, 0.8)$: 1,000 Episodes . . . . 91
4.25 $2^{nd}$ Order Value Function for $(\omega_n, \zeta) = (6, 0.8)$: 5,000 Episodes . . . . 91
4.26 $2^{nd}$ Order Value Function for $(\omega_n, \zeta) = (6, 0.8)$: 10,000 Episodes . . . 92
4.27 Average $2^{nd}$ Order Values for $(\omega_n, \zeta) = (6, 0.8)$: 0 Episodes . . . . . . 93
4.28 Average $2^{nd}$ Order Values for $(\omega_n, \zeta) = (6, 0.8)$: 200 Episodes . . . . . 94
4.29 Average $2^{nd}$ Order Values for $(\omega_n, \zeta) = (6, 0.8)$: 1,000 Episodes . . . . 94
4.30 Average $2^{nd}$ Order Values for $(\omega_n, \zeta) = (6, 0.8)$: 5,000 Episodes . . . . 95
4.31 Average $2^{nd}$ Order Values for $(\omega_n, \zeta) = (6, 0.8)$: 10,000 Episodes . . . 95
4.32 Average $\omega_n$ Value History for $(\omega_n, \zeta) = (6, 0.8)$ . . . . . . . . . . . . . 96
4.33 Average $\zeta$ Value History for $(\omega_n, \zeta) = (6, 0.8)$ . . . . . . . . . . . . . . 96
4.34 $2^{nd}$ Order Value Function for $(\omega_n, \zeta) = (10, 1.2)$: 0 Episodes . . . . . . 98
4.35 $2^{nd}$ Order Value Function for $(\omega_n, \zeta) = (10, 1.2)$: 200 Episodes . . . . 99
4.36 $2^{nd}$ Order Value Function for $(\omega_n, \zeta) = (10, 1.2)$: 1,000 Episodes . . . 99
4.37 $2^{nd}$ Order Value Function for $(\omega_n, \zeta) = (10, 1.2)$: 5,000 Episodes . . . 100
4.38 $2^{nd}$ Order Value Function for $(\omega_n, \zeta) = (10, 1.2)$: 10,000 Episodes . . . 100
4.39 Average $2^{nd}$ Order Values for $(\omega_n, \zeta) = (10, 1.2)$: 0 Episodes . . . . . 101
4.40 Average $2^{nd}$ Order Values for $(\omega_n, \zeta) = (10, 1.2)$: 200 Episodes . . . . 102
4.41 Average $2^{nd}$ Order Values for $(\omega_n, \zeta) = (10, 1.2)$: 1,000 Episodes . . . 102
4.42 Average $2^{nd}$ Order Values for $(\omega_n, \zeta) = (10, 1.2)$: 5,000 Episodes . . . 103
4.43 Average $2^{nd}$ Order Values for $(\omega_n, \zeta) = (10, 1.2)$: 10,000 Episodes . . . 103
4.44 Average $\omega_n$ Value History for $(\omega_n, \zeta) = (10, 1.2)$ . . . . . . . . . . . . 104
4.45 Average $\zeta$ Value History for $(\omega_n, \zeta) = (10, 1.2)$ . . . . . . . . . . . . . 104
4.46 $V_1$ for Robot Speed Plot . . . . . . . . . . . . . . . . . . . . . . . . 108
4.47 $V_2$ for Robot Heading with Velocity Changes Plot . . . . . . . . . . . 109
5.1 Hierarchical Multiagent System Diagram . . . . . . . . . . . . . . . . 122

5.2 Hierarchical Multiagent System . . . . . . . . . . . . . . . . . . . . . . 123
5.3 Homogeneous Multiagent Robot Paths - Sim 1 . . . . . . . . . . . . . . 126
5.4 Homogeneous Multiagent Robot States - Sim 1 . . . . . . . . . . . . . . 127
5.5 Homogeneous Multiagent Robot Command - Sim 1 . . . . . . . . . . . 127
5.6 Homogeneous Multiagent Robot Speed - Sim 1 . . . . . . . . . . . . . . 128
5.7 Homogeneous Multiagent Robot Paths - Sim 2 . . . . . . . . . . . . . . 130
5.8 Homogeneous Multiagent Robot States - Sim 2 . . . . . . . . . . . . . . 131
5.9 Homogeneous Multiagent Robot Command - Sim 2 . . . . . . . . . . . 131
5.10 Homogeneous Multiagent Robot Speed - Sim 2 . . . . . . . . . . . . . . 132
5.11 Homogeneous Multiagent Robot Paths - Sim 3 . . . . . . . . . . . . . . 134
5.12 Homogeneous Multiagent Robot States - Sim 3 . . . . . . . . . . . . . . 135
5.13 Homogeneous Multiagent Robot Command - Sim 3 . . . . . . . . . . . 135
5.14 Homogeneous Multiagent Robot Speed - Sim 3 . . . . . . . . . . . . . . 136
5.15 Homogeneous Multiagent Robot Paths - Sim 4 . . . . . . . . . . . . . . 138
5.16 Homogeneous Multiagent Robot States - Sim 4 . . . . . . . . . . . . . . 138
5.17 Homogeneous Multiagent Robot Command - Sim 4 . . . . . . . . . . . 139
5.18 Homogeneous Multiagent Robot Speed - Sim 4 . . . . . . . . . . . . . . 139
5.19 Heterogeneous Multiagent Robot Paths - Sim 1 . . . . . . . . . . . . . 145
5.20 Heterogeneous Multiagent Robot States - Sim 1 . . . . . . . . . . . . . 145
5.21 Heterogeneous Multiagent Robot Command - Sim 1 . . . . . . . . . . . 146
5.22 Heterogeneous Multiagent Robot Speed - Sim 1 . . . . . . . . . . . . . 146
5.23 Heterogeneous Robot Paths - Sim 1 (More Learning) . . . . . . . . . . 148
5.24 Heterogeneous Robot States - Sim 1 (More Learning) . . . . . . . . . . 149
5.25 Heterogeneous Robot Command - Sim 1 (More Learning) . . . . . . . . 149
5.26 Heterogeneous Robot Speed - Sim 1 (More Learning) . . . . . . . . . . 150
5.27 Heterogeneous Multiagent Robot Paths - Sim 2 . . . . . . . . . . . . . 152

5.28 Heterogeneous Multiagent Robot States - Sim 2 . . . . . . . . . . . . . 153
5.29 Heterogeneous Multiagent Robot Command - Sim 2 . . . . . . . . . . . 153
5.30 Heterogeneous Multiagent Robot Speed - Sim 2 . . . . . . . . . . . . . 154
5.31 Heterogeneous Multiagent Robot Paths - Sim 3 . . . . . . . . . . . . . 156
5.32 Heterogeneous Multiagent Robot States - Sim 3 . . . . . . . . . . . . . 156
5.33 Heterogeneous Multiagent Robot Command - Sim 3 . . . . . . . . . . . 157
5.34 Heterogeneous Multiagent Robot Speed - Sim 3 . . . . . . . . . . . . . 157
5.35 Heterogeneous Multiagent Robot Paths - Sim 4 . . . . . . . . . . . . . 159
5.36 Heterogeneous Multiagent Robot States - Sim 4 . . . . . . . . . . . . . 159
5.37 Heterogeneous Multiagent Robot Command - Sim 4 . . . . . . . . . . . 160
5.38 Heterogeneous Multiagent Robot Speed - Sim 4 . . . . . . . . . . . . . 160
A.1 Inverted Pendulum on a Cart . . . . . . . . . . . . . . . . . . . . . . . 174

LIST OF TABLES


TABLE Page

3.1 Reward Function Success Rate . . . . . . . . . . . . . . . . . . . . . . . 34

3.2 SDQL Pendulum Result - Constant Reward . . . . . . . . . . . . . . . 39

3.3 SDQL Pendulum Result - Quadratic Reward . . . . . . . . . . . . . . . 42

3.4 SDQL Robot Result - Quadratic Reward . . . . . . . . . . . . . . . . . 47

4.1 FODL Robot Value Function: $\tau = 0.2$ . . . . . . . . . . . . . . . . . . . 82

4.2 FODL Robot Value Function: $\tau = 1$ . . . . . . . . . . . . . . . . . . . . 87

4.3 $V_2$ after 10,000 Episodes: $(\omega_n, \zeta) = (6, 0.8)$ . . . . . . . . . . . . . . . . 97

4.4 $V_2$ after 10,000 Episodes: $(\omega_n, \zeta) = (10, 1.2)$ . . . . . . . . . . . . . . . 105

4.5 $V_1$ for Robot Speed: $\tau = 1$ . . . . . . . . . . . . . . . . . . . . . . . . 107

4.6 $V_2$ for Robot Heading with Velocity Changes: $(\omega_{n,\psi}, \zeta_\psi) = (6, 0.8)$ . . 108

4.7 FODL with Sampling Ranges (Short) . . . . . . . . . . . . . . . . . . . 111

4.8 FODL with Sampling Ranges (Medium) . . . . . . . . . . . . . . . . . 112

4.9 FODL with Sampling Ranges (Long) . . . . . . . . . . . . . . . . . . . 112

4.10 FODL with Sampling Ranges (Tiny) . . . . . . . . . . . . . . . . . . . 113

4.11 SODL with Sampling Ranges (Short) . . . . . . . . . . . . . . . . . . . 114

4.12 SODL with Sampling Ranges (Long) . . . . . . . . . . . . . . . . . . . 115

5.1 Multiagent Goals . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 125

5.2 Homogeneous Multiagent System ICs - Sim 1 . . . . . . . . . . . . . . 125

5.3 Homogeneous Multiagent Goal Assignment - Sim 1 . . . . . . . . . . . 126

5.4 Homogeneous Multiagent System ICs - Sim 2 . . . . . . . . . . . . . . 129

5.5 Homogeneous Multiagent Goal Assignment - Sim 2 . . . . . . . . . . . 129

5.6 Homogeneous Multiagent System ICs - Sim 3 . . . . . . . . . . . . . . 133

5.7 Homogeneous Multiagent Goal Assignment - Sim 3 . . . . . . . . . . . 134
5.8 Homogeneous Multiagent System ICs - Sim 4 . . . . . . . . . . . . . . 137
5.9 Homogeneous Multiagent Goal Assignment - Sim 4 . . . . . . . . . . . 137
5.10 Agent 1 Learned Dynamics . . . . . . . . . . . . . . . . . . . . . . . . 143
5.11 Agent 2 Learned Dynamics . . . . . . . . . . . . . . . . . . . . . . . . 143
5.12 Agent 3 Learned Dynamics . . . . . . . . . . . . . . . . . . . . . . . . 143
5.13 Heterogeneous Multiagent System ICs - Sim 1 . . . . . . . . . . . . . . 144
5.14 Heterogeneous Multiagent Goal Assignment - Sim 1 . . . . . . . . . . . 144
5.15 Heterogeneous Multiagent System ICs - Sim 2 . . . . . . . . . . . . . . 151
5.16 Heterogeneous Multiagent Goal Assignment - Sim 2 . . . . . . . . . . . 151
5.17 Heterogeneous Multiagent System ICs - Sim 3 . . . . . . . . . . . . . . 155
5.18 Heterogeneous Multiagent Goal Assignment - Sim 3 . . . . . . . . . . . 155
5.19 Heterogeneous Multiagent System ICs - Sim 4 . . . . . . . . . . . . . . 158
5.20 Heterogeneous Multiagent Goal Assignment - Sim 4 . . . . . . . . . . . 158

LIST OF ALGORITHMS


ALGORITHM — Page

2.1 $Q$-learning . . . 8

2.2 $\varepsilon$-greedy . . . 10

2.3 $k$-Nearest Neighbor . . . 15

3.1 Sampled-Data $Q$-learning (SDQL) . . . 25

4.1 First-Order Dynamics Learning (FODL) . . . 62

4.2 FODL with SDQL . . . 64

4.3 Second-Order Dynamics Learning (SODL) . . . 73

4.4 SODL with SDQL . . . 74

NOMENCLATURE

| | |
|---|---|
| $\alpha$ | Step-Size Parameter |
| $\beta$ | Random Variable with Uniform Distribution |
| $\gamma$ | Discount Rate |
| $\Delta$ | Change |
| $\delta$ | Temporal-Difference |
| $\varepsilon$ | Exploration Probability |
| $\zeta$ | Damping Ratio |
| $\zeta_\psi$ | Heading Angle Damping Ratio |
| $\theta$ | Angle of Pendulum |
| $\lambda$ | Laplace Parameter |
| $\pi$ | Policy |
| $\tau$ | Time Constant |
| $\tau_V$ | Speed Time Constant |
| $\chi$ | Instance |
| $\psi$ | Heading Angle |
| $\psi_c$ | Commanded Heading Angle |
| $\omega_n$ | Natural Frequency |
| $\omega_{n,\psi}$ | Heading Angle Natural Frequency |
| $\nabla$ | Gradient |
| $\vert$ | Conditional Probability |
| , | Logical And |
| $A$ | Action-Space |
| $A_i$ | Agent $i$, where $i$ = 1,2,3... |

| | |
|---|---|
| $A_s$ | Supervisor Agent |
| $a$ | RL Action |
| $a'$ | Next RL Action |
| DP | Dynamic Programming |
| $C_i$ | Partial Fraction Expansion Constant $i$, where $i$ = 1,2,3... |
| $c$ | Index |
| $Cov[\cdot]$ | Covariance |
| $d$ | Euclidean Distance |
| $E[\cdot]$ | Expected Value (Mean) |
| $F$ | Force |
| $\underline{F}$ | Force Vector |
| $f(T)$ | Sample Time Function |
| FODL | First-Order Dynamics Learning |
| $g$ | Gravity |
| $G_i$ | Goal $i$, where $i$ = 1,2,3... |
| $h$ | Integration Time Step |
| $I$ | Network Inputs |
| $i$ | Index |
| $j$ | Index |
| j | Imaginary Number |
| $K$ | Kinetic Energy |
| $k$ | Number of Nearest Neighbors |
| $k_i$ | 4th-Order Runge-Kutta Increment Parameters, where $i$ = 1,2,3,4 |
| $L$ | Length |
| $L_f$ | Lagrangian Function |

| | |
|---|---|
| $\mathcal{L}\{\cdot\}$ | Laplace Transform |
| $M$ | Cart Mass |
| $m$ | Pendulum Mass |
| MDP | Markov Decision Process |
| $M_p$ | % Overshoot |
| $n$ | Dimension of State-Space |
| ODE | Ordinary Differential Equation |
| $\mathbf{P}$ | State-Transition Probability Matrix |
| $P$ | Probability |
| $Q$ | Action-Value Function |
| $Q_i$ | Generalized Forces |
| $\hat{Q}$ | Approximate Action-Value Function |
| $q_i$ | Generalized Coordinates, where $i$ = 1,2,3,... |
| $R$ | Expected Return |
| $r$ | Reward |
| $\underline{r}$ | Displacement Vector |
| $r_c$ | Constant Reward |
| RL | Reinforcement Learning |
| $s$ | RL State |
| $s'$ | Next RL State |
| $\overline{s}$ | $k$-Nearest RL States |
| $\tilde{s}$ | Augmented RL State Vector |
| $s_c$ | Commanded State |
| SDQL | Sampled-Data Q-learning |
| SMDP | Semi-Markov Decision Process |

| | |
|---|---|
| SODL | Second-Order Dynamics Learning |
| $steps^{(j)}$ | Number of Steps in Episode $j$ |
| $T$ | Sample Time |
| $t$ | Current Timestep |
| $t_f$ | Final Time |
| $t_p$ | Peak Time |
| $t_r$ | Rise Time |
| $t_s$ | Settling Time |
| TD | Temporal-Difference |
| $U$ | Potential Energy |
| $V$ | Speed |
| $V_c$ | Commanded Speed |
| $V_{max}$ | Maximum Speed |
| $V_T$ | Sample Time Value Function |
| $V_1$ | First-Order Dynamics Value Function |
| $V_2$ | Second-Order Dynamics Value Function |
| $v$ | Predicted Value of Next State-Action Pair |
| $v_c$ | Velocity of Cart |
| $v_m$ | Velocity of Mass |
| $Var[\cdot]$ | Variance |
| $visits(\cdot)$ | Number of Times to Visit Parameter $\cdot$ |
| $\dot{W}$ | Work Rate |
| $w_i$ | Weight $i$, where $i$ = 1,2,3... |
| $\underline{w}$ | Vector of Weights |
| $x$ | State |

| | |
|---|---|
| $\dot{x}$ | State Velocity |
| $\ddot{x}$ | State Acceleration |
| $\ddot{x}_{est}$ | Estimated State Acceleration |
| $x_c$ | Command in $x$-direction |
| $X$ | Weighted Input Sum |
| $y_c$ | Command in $y$-direction |
| $z_1$ | Overdamped Solution Constant |
| $z_2$ | Overdamped Solution Constant |

# 1. INTRODUCTION

Control of systems with unknown agent dynamics or in unknown environments has received much attention over recent years. Solving the problem of determining how to plan paths for controlling agents in a previously unknown environment has lead to the investigation of such methods as probability road maps and rapidly-expanding random trees [5, 6]. Alternatively, learning to control agents whose dynamics are previously unknown has led to a variety of methods branched by model-based or model-free methods. Model-based control methods require determining a model of the system, which can be user-time consuming and complex. Methods of system identification [48], reduced model control [3], uncertainty analysis [8], and even artificial neural network-based modeling [38, 21] have been approached for resolving unknown models or complex models.

In recent years, Reinforcement Learning (RL) has been an extensively investigated area of research in the field of machine learning. Machine learning covers a wide variety of autonomous classification and control methods that have been utilized for a number of tasks ranging from game playing [37] to protein modeling [16] to aircraft control and navigation [52]. In particular, RL has been a popular tool for solving problems such as dynamical system control, gain scheduling, maze navigation, and game playing. There has been wide success in many of the applications, with the most commonly explored being game playing scenarios such as TD-Gammon [46]. However, researchers have often encountered problems when casting the control of dynamical systems as an RL problem. The state-to-action mapping provided by RL techniques makes the idea of using them for control problems attractive, and this is especially the case with $Q$-learning due to its proven convergence to optimal-

ity [20, 22, 23, 42, 24]. RL methods like $Q$-learning are appealing because they can achieve this mapping experimentally without the need of a model [50]. Although this is indeed the case, implementing these in real-world problems with time dynamics has proven to be very difficult [45].

## 1.1 Motivation

In this dissertation, RL is studied for its use in dynamical system control due to its model-free property for learning a behavioral mapping of states to control. While more robust model-based methods of control may work better in many cases, determining models for systems for which the dynamics are unknown can be difficult and time consuming. Studying the problems associated with implementing RL methods in dynamical systems reveals that often the failures are not caused by the basic approach of algorithms like $Q$-learning. Typically, the problem is either a failure in properly representing the problem, inaccurate function approximation, or both. When choosing to implement the popular Watkins' $Q$-learning in a dynamical system scenario, it is necessary to realize that the algorithm does not explicitly account for time [50]. Time dependency is often either not accounted for during the learning process or overlooked completely, but accounting for time in the selection of actions (or control) is important. For instance, when handling sampled-data systems, small changes in sample time can cause drastic changes to the stability of the control policy. Time is often abstracted out of the system, with dynamical system control being treated like a static game of chess. Learning control policies in dynamical systems will fail unless the dynamics are accounted for somehow in the learning process [2],[1].

One research area that has recently received much attention has been the control of cooperative multiagent systems through the use of $Q$-learning-based algorithms.

Learning to control multiple agents for the purpose of cooperatively achieving a specified goal is an appealing research topic with high complexity. Some research in this area has involved comparing the effects of using $Q$-learning-based methods to determine joint action selection between different agents to the learning of agent actions independently [7]. Other research has investigated stochastic game extensions to this and treated the system as non-cooperative by having agents consider only themselves with no knowledge of the existence of other agents [17]. Systems of agents that need to coordinate their actions without knowledge of each other's actions is an important area of research for the general application to systems of agents that do not have the ability to communicate with one another [29], [19]. Even so, other research has been conducted involving the improvement of $Q$-learning approaches for determining joint actions through the use of Bayesian inference to estimate strategies [47].

In most of the research discussed above, multiple agents are simulated in games that have no dependence on time. Time-dependent agent dynamics cause a fundamental change to the system, and accounting for time dependencies in multiagent systems has received little attention. This may be due to the fact that it is difficult to learn control policies for a single *time-dependent* agent using RL approaches. To address this, a topic that needs to be investigated is the learning of the longest sample time for a sampled-data system that can achieve a goal. Controlling real continuous-time systems generally requires computer-based control, so sampling of the continuous system is necessary. Problems arise in attempting to use a policy derived in simulation on an actual hardware experiment without considering the sample time used. In this case, if a model of the dynamics exists it is trivial to determine the best sample time by classical methods. However, here we consider the case where a model is not available and RL is being utilized for its model-free property. Thus there is a need to determine the optimal sample time without the use of a model.

Another area of investigation that is needed for the learning of multiagent system control policies is approximation of the dynamics. The learning process does not explicitly account for agent time dynamics, so that information is essentially abstracted out of the learning. Since the main benefit of RL approaches like $Q$-learning is to learn a control policy without the need for a model, the benefits of determining a model have been overlooked. Having some knowledge of the time dynamics is needed for determining the decisions to be made by the agents, and this is especially true in heterogeneous multiagent systems. A full model may not be necessary, but some approximation of the individual agent dynamics can be very beneficial for determining global behavior rules in time-to-goal command and control scenarios.

## 1.2 Scope and Contribution

In this dissertation, RL-based control approaches are extended to systems with unknown time dynamics. The scope is limited to the cases of simulated examples where the simulations have time dynamics but the RL agent learning to control the system has no access to that information. The systems considered will have either first- or second-order linear dynamics, and all systems considered will be sampled-data systems. The dynamics of all agents involved are deterministic, so no noise or uncertainty is considered. There is no available model for the system that can be used for classic control methods or even for classic determination of an appropriate sample time.

One unique contribution is an RL-based algorithm that is capable of determining the optimal sample time while simultaneously learning the control policy. This requires maximizing a reward that is a function of the sample time itself. A second major contribution of this research is the determination of an RL-based algorithm capable of learning an approximation of agent dynamics. This involves learning

state-to-state time transitions, and the approximations learned include a time constant for first-order systems and the proper combination of natural frequency and damping ratio for second-order systems. The information learned is then used in the scheme of a heterogeneous multiagent system. The full contribution of this research is a methodology for learning optimal sample times for agents, approximations of agents' time dynamics, and individual agent control policies. This collective result allows for the control of heterogeneous multiagent systems by means of hierarchical commands provided by a high-level automated supervisor agent with all of the knowledge learned by these algorithms.

# 2. REINFORCEMENT LEARNING*

There are multiple classes of algorithms that fall within the definition of Reinforcement Learning. Currently, the RL algorithms that are most used in research are Temporal-Difference (TD) methods. TD methods are actually a conceptual combination from two other classes of algorithms known as Dynamic Programming (DP) and Monte Carlo [42]. Like Monte Carlo, TD methods use experience through interaction with the system to update the quality of the value function without the need of a model. Like DP, TD methods do not have to wait until the end to improve the value function, but rather update it along the way. This improves convergence time and also makes TD methods usable in online learning. In this chapter, the TD algorithm known as $Q$-learning will be discussed, along with the limitations that led to the development of extended $Q$-learning-based algorithms for the use of learning in dynamical sampled-data systems [25].*

## 2.1 $Q$-learning

Of the various formulations of RL algorithms, Watkins' $Q$-learning has been the most accepted and utilized algorithm for its proven convergence to the optimal action-value function [50]. $Q$-learning is a TD method that learns the optimal action-value function in an off-policy manner [42]. This means that the policy used during a learning episode is not necessarily the same as the one that is updated at each timestep. The $Q$-learning algorithm is based upon an action-value update rule that uses a greedy policy to determine a predicted value for the state-action pair at the

*Part of the material reported in this chapter is reproduced with permission of John Wiley & Sons, Inc. from *Reinforcement Learning and Approximate Dynamic Programming for Feedback Control*, eds. Frank L. Lewis and Derong Liu, 2012, John Wiley & Sons, Inc., Hoboken, NJ. Copyright 2013 by The Institute of Electrical and Electronics Engineers, Inc.

next (future) timestep. The actual action selection may not be done using a greedy policy, and in fact it is typically better for optimality to include some degree of exploration in the policy. The rule used for updating the action-value function is as follows.

$$Q(s,a) \leftarrow Q(s,a) + \alpha\delta \tag{2.1}$$

In Equation 2.1, $Q$ is the action-value function, $s$ is the state at the current timestep, $a$ is the action selected for the current timestep using the agent's policy (e.g., $\varepsilon$-greedy), $\alpha$ is the step-size parameter, and $\delta$ is the temporal-difference. The value of $\alpha$ is always in the interval (0,1), and can either be held constant or varied by some user-defined function. In some cases, $\alpha$ is designed to decrease within an episode according to how often the agent revisits the same state, essentially punishing the agent for repeating itself unnecessarily. The definition of $\delta$ is the main term that distinguishes $Q$-learning from other TD algorithms, and it can be computed using Equation 2.2.

$$\delta = r + \gamma \max_{a'} Q(s', a') - Q(s,a) \tag{2.2}$$

In Equation 2.2, $r$ is the reward received from the system, $s'$ is the future state (due to taking action $a$), $a'$ is the action that would be taken using a greedy policy when in state $s'$, and $\gamma$ is the weight for the future value. The selection of $\gamma$ affects convergence time, and it is always within the range (0,1). The value of $\gamma$ can either be kept constant throughout learning, or it can be chosen to vary episodically so that later learning episodes value the future prediction differently than early episodes. Combining Equations 2.1 and 2.2 leads to the full $Q$-learning update equation, shown in Equation 2.3.

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)] \tag{2.3}$$

This update rule is the backbone of the famous Watkins' $Q$-learning algorithm. It is an off-policy TD algorithm with a user-determined policy for selecting actions at each timestep, but uses a fully-greedy policy with the action-value function when updating the action-value function. Rather than utilizing past information to perform this update, this algorithm uses the predicted future state-action pair chosen greedily. The Watkins' $Q$-learning algorithm is displayed in Algorithm 2.1.

**Algorithm 2.1** $Q$-learning

- Initialize $Q(s,a)$ arbitrarily
- Repeat for each episode:
  - Initialize $s$
  - Repeat for each timestep:
    - Choose $a$ from $s$ using policy derived from $Q(s,a)$ (e.g., $\varepsilon$-Greedy)
    - Take action $a$, observe $r$, $s'$
    - $Q(s,a) \leftarrow Q(s,a) + \alpha$ [ $r + \gamma$ $\max_{a'}$ $Q(s',a')$ - $Q(s,a)$ ]
    - $s \leftarrow s'$
  - Until $s$ is terminal

## 2.2 $\varepsilon$-greedy

When determining a policy for selecting actions during the learning process, there are a number of possibilities from which to choose. The most commonly used policies for the $Q$-learning process are exploration [12], greedy [10], and $\varepsilon$-greedy methods [15]. An exploration policy simply means that at any given state and any given timestep, an action is selected completely at random from the possible action-space.

This is demonstrated by Equation 2.4, where $A$ represents the full action-space.

$$a \sim U(0, 1) \in A \tag{2.4}$$

Conversely, a fully greedy method requires that the action-value function be exploited for choosing an action at any given timestep. This is demonstrated by Equation 2.5.

$$a = \max_{a} Q(s, a) \tag{2.5}$$

Both of these extremes have limitations that can inhibit the learning process due to either slowing the convergence process or converging to paths that may not be optimal. It is often necessary to find a middle ground, and the $\varepsilon$-greedy method accomplishes just that [42].

An $\varepsilon$-greedy policy uses a user-defined probability, $\varepsilon$, that determines whether to choose actions randomly or according to the action-value function each time a new action must be taken. This speeds up the convergence time by reinforcing paths that have already been designated either good or bad while still allowing for new paths to be explored for optimality [51]. In the early episodes the agent is required to explore due to lack of knowledge, regardless of the value assigned to $\varepsilon$. This method can be implemented during the action selection portion of the learning process according to Algorithm 2.2.

The chosen value of $\varepsilon$ can have a drastic effect on the number of episodes required to achieve convergence, and $\varepsilon$ can either be chosen as a constant value or a variable. It is usually of benefit to choose a high probability of exploration in early episodes, and then anneal the value of $\varepsilon$ in later episodes. Decreasing $\varepsilon$ as more episodes are completed leads to encouraging exploration in early episodes and exploitation in

**Algorithm 2.2** $\varepsilon$-greedy

- Choose $\varepsilon \in [0,1]$
- Repeat for each action selection:
  - Generate random value $\beta \sim U(0,1)$
  - If $\beta \geq 1$ - $\varepsilon$
    - $a \leftarrow$ random
  - If $\beta < 1$ - $\varepsilon$
    - $a \leftarrow \max_a Q(s,a)$

later episodes. Maintaining a nonzero probability of exploration during late learning (e.g., $\varepsilon = 0.05$) typically helps the learning process by allowing the agent to discover a better path that may not have been discovered yet.

## 2.3 Function Approximation

When applying RL methods like $Q$-learning, a key issue that causes problems with the final learned function is *generalization* [42]. The learning of an action-value function, $Q(s,a)$ results in a policy that requires a table look-up in order to be used. Tables are by nature discrete, so encountering every possible combination of state-action pairs is practically impossible when the system is continuous. Discretization of the state- and action-spaces is a common way of handling this problem, but generalization to state-action pairs not available in the final table must be addressed. It is for this reason that function approximation methods have been used in conjunction with RL.

Approximation of the action-value function can be accomplished through several different methods. One option is a gradient descent approach for mapping the state-action pairs to continuous weights [40, 18]. Least Squares methods have also been

explored for keeping the state-space continuous [44, 26]. Several other continuous implementations have also been investigated, including kernel-based methods [44], viscosity solution methods [33], online neural network approximations [11], and locally weighted regression [41]. A drawback of these continuous function approximators is the need to determine appropriate basis functions. While they have proven to be successfully implemented in cases where there is an easily determined appropriate basis function set, determining these basis functions is not always straightforward and is very difficult in complex systems. Machine learning methods other than RL are most often used for the approximation of value functions without the need of basis functions. This can include artificial neural networks [39], genetic algorithms [28], and instance-based methods like $k$-Nearest Neighbor [22, 31].

### *2.3.1 Artificial Neural Networks*

One of the most popular methods of machine learning today is the artificial neural network (ANN). Artificial neural networks learn hidden behavioral patterns between inputs and outputs by attempting to simulate the complex interactions of neurons in the human brain. The basic unit of the ANN is called a perceptron (analogous to neuron), and is represented in Figure 2.1.

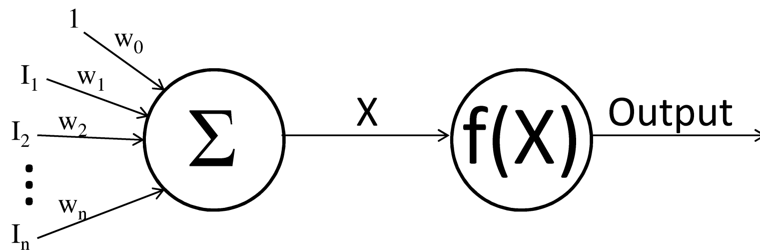


Figure 2.1: Perceptron

$$X = \sum_{i=0}^{n} w_i I_i \quad (2.6)$$

The various inputs, $I_0$-$I_n$, are combined by a weighted sum, with $I_0$=1 always present to eliminate an inherent bias. The weighted sum, represented in Equation (2.6), is provided as an input to a normalizing function. This function is often a step function (Output = +1 if X is positive, -1 if X is negative, 0 if X=0), but many other functions have often been used in its place (e.g., sigmoid). Sometimes, no normalizing function is present at all, and the output is simply the weighted sum X. The choice of this function is entirely up to the user, and benefits vary depending upon the problem.

An ANN is composed of a network of these perceptrons that are comprised of at least two layers, but usually more. An input layer is necessary, where each input is given a separate node, and a similar output layer is composed of a single node for each output. The middle layers are called the "hidden layers", and are chosen according to user design. Once the structure of the network has been set, the network is trained by feeding training data through the network and adjusting the weights to determine the most appropriate relationship for minimizing the output error. There are several different algorithms available for training a neural network, one of the most widely-used of which is the backpropagation algorithm. This algorithm first propagates a single training case through the network, and then uses the errors between the training data output and the network output to work backward and adjust weights accordingly [32]. ANNs have a reputation for being both highly accurate and fast when compared to other complex machine learning methods, and so they have often been considered for approximating the action-value function as a network of weights.

### *2.3.2 Genetic Algorithms*

Genetic algorithms are a class of machine learning algorithms that attempt to mimic the genetic behavior behind the theory of “survival of the fittest”. The idea is to provide a “population” of hypotheses that are used to approximate an unknown function, where the best hypothesis is initially unknown. The notion of which hypothesis is best depends upon a fitness function that the user defines. This fitness function is used to test individual hypotheses and provide a score based on its evaluated fitness [32].

In the first generation, hypotheses that make up the population are usually randomly generated. A single hypothesis is typically represented as a binary string, where different bit combinations represent different functions. The fitness function is designed to check how well the hypothesis approximates the function, and then generates a probability according to the following equation:

$$P(h_i) = \frac{F(h_i)}{\sum_{j=1}^{n} F(h_j)} \tag{2.7}$$

In Equation (2.7), $P(h_i)$ is the probability of selecting hypothesis $h_i$, $F(h_i)$ is the fitness function evaluated for $h_i$, and $n$ is the total number of hypotheses in the population. This probability is used to determine which hypotheses are chosen for repopulation, and then they either perform crossover operations (two hypotheses are combined to form two new hypotheses), or they mutate (one hypothesis has a single bit changed to make the new hypothesis). Once the new generation is completed, the process continues until a single hypothesis is discovered that has a fitness greater than the user-defined minimum. Using this method, it is possible to approximate the action-value function as a string of bits that represent a set of functions and weights.

### 2.3.3 k-Nearest Neighbor

Of the various machine learning methods that have been implemented in practice for action-value function approximation, the simplest and most often used has been $k$-Nearest Neighbor. It is derived from the basic assumption that each instance has the most in common with instances that are closest to it in Euclidean space [32]. In Euclidean space, $n$-dimensional information instances are given ordinates for determination of distance. The distances between these instances are determined according to the common distance formula for $n$-dimensional space, shown in Equation 2.8.

$$d(x_i, x_j) = \sqrt{\sum_{c=1}^{n} (s_c(\chi_i) - s_c(\chi_j))^2} \tag{2.8}$$

In Equation 2.8 the distance is denoted as $d$, with $\chi_i$ and $\chi_j$ being the $i$th and $j$th instances. The state dimension index is denoted by $c$, and the variables $s$ and $n$ are the state and the dimension of the state-space, respectively. Using Equation 2.8, the Euclidean distance can be determined between any two points in n-dimensional space. The $k$-Nearest Neighbor algorithm then classifies an instance based upon the average of the $k$-nearest points by Euclidean distance. This is an ideal way of handling function approximation of the action-value function due to the fact that no information is lost during the approximation. The $Q$-matrix holds a discrete table of values representing the state-action pair values according to how the state- and action-spaces were discretized. This learned function is retained, and the $k$-Nearest Neighbor algorithm can be used to determine values for any states that fall between the discrete states listed in the $Q$-matrix. The $k$-Nearest Neighbor algorithm as it applies to action-value function approximation is shown in Algorithm 2.3.

These are just a few among many machine learning methods that can be used

**Algorithm 2.3** $k$-Nearest Neighbor

- Choose value of $k \in$ positive integers
- For the current state, $s_t$:
  - Find the $k$ states in $Q(s,a)$ closest to $s_t$ by Equation 2.8
  - Create new set from the $k$-nearest states, denoted $\bar{s}$
  - Repeat for each possible action, $a$:
    - $Q(s_t,a) = \frac{1}{k}\sum_{i=1}^{k} Q(\bar{s}_i, a)$

in conjunction with Temporal-Difference RL algorithms to approximate the action-value function. In theory, any instance-based or pattern-learning method could be used for function approximation. Examples of how these methods can be used for function approximation are available in works by Kirkpatrick and Valasek [20, 22, 23, 24], Poggio and Girosi [36], and Lampton [27]. Due to the simplicity of implementation and accuracy of approximation, the approach used for function approximation in all examples in this dissertation will be $k$-Nearest Neighbor.

## 2.4 $Q$-learning Control Example

To demonstrate the ability to control a dynamical system using $Q$-learning, a bi-directional robot was simulated. This single agent was simulated with identical dynamics in each of its states (x,y) with data sampling occurring at every $T$=0.05sec. This agent was prescribed a single time constant, $\tau$ for simulating its dynamics in both the x- and y-directions. The environment the robot is allowed to traverse is a 20m by 20m square with the origin at the center. The governing equations of motion are shown in Equation 2.9, where the subscript $c$ denotes the commanded value.

$$\dot{x} = (x_c - x)\tau^{-1}$$
$$\dot{y} = (y_c - y)\tau^{-1} \tag{2.9}$$

These agent dynamics were simulated using a 4th-order Runge-Kutta integration scheme. Runge-Kutta methods are a commonly used method of numerically determining approximate solutions to ordinary differential equations. Evaluating the integration for a 4th-order Runge-Kutta solver involves determining 4 increment parameters at each integration instance. These 4 increments are based on the slope at the beginning of the interval ($k_1$), the slope at the midpoint of the interval ($k_2$), the adjusted slope at the interval midpoint ($k_3$), and the slope at the end of the interval ($k_4$). So given some dynamics, $\dot{x} = f(t, x)$, the solution at each time step can be determined by Equations 2.10-2.15.

$$\dot{x} = f(t, x),\ \ x(t_0) = x_0 \tag{2.10}$$

$$k_1 = hf(t, x_t) \tag{2.11}$$

$$k_2 = hf(t + \frac{1}{2}h, x_t + \frac{1}{2}k_1) \tag{2.12}$$

$$k_3 = hf(t + \frac{1}{2}h, x_t + \frac{1}{2}k_2) \tag{2.13}$$

$$k_4 = hf(t + h, x_t + k_3) \tag{2.14}$$

$$x_{t+h} = x_t + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \tag{2.15}$$

This allows for simulation of integrated equations of motion for some small integration timestep, $h$. At each sampling timestep,$T$, the learner evaluates the current state and chooses the appropriate action based on an $\varepsilon$-greedy policy in a $Q$-learning scheme. The possible actions are up(+y), down(-y), left(-x), and right(+x). At each $T$, the robot is required to take a new action, which corresponds to a new commanded next state. An example of how the action-space is created for a particular discretization is shown in Equation 2.16, with the proper translation of action to commanded value as shown in Equations 2.18 and 2.19.

$$a \in A = \begin{bmatrix} \uparrow & \downarrow & \rightarrow & \leftarrow & \text{stay} \end{bmatrix} \tag{2.16}$$

$$= \begin{bmatrix} +1 & -1 & +1 & -1 & 0 \end{bmatrix} m \tag{2.17}$$

$$x_c = \begin{cases} x_c + a & \text{if } a = A_3, A_4 \\ x_c & \text{else} \end{cases} \tag{2.18}$$

$$y_c = \begin{cases} y_c + a & \text{if } a = A_1, A_2 \\ y_c & \text{else} \end{cases} \tag{2.19}$$

After simulating this problem using Watkins' $Q$-learning for determining the control policy, the result is the ability to control the robot to move from randomly initialized points to a specified goal within a tolerance of $\pm 1m$. Figures 2.2-2.7 demonstrate this ability. The simulation was tested for two different discretization levels and two different goals to demonstrate that the learning methodology is not dependent on a particular discretization or goal. The first simulation uses a coarse discretization of 5m steps with a goal of $[x, y] = [0, 0]$. The second simulation uses

a finer discretization of 1m steps with a goal of $[x, y] = [3, 6]$. In these figures, the starting point is marked by $\circ$ and the final point is marked by $*$.

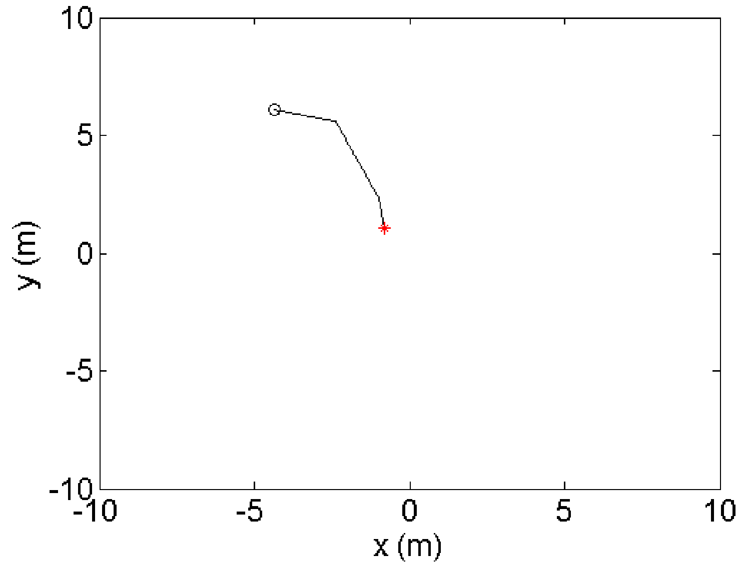


Figure 2.2: $Q$-learning Simulation of Robot - Goal = [0,0]

These results show that $Q$-learning is a capable method of learning to control an agent without the need of a dynamical model in the control policy. However, the determined control policy will be dependent upon the sampling rate, which is not always set in stone in advance and classically needs a model to determine. Determining the best sampling rate using an RL framework is discussed in the following chapter.

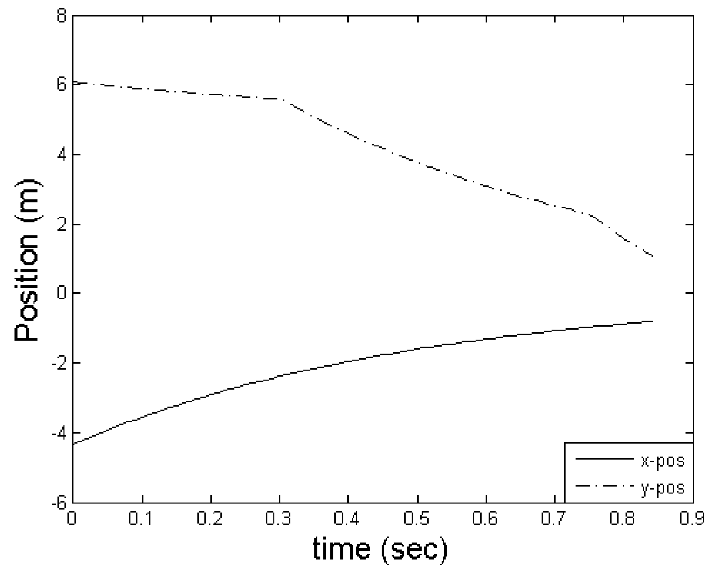


Figure 2.3: Time History of Robot - Goal = [0,0]

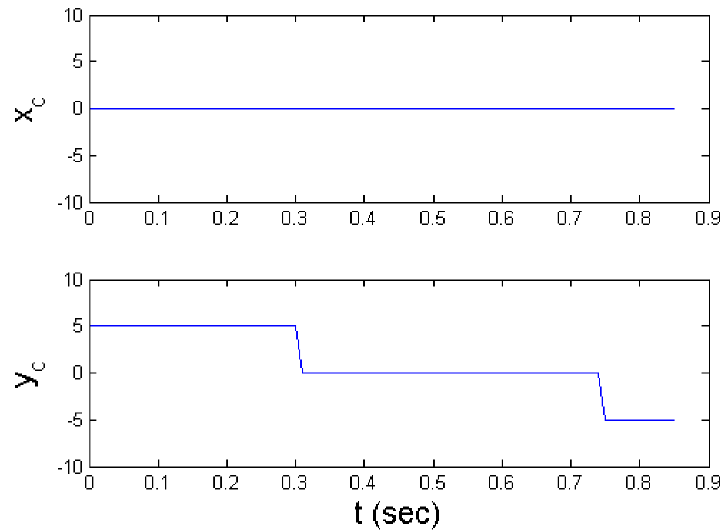


Figure 2.4: Command History of Robot - Goal = [0,0]

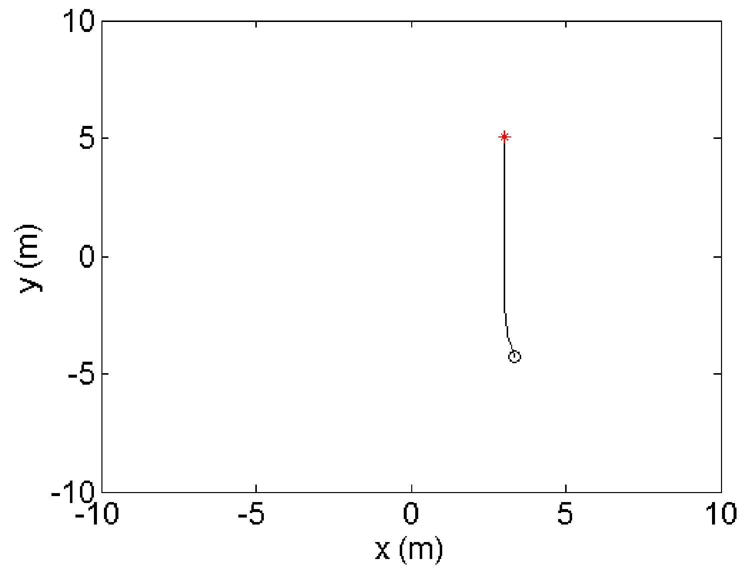


Figure 2.5: $Q$-learning Simulation of Robot - Goal = [3,6]

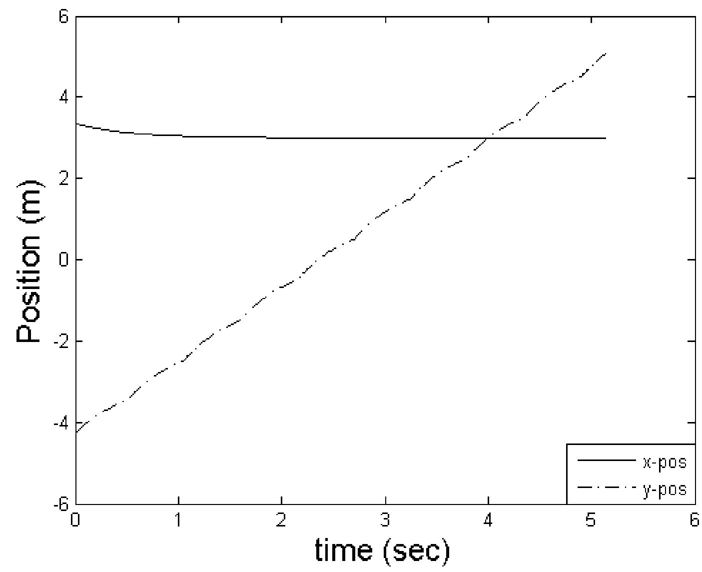


Figure 2.6: Time History of Robot - Goal = [3,6]

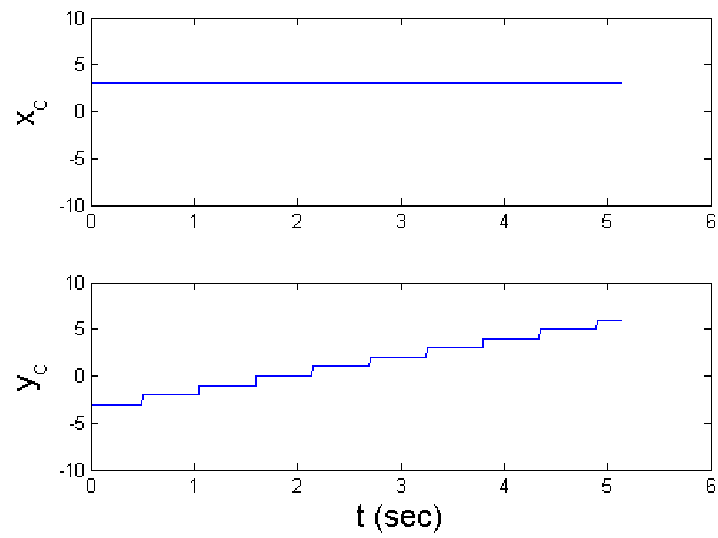


Figure 2.7: Command History of Robot - Goal = [3,6]

# 3. SAMPLED-DATA $Q$-LEARNING*

When Algorithm 2.1 is used in dynamical systems, the problem of handling time-dependencies arises. It is often overlooked that using the learned computer-based control policy on a hardware model results in a sampled-data system. The control policy is implemented on a discrete device, but the policy is learned assuming continuous dynamics. In some cases, the user realizes this and assumes a sampled-data system during the learning process, but the $Q$-learning algorithm will converge to a policy that is useful solely for that sampling rate that may not be the best rate. Here, an attempt to overcome this issue is addressed by wrapping the sample time into the learning process [25].*

## 3.1 Sampled-Data $Q$-learning Algorithm

Determining the best sample time depends on defining what is meant by "best". In general, the best sampling rate is the slowest rate that is still capable of the desired performance. In many cases it is desired to recover the frequency signal, which as a rule of thumb is achieved by sampling at twice the Nyquist frequency, or about 8-10 times the bandwidth frequency [9]. To guarantee recovery of amplitude, sampling needs to be even faster. However, in this case all that is desired is to achieve proper control authority. The goal of this algorithm is to determine the slowest sampling rate that still allows the agent to reach the goal with a success rate that is comparable to faster sample times.

Incorporating the sample time, denoted $T$, into the learning process requires

---

*Part of the material reported in this chapter is reproduced with permission of John Wiley & Sons, Inc. from *Reinforcement Learning and Approximate Dynamic Programming for Feedback Control*, eds. Frank L. Lewis and Derong Liu, 2012, John Wiley & Sons, Inc., Hoboken, NJ. Copyright 2013 by The Institute of Electrical and Electronics Engineers, Inc.

determining the value of individual sample times without adversely affecting the stability of the system during an episode. It would therefore be wise to not allow a sample time to change during a single episode. However, determining optimal action-value functions for a range of sample times requires incorporating it into the state-space. It is therefore necessary to append the state-space with $T$ while not allowing it to be affected by the action-space.

This gives rise to the question of how a particular sample time is to be selected when it cannot be affected by the action-space. It is necessary that a value be associated with each possible selection of $T$, but $T$ must be held constant throughout an episode. It is therefore proposed that a state-value function for $T$ be determined using Monte Carlo-based learning [42]. The value function can be updated according to an every-visit Monte Carlo method, shown in Equation 3.1.

$$V_T(T) \leftarrow V_T(T) + \alpha(R - V_T(T)) \tag{3.1}$$

This update rule will be the basis for *Sampled-Data Q-learning*, which is illustrated in Figure 3.1. At the beginning of each episode, the sample time value-function, $V_T$, can be used to select $T$ for the episode according to a user-defined policy. The sample time is appended to the system state vector, $s$, so that the normal $Q$-learning update rule will determine separate control policies according to different values of $T$. When the reward for a given timestep is determined, it is used to update both $Q$ and the total rewards for the episode. At the end of the episode, $V_T$ is updated by using the return, $R$. The return can be defined as some function of the rewards received, commonly either the total rewards for the episode or the average rewards for the episode. In this research, the average return was used in all simulations so that shorter paths to the goal are favored over circuitous ones.

This update rule causes $V_T$ to evolve such that episodes that experience more positive rewards than negative rewards result in the value associated with that particular $T$ increasing. Likewise, when an episode experiences more negative rewards than positive, the total value for $T$ after the episode ends will have decreased. Using this new sampled-data value function and update rule, the Sampled-Data $Q$-learning algorithm takes the form shown in Algorithm 3.1.

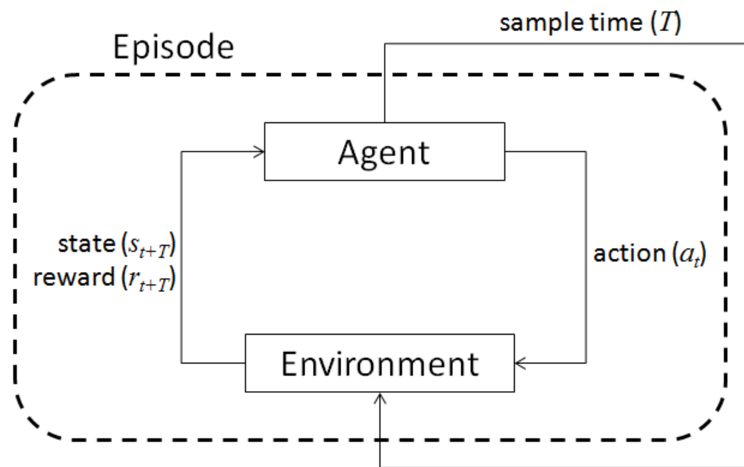


Figure 3.1: Sampled-Data $Q$-learning Diagram

## 3.2 Markov Property

To maintain that RL is a reasonable approach for this scenario, it must be demonstrated that the changes to the state-space do not affect the Markov Property. The Markov Property states that predicting the probabilistic next state of the system depends solely on the information of the current timestep, and not on any information prior to the current timestep. For this system, the Markov Property is defined according to Equation 3.2.

**Algorithm 3.1** Sampled-Data $Q$-learning (SDQL)

- Initialize $Q(\tilde{s},a)$ arbitrarily
- Initialize $V_T(T)$ arbitrarily
- Repeat for each episode:
  - Choose $T$ using policy derived from $V_T(T)$ (e.g., $\varepsilon$-Greedy)
  - Initialize $R = 0$
  - Initialize $s$, initialize $\tilde{s}$ by appending $T$ to $s$
  - Repeat for each sample timestep, $T$:
    - Choose $a$ from $\tilde{s}$ using policy derived from $Q(\tilde{s},a)$ (e.g., $\varepsilon$-Greedy)
    - Take action $a$, observe $r$, $\tilde{s}'$
    - $Q(\tilde{s},a) \leftarrow Q(\tilde{s},a) + \alpha$ [ $r + \gamma \max_{a'} Q(\tilde{s}',a')$ - $Q(\tilde{s},a)$ ]
    - $\tilde{s} \leftarrow \tilde{s}'$
  - Until $\tilde{s}$ is terminal
  - $R$ = average($r$)
  - $V_T(T) \leftarrow V_T(T) + \alpha$ [ $R$ - $V_T(T)$ ]

$$P(s_{t+T}|s_t, a_t, s_{t-T}, a_{t-T}, ..., s_0, a_0) = P(s_{t+T}|s_t, a_t) \tag{3.2}$$

For Sampled-Data $Q$-learning, the state-space is adjusted by appending the sample time, $T$, to the state vector as shown in Equation 3.3.

$$\tilde{s} = [s, T] \tag{3.3}$$

To ensure that this addition of a temporal element to the state-space does not change the Markovian behavior, it must shown that the definition of Equation 3.2 still holds for the new state-space. So given that the original system without the sample time appended is a Markov Process, the following question must be answered:

$$P(\tilde{s}_{t+T}|\tilde{s}_t, a_t, \tilde{s}_{t-T}, a_{t-T}, ..., \tilde{s}_0, a_0) \stackrel{?}{=} P(\tilde{s}_{t+T}|\tilde{s}_t, a_t) \tag{3.4}$$

First, it is important to recognize that the conditional probability can be rewritten according to Equation 3.5. Since the current (and past) state-action pairs are known to have occurred, the probability of their occurrence is equal to 1.

$$P(s_{t+T}|s_t, a_t) = \frac{P(s_{t+T}, s_t, a_t)}{\cancelto{1}{P(s_t, a_t)}} = P(s_{t+T}, s_t, a_t) \tag{3.5}$$

This information can be used to reduce the full conditional probability as shown in Equations 3.6-3.8.

$$P(\tilde{s}_{t+T}|\tilde{s}_t, a_t, \tilde{s}_{t-T}, a_{t-T}, ..., \tilde{s}_0, a_0) = P(s_{t+T}, T|s_t, a_t, s_{t-T}, a_{t-T}, ..., s_0, a_0, T) \tag{3.6}$$

$$= \frac{P(s_{t+T}, s_t, a_t, s_{t-T}, a_{t-T}, ..., s_0, a_0, T)}{\cancelto{1}{P(s_t, a_t, s_{t-T}, a_{t-T}, ..., s_0, a_0, T)}} \tag{3.7}$$

$$= P(s_{t+T}, s_t, a_t, s_{t-T}, a_{t-T}, ..., s_0, a_0, T) \tag{3.8}$$

The sample time $T$ is chosen prior to the process, and remains unchanged throughout the process. Since the time between state measurements is known, the probability of the sample time being $T$ given the state and action information is equal to 1. By the definition of conditional probability, Equation 3.8 can be reduced as shown in Equations 3.9 and 3.10.

$$\cancelto{1}{P(T|s_{t+T}, s_t, a_t, s_{t-T}, a_{t-T}, ..., s_0, a_0)} = \frac{P(s_{t+T}, s_t, a_t, s_{t-T}, a_{t-T}, ..., s_0, a_0, T)}{P(s_{t+T}, s_t, a_t, s_{t-T}, a_{t-T}, ..., s_0, a_0)} \tag{3.9}$$

$$P(s_{t+T}, s_t, a_t, s_{t-T}, a_{t-T}, ..., s_0, a_0, T) = P(s_{t+T}, s_t, a_t, s_{t-T}, a_{t-T}, ..., s_0, a_0, ) \tag{3.10}$$

Now, it can be seen that this is the original Markovian system, which reduces as shown in Equations 3.11-3.13.

$$\begin{aligned} P(s_{t+T}, s_t, a_t, s_{t-T}, a_{t-T}, ..., s_0, a_0) &= P(s_{t+T}|s_t, a_t, s_{t-T}, a_{t-T}, ..., s_0, a_0) \\ &\quad \times \cancelto{1}{P(s_t, a_t, s_{t-T}, a_{t-T}, ..., s_0, a_0)} && (3.11) \\ &= P(s_{t+T}|s_t, a_t, s_{t-T}, a_{t-T}, ..., s_0, a_0) && (3.12) \\ &= P(s_{t+T}|s_t, a_t) && (3.13) \end{aligned}$$

Since the full next-step conditional probability of the modified state-space results in the same prediction probability of the unmodified state-space, it can be seen that the Markov Property is preserved for a system that was originally Markovian and has the sample time appended to the state-space.

### 3.3 Reward Shaping

An important consideration in forming the RL problem is the choice of reward structure. Rewards are used to define the problem to be solved, guiding the learner towards specific goals and away from undesirable states. Typically, constant positive rewards are given for achieving a goal state, constant negative rewards are given for

ending up in bad states, and neutral (zero) rewards are given for all other states. While this is the most commonly used system, rewards can be shaped in other ways. Choosing a reward function can be dangerous since one must ensure that the learner does not change the overall goal of convergence [34]. As demonstrated by , to maintain that the near-optimal policies remain unchanged it is necessary to shape the rewards by adding a potential function to the original reward structure.

In Sampled-Data $Q$-learning, the reward is shaped by multiplying the reward by a function of the sample time. In the case where the function multiplied is invariant of the reward constant, it can be factored out of the online process and be multiplied by the value function itself. If $V_T^{(0)}(T)$ is the initial value function at sample time $T$, $V_T^{(1)}(T)$ is the value function after 1 episode at sample time $T$, and $R^{(1)}(T)$ is the return received in episode 1 as a function of $T$, then by the Sampled-Data $Q$-learning update rule the new value function is as shown in Equation 3.14.

$$V_T^{(1)}(T) = (1-\alpha)V_T^{(0)}(T) + \alpha R^{(1)}(T) \tag{3.14}$$

When the learning progresses to the second episode experienced at sample time $T$, the value function is similarly updated. It can easily be seen that substituting Equation 3.14 into Equation 3.15 results in the form shown in Equation 3.16

$$V_T^{(2)}(T) = (1-\alpha)V_T^{(1)}(T) + \alpha R^{(2)}(T) \tag{3.15}$$

$$= (1-\alpha)^2 V_T^{(0)}(T) + (1-\alpha)\alpha R^{(1)}(T) + \alpha R^{(2)}(T) \tag{3.16}$$

In the same way, the value after 3 episodes can be expanded. If one considers the value after $i$ episodes, the expanded value function can be derived. This is shown in

Equation 3.20.

$$V_T^{(3)}(T) = (1-\alpha)V_T^{(2)}(T) + \alpha R^{(3)}(T) \tag{3.17}$$

$$= (1-\alpha)^3 V_T^{(0)}(T) + (1-\alpha)^2 \alpha R^{(1)}(T) + (1-\alpha)\alpha R^{(2)}(T) + \alpha R^{(3)}(T) \tag{3.18}$$

$$= (1-\alpha)^3 V_T^{(0)}(T) + \alpha \sum_{j=1}^{3} (1-\alpha)^{3-j} R^{(j)}(T) \tag{3.19}$$

$$\vdots$$

$$V_T^{(i)}(T) = (1-\alpha)^i V_T^{(0)}(T) + \alpha \sum_{j=1}^{i} (1-\alpha)^{i-j} R^{(j)}(T) \tag{3.20}$$

The fully expanded form of the value function shown in Equation 3.20 shows that because $0 < \alpha < 1$, as $i \to \infty$ the initial value $V_T^{(0)}(T)$ becomes a negligible term. Because the initial value is arbitrary and negligible, it is simplest to choose an initial value of $V_T^{(0)}(T) = 0$. This simplifies the value function to the form in Equation 3.21.

$$V_T^{(i)}(T) = \alpha \sum_{j=1}^{i} (1-\alpha)^{i-j} R^{(j)}(T) \tag{3.21}$$

In the original problem, the reward structure is a constant chosen in each episode based on the goal and boundary specifications, and the episode terminates when a non-zero reward is received. If this reward structure is unmodified, and not a function of the sample time, Equation 3.21 becomes defined according to Equation 3.22, where $\tilde{V}_T^{(i)}$ is the value function as a function of only the constant reward after $i$ episodes, $steps^{(j)}$ is the number of steps experienced within episode $j$, and the return $R^{(j)}$ from episode $j$ is the average reward received during that episode.

$$\tilde{V}_T^{(i)} = \alpha \sum_{j=1}^{i} \frac{(1-\alpha)^{i-j} r_c^{(j)}}{steps^{(j)}} \tag{3.22}$$

If the reward function $r^{(j)}(T)$ is considered to be a linear combination of the constant reward given in each episode, $r_c^{(j)}$, and some independent function of the sample time, $f(T)$, the value function can be further expanded. If the reward function is additive, it is as shown in Equation 3.23.

$$r^{(j)}(T) = r_c^{(j)} + f(T) \tag{3.23}$$

With this reward structure, the value function becomes as shown in Equation 3.24.

$$V_T^{(i)}(T) = \alpha \sum_{j=1}^{i} (1-\alpha)^{i-j} \frac{(r_c^{(j)} + f(T))}{steps^{(j)}} \tag{3.24}$$

This can be further expanded, and the resulting value function is as shown in Equation 3.25

$$V_T^{(i)}(T) = \alpha \sum_{j=1}^{i} \frac{(1-\alpha)^{i-j} r_c^{(j)}}{steps^{(j)}} + \alpha f(T) \sum_{j=1}^{i} \frac{(1-\alpha)^{i-j}}{steps^{(j)}} \tag{3.25}$$

With the original constant reward function, the value function was defined according to Equation 3.22, and was given the notation $\tilde{V}_T^{(i)}$. With this substitution, it can be seen how the additive function $f(T)$ affects the overall value in Equation 3.26.

$$V_T^{(i)}(T) = \tilde{V}_T^{(i)} + \alpha f(T) \sum_{j=1}^{i} \frac{(1-\alpha)^{i-j}}{steps^{(j)}} \tag{3.26}$$

So if the function $f(T)$ is independent of episodic rewards, it can be factored out of the online process and added to the value function itself outside of the episodes. Similarly, it can be considered how a sample time function $f(T)$ affects the value

function if it is multiplicative. This reward structure is as shown by Equation 3.27.

$$r^{(j)}(T) = r_c^{(j)} f(T) \tag{3.27}$$

By substituting the reward structure of Equation 3.27 into the value function of Equation 3.21, the value function becomes as defined in Equation 3.28.

$$V_T^{(i)}(T) = \alpha \sum_{j=1}^{i} \frac{(1-\alpha)^{i-j} r_c^{(j)} f(T)}{steps^{(j)}} \tag{3.28}$$

The function $f(T)$ can be easily factored out, and this results in the form shown in Equation 3.29.

$$V_T^{(i)}(T) = \alpha f(T) \sum_{j=1}^{i} \frac{(1-\alpha)^{i-j} r_c^{(j)}}{steps^{(j)}} \tag{3.29}$$

Now if the original value function $\tilde{V}_T^{(i)}$ is substituted into Equation 3.29, it can be seen that the sample time function $f(T)$ can be factored out of the online process. This is shown in Equation 3.30.

$$V_T^{(i)}(T) = f(T) \tilde{V}_T^{(i)} \tag{3.30}$$

By comparing Equations 3.26 and 3.30, it can be seen that either form of the reward structure results in the ability to factor the sample time function out of the online process, but the multiplicative case is much simpler to implement. This property depends on the function $f(T)$ being independent of each episode, and so it must be used the same regardless of the constant reward, $r_c$. For example, if the reward was shaped so that the positive goal reward was multiplied by a function $f(T)$ but the negative boundary reward was left constant, the function $f(T)$ could

not be factored out of the episodic summation.

It is now interesting to consider the slopes of these value functions considered over the sample times. By analyzing these derivatives, it can be seen that the change in value over sample times after $i$ episodes has a significant difference between the two forms of the value function. As can be seen in Equation 3.31, when the additive form of the reward function is used from Equation 3.23, the constant reward drops out of the derivative. However, it can be seen in Equation 3.32 that when the multiplicative form of the reward is used from Equation 3.27 the constant reward does have an effect on the variation in value over $T$. So choosing which form to use depends on whether it is desired for the reward given in an episode to affect the variation in value. Since the desire is to encourage achieving more positive rewards and fewer negative rewards based on the value of $T$ chosen, the multiplicative form will be used.

$$\frac{\partial V_T^{(i)}}{\partial T}(T) = \alpha \frac{\partial f(T)}{\partial T} \sum_{j=1}^{i} \frac{(1-\alpha)^{i-j}}{steps^{(j)}} \tag{3.31}$$

$$\frac{\partial V_T^{(i)}}{\partial T}(T) = \alpha \frac{\partial f(T)}{\partial T} \sum_{j=1}^{i} \frac{(1-\alpha)^{i-j} r_c^{(j)}}{steps^{(j)}} \tag{3.32}$$

The next consideration that must be made is what function should be used for shaping the rewards with sample times. To shape the reward, it must be considered what problem is being solved. In the case of $Q$-learning-based control, the problem being solved is the reaching of a goal state while avoiding boundary states. In Sampled-Data $Q$-learning, the objective is expanded to also determine the largest sample time that can be used while still solving the original $Q$-learning control problem. This indicates that the reward must be a function of the sample time, but the choice of the function can have implications on the result. One possible implementation would be the reward structure in Equation 3.33.

$$r(T) = r_c T \tag{3.33}$$

Considering this reward reveals a simple linearly multiplicative function of the sample time where $f(T) = T$. This means that the first derivative is invariant of the sample time. This is the simplest function of the sample time that can be used, but the fact that the slope in value function is invariant of sample time may be problematic. The next simplest implementation would be a polynomial of degree 2, and it has the benefit of having a slope that varies with sample time.

$$r(T) = r_c T^2 \tag{3.34}$$

This quadratic implementation has the benefit of making rewards larger as sample time increases than the linear counterpart, but only when $T > 1$. So it can be beneficial to use the square root function instead when $T < 1$. This final version of the reward structure is shown in Equations 3.35-3.37, where the sample time function $f(T)$ is multiplicative and quadratic in $T$.

$$r(T) = \begin{cases} r_c\sqrt{T} & \text{if } 0 < T < 1 \\ r_c T^2 & \text{if } T \geq 1 \end{cases} \tag{3.35}$$

$$V_T^{(i)}(T) = \begin{cases} \alpha T^{\frac{1}{2}} \sum_{j=1}^{i} \frac{(1-\alpha)^{i-j} r_c^{(j)}}{steps^{(j)}} & \text{if } 0 < T < 1 \\ \alpha T^2 \sum_{j=1}^{i} \frac{(1-\alpha)^{i-j} r_c^{(j)}}{steps^{(j)}} & \text{if } T \geq 1 \end{cases} \tag{3.36}$$

$$\frac{\partial V_T^{(i)}}{\partial T}(T) = \begin{cases} \frac{1}{2}\alpha T^{-\frac{1}{2}} \sum_{j=1}^{i} \frac{(1-\alpha)^{i-j} r_c^{(j)}}{steps^{(j)}} & \text{if } 0 < T < 1 \\ 2\alpha T \sum_{j=1}^{i} \frac{(1-\alpha)^{i-j} r_c^{(j)}}{steps^{(j)}} & \text{if } T \geq 1 \end{cases} \tag{3.37}$$

To ensure that the choice of reward functions has merit, a variety of reward functions were tested in simulation using the 2D robot example. After allowing each

simulation to run for 10,000 episodes with different reward functions, the resulting best sample time was used in a Monte Carlo simulation. Table 3.1 shows the percentage of successful goal achievements based on the sample time used in the simulation, and if a reward function chose that sample time it is indicated. Table 3.1 reveals that for this simulation, of the reward functions tested a quadratic reward provides the longest sample time before success begins to decline.

Table 3.1: Reward Function Success Rate

| $T$ | Successes | $f(T)$ |
|---|---|---|
| 0.01 | 84% | 1 |
| 0.02 | 83% | |
| 0.03 | 87% | |
| 0.04 | 82% | $T$ |
| 0.05 | 84% | |
| 0.06 | 83% | $T^2$ |
| 0.07 | 77% | $T^3, e^T$ |
| 0.08 | 75% | |
| 0.09 | 71% | |
| 0.10 | 69% | |

### 3.4 Sampled-Data $Q$-learning Examples

To demonstrate the Sampled-Data $Q$-learning algorithm, examples of a dynamical system controlled by $Q$-learning were constructed. The first example shown here was chosen to be an inverted pendulum on a cart. This is a good example of a simple dynamical system where the choice of sample time can have a drastic impact on the ability to control the system successfully. The other examples shown here are 2 different approaches at simulating a simple robot with first-order dynamics, which will be the basis for dynamics learning and later used for multiagent system

simulations.

### *3.4.1 Inverted Pendulum*

In this example, the system consists of an inverted pendulum on a cart. The pendulum is allowed to rotate freely about the point of connection, and the only control input available is the force applied to the cart along the $x$-axis. This system is shown in Figure 3.2.

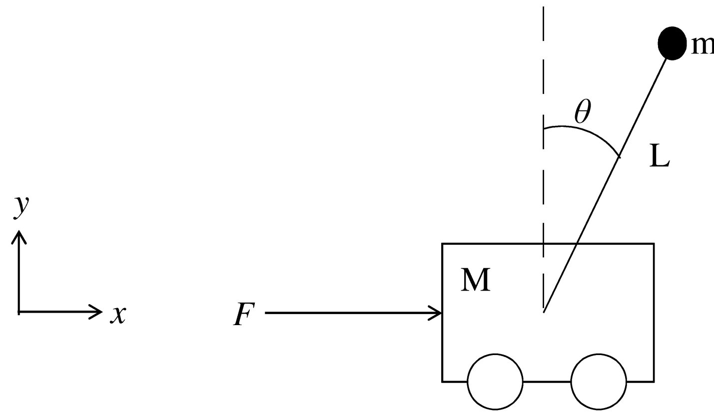


Figure 3.2: Inverted Pendulum

The position-level states that describe this system's orientation are $x$ and $\theta$. To describe the dynamics of this system, the state-space of Equation 3.38 is used.

$$\underline{x} = [x \quad \dot{x} \quad \theta \quad \dot{\theta}]^{\mathrm{T}} = [x_1 \quad x_2 \quad x_3 \quad x_4]^{\mathrm{T}} \tag{3.38}$$

The full nonlinear dynamics can then be derived in this form, as is demonstrated in Appendix A. The final dynamical equations used to simulate this inverted pendulum on a cart are shown in Equation 3.39.

$$
\begin{aligned}
\dot{x_1} &= x_2 \\
\dot{x_2} &= \frac{F + mLx_4^2 \sin(x_3) - mg\sin(x_3)\cos(x_3)}{M + m\sin^2(x_3)} \\
\dot{x_3} &= x_4 \\
\dot{x_4} &= \frac{g\sin(x_3) - \dot{x_2}\cos(x_3)}{L}
\end{aligned} \tag{3.39}
$$

To determine the proper state-space for RL, one might assume that the full state vector should be used. However, in this case it can be seen that including the position ($x$) and the cart velocity ($\dot{x}$) will not contribute much since it does not affect the pendulum orientation ($\theta$) explicitly. In fact, it can be seen from the equations of motion in Equation 3.39 that it is the acceleration of the cart that explicitly contributes to the pendulum orientation dynamics. Since the only states propagated and known for the cart at any given time interval are $x$ and $\dot{x}$, an estimate of the acceleration is used for the RL state space according to:

$$
\ddot{x}_{est} = \frac{\dot{x}_t - \dot{x}_{t-1}}{T} \tag{3.40}
$$

To learn action-value functions for each sample time $T$, the state-space must also include the sample time. The action-space must likewise be selected, but in this problem the only thing that makes sense is to use the force $F$ as the action since it is the only control. The state- and action-spaces for the RL formulation of this problem are as shown in Equations 3.41 and 3.42, respectively.

$$
s = [\ddot{x}_{est} \quad \theta \quad \dot{\theta} \quad T]^{\mathrm{T}} \tag{3.41}
$$

$$
a = F \tag{3.42}
$$

The next step is to define the reward structure. The rewards depend upon both the goals of the system and the constraints placed on the system. For this example, the goal was determined to be the state where both $\theta = 0$ and $\dot{\theta} = 0$, with a tolerance of $\pm 1^\circ$ and $\pm 2^\circ$/sec, respectively. If this goal was achieved, a positive reward of +10 was used to update the action-value function. The horizontal position and velocity of the cart were not considered for determination of the goal achievement. This structure provides the ranges for which positive rewards are achieved, but it is likewise necessary to define the constraints for which negative rewards are given. Again, the cart position and velocity do not contribute to the negative rewards by remaining unconstrained. The pendulum orientation was constrained so that it would not be allowed to move beyond $\pm 12^\circ$ without incurring negative rewards. The angular velocity was constrained so that it was not allowed to go beyond $\pm 20^\circ$/sec without incurring negative rewards. If these constraints were violated, a negative reward of -10 was used to update the action-value function. All orientations that do not yield either a positive or negative reward receive a neutral reward of 0. Thus the reward structure is:

$$r = \begin{cases} +10 & \text{if } |\theta| < 1^\circ \text{ and } |\dot{\theta}| < 2^\circ/\text{sec} \\ -10 & \text{if } |\theta| > 12^\circ \text{ or } |\dot{\theta}| > 20^\circ/\text{sec} \\ 0 & \text{otherwise} \end{cases} \tag{3.43}$$

With the RL representation determined, the simulation was performed. The parameters for the inverted pendulum cart were set as follows. The mass of the cart, $M$, was set to be 5kg. The mass at the end of the pendulum, $m$, was set to be 1kg, with the pendulum rod considered to be massless for simplicity. The length of the pendulum rod, $L$, was chosen to be 2m. The total range of possible actions was chosen to be $F$ = -40N to 40N in 20N intervals. This system was

then simulated continuously by using a 4th-order Runge-Kutta method with the full nonlinear dynamics from Equation 3.39 and a small integration timestep of $h = 0.01$sec. The system was treated as a sampled-data system by sampling the states at a rate of $T$ chosen by an $\varepsilon$-greedy policy from $V_T$ at the beginning of each episode. The value assigned to $\varepsilon$ for this simulation was $\varepsilon = 0.7$. The total range of possible sample times from which to choose was $T = 0.01$sec to 1.0sec in 0.01sec intervals. For this simulation, $\alpha$ was chosen as a constant 0.01 and $\gamma$ was selected to be a constant 0.7. Each learning simulation was allowed to last until $t_f = 5$sec.

When determining the length of an episode, there are many possibilities. Here, an episode is defined to be some significant portion of the learning process. For this particular problem, an episode was determined to last until either $t = t_f$ or $|\theta| > 12°$. The simulation was then allowed to run for many episodes to converge to both the optimal value of $T$ and the optimal action-value function, $Q$.

When using the SDQL algorithm on the inverted pendulum example, two different cases were tested. The first scenario involves using no reward shaping based on sample time. This is done as a sanity check since it should converge to a minimum sample time in the event that there is no sample time reward shaping. The second scenario tested sample time reward shaping.

Using the representation of the inverted pendulum control problem described above, the simulation was allowed to learn for 10,000 episodes using MATLAB. This translated to roughly 12 hours of real time. After the learning was completed, the action-value function was tested, and an example of this is shown in the following figures. The sample time value function, $V_T$, converged to the highest value of the sample time being $T = 0.01$sec. This makes sense for this particular experiment since no constraints were placed on sample time length in the reward structure. With no function of the sample time included in the reward function, the sample time value

function after 10,000 episodes is as shown in Table 3.2. The mean sample time value and variance of values over sample times are shown in Equations 3.44 and 3.45.

Table 3.2: SDQL Pendulum Result - Constant Reward

| $T(sec)$ | $V_T$ |
|---|---|
| 0.01 | 11.7 |
| 0.02 | 4.7 |
| 0.03 | 4.6 |
| 0.04 | 7.1 |
| 0.05 | 2.7 |
| 0.06 | 1.7 |
| 0.07 | 4.6 |
| 0.08 | 2.8 |
| 0.09 | 3.4 |
| 0.10 | 1.8 |

$$E[V_T] = 4.51 \tag{3.44}$$

$$Var[V_T] = 9.01 \tag{3.45}$$

The action-value function is too large to report here, but the mean and covariance over possible actions is of interest. These are shown in Equations 3.46 and 3.47, respectively. In Equation 3.46, it is seen that the average values are all close, but the largest values are on the smaller forces. The covariance matrix in Equation 3.47 shows that the values in the action-value function do not vary drastically by state.

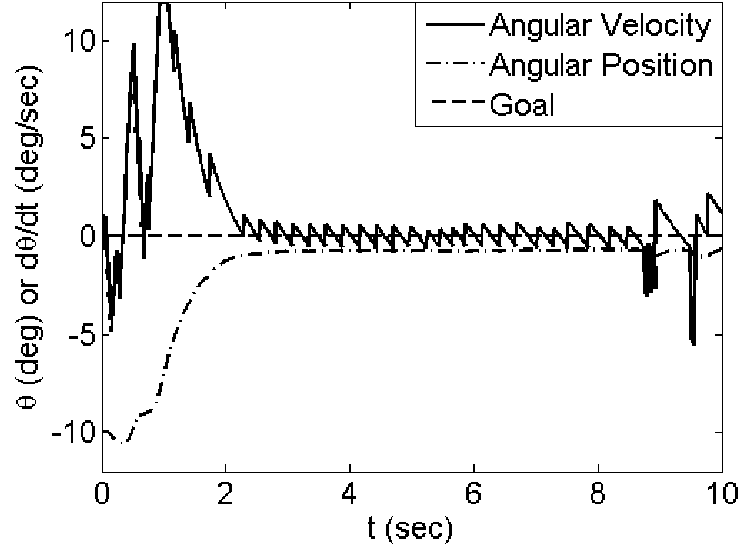


Figure 3.3: Inverted Pendulum State Time History: Constant Reward

$$E[Q] = \begin{cases} 0.3031 & , F = -40N \\ 0.5751 & , F = -20N \\ 0.4941 & , F = 0N \\ 0.4640 & , F = 20N \\ 0.4264 & , F = 40N \end{cases} \tag{3.46}$$

$$Cov[Q] = \begin{bmatrix} 110 & -1 & 13 & -2 & 1 \\ -1 & 118 & 4 & 2 & 3 \\ 13 & 4 & 172 & -5 & 11 \\ -2 & 2 & -5 & 113 & 2 \\ 1 & 3 & 11 & 2 & 113 \end{bmatrix} \times 10^{-2} \tag{3.47}$$

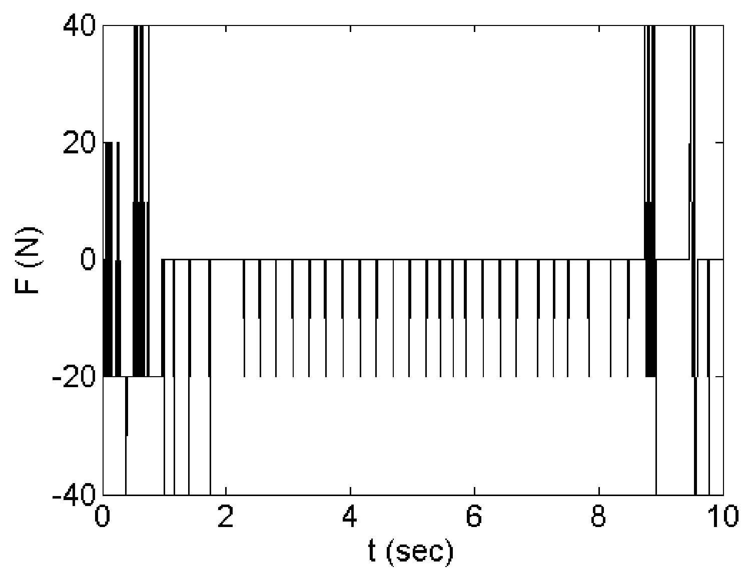


Figure 3.4: Inverted Pendulum Force Time History: Constant Reward

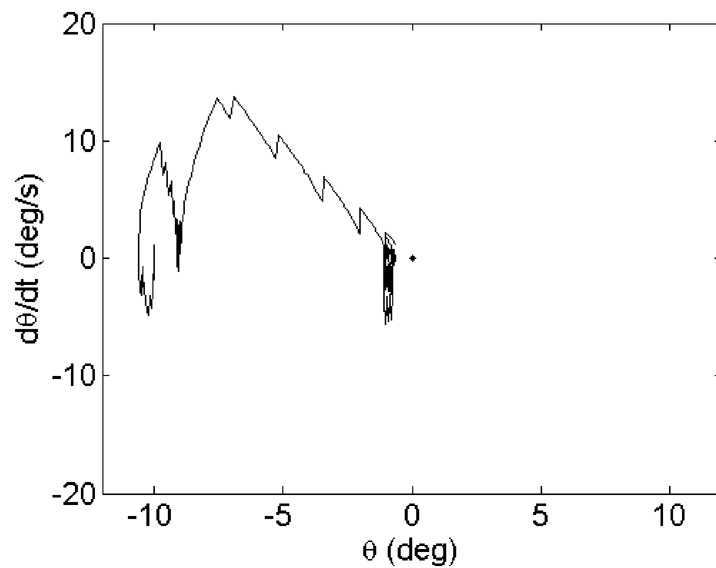


Figure 3.5: Inverted Pendulum Phase Diagram: Constant Reward

In Figure 3.3, the cart began stationary with no pendulum angular velocity and an initial orientation of $\theta = -10°$. As can be seen, by exploiting the learned action-value function the control policy was able to stabilize the pendulum to the goal within the 10 second timespan alloted. After reaching the goal region, the control policy was able to use occasional impulses to keep the angle within the goal region of $0° \pm 1°$, as seen in Figure 3.4. Figure 3.5 shows that the system stabilizes near the point (0,0) in phase space. This indicates a successful control policy learned using Sampled-Data $Q$-learning.

The second case tested involves reward shaping based on the sample time, with a multiplicative quadratic reward function. After 10,000 learning episodes, the SDQL algorithm was able to converge to a maximum-value sample time of $T = 0.08$ sec. The $V_T$ values are reported in Table 3.3. The mean sample time value and variance of values over sample times are shown in Equations 3.48 and 3.49.

Table 3.3: SDQL Pendulum Result - Quadratic Reward

| $T(sec)$ | $V_T$ |
|---|---|
| 0.01 | 2.7 |
| 0.02 | 5.2 |
| 0.03 | 7.0 |
| 0.04 | 8.7 |
| 0.05 | 9.1 |
| 0.06 | 6.0 |
| 0.07 | 9.8 |
| 0.08 | 10.9 |
| 0.09 | 9.9 |
| 0.10 | 9.3 |

$$E[V_T] = 7.86 \tag{3.48}$$

$$Var[V_T] = 6.60 \tag{3.49}$$

With the control policy using the maximum-value sample time of $T = 0.08$ sec, the inverted pendulum cart is successfully controlled as shown in Figures 3.6-3.8. Figure 3.6 shows that while it is more difficult to control the cart with the longer sample time limitations, it is able to successfully move the pendulum to the center and use force inputs to keep it in the tolerance of $\pm 1°$.

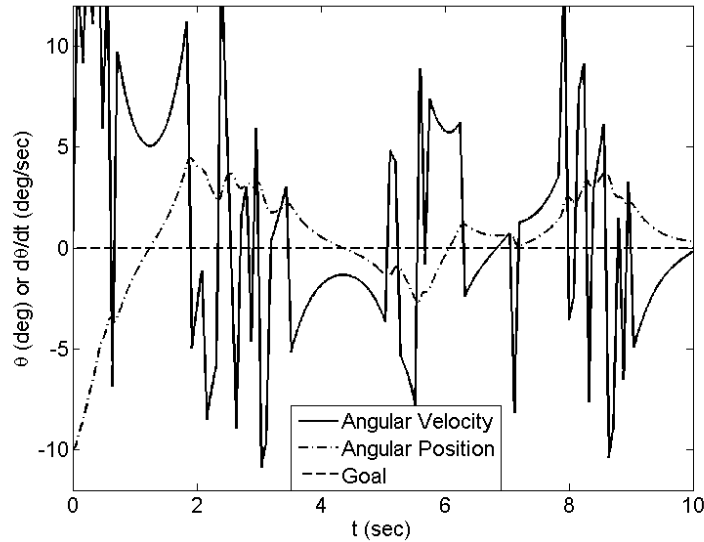


Figure 3.6: Inverted Pendulum State Time History: Quadratic Reward

The action-value function is too large to report here, but the mean and covariance over possible actions is of interest. These are shown in Equations 3.50 and 3.51, respectively. The values in the action-value function are smaller than before due to the reward shaping, and so the mean values are an order of magnitude smaller while the covariances are 3 orders of magnitude smaller. In Equation 3.50, it is seen that the average values favor smaller force responses. The greatest reinforcement occurred

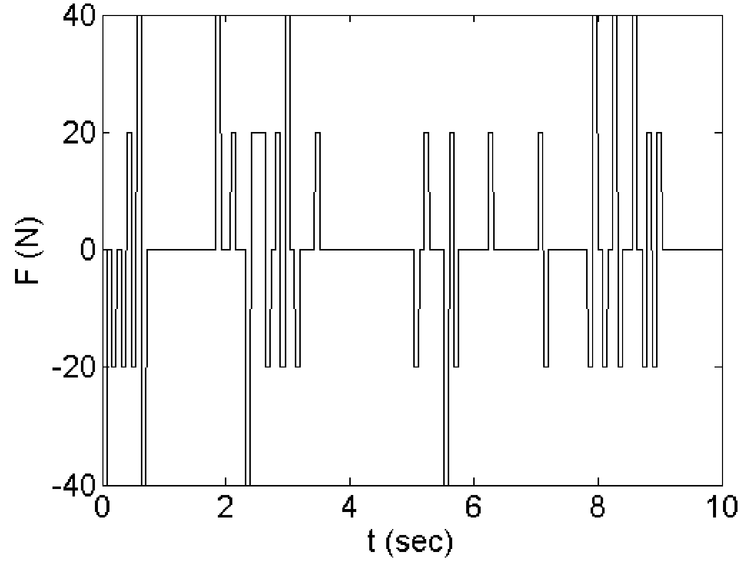


Figure 3.7: Inverted Pendulum Force Time History: Quadratic Reward

at $F = 0N$ due to time spent within the goal range. The larger forces are less helpful as the pendulum approaches the goal so they are favored less often. The covariance matrix in Equation 3.51 shows that the values in the action-value function do not vary drastically by state.

$$E[Q] = \begin{cases} 0.0257 & , F = -40N \\ 0.0351 & , F = -20N \\ 0.0538 & , F = 0N \\ 0.0316 & , F = 20N \\ 0.0243 & , F = 40N \end{cases} \tag{3.50}$$

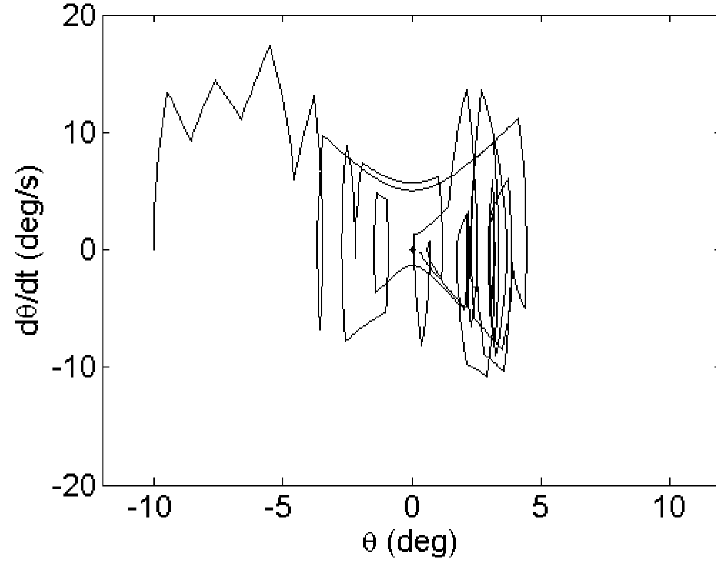


Figure 3.8: Inverted Pendulum Phase Diagram: Quadratic Reward

$$Cov[Q] = \begin{bmatrix} 130 & 4 & -18 & -10 & 16 \\ 4 & 120 & 35 & 9 & 1 \\ -18 & 35 & 410 & 48 & 4 \\ -10 & 9 & 48 & 170 & 3 \\ 16 & 1 & 4 & 3 & 92 \end{bmatrix} \times 10^{-5} \tag{3.51}$$

### *3.4.2 Rotating Robot*

For the next experiment, the Sampled-Data $Q$-learning algorithm was tested on a robot that travels forward with a constant speed and is able to rotate to change its heading angle. This is an important example since it is analogous to high-level commands of aircraft systems. This system is crafted so that the only actions required by the learning agent are commanded changes in heading angle.

$$\dot{x} = V \cos \psi \tag{3.52}$$

$$\dot{y} = V \sin \psi \tag{3.53}$$

$$\dot{\psi} = (\psi_c - \psi)\tau^{-1} \tag{3.54}$$

After each sample timestep $T$, the learner evaluates the current position and heading angle for making a choice of commanded change in heading angle. The time constant $\tau$ for the commanded change of the heading angle $\psi$ is chosen to be $\tau = 0.2$ sec. The options for commanded change in heading are as shown in Equation 3.55.

$$\psi_c = [-45^\circ \; 0^\circ \; 45^\circ]^{\mathrm{T}} \tag{3.55}$$

With the dynamics described in Equations 3.52-3.55, the commanded bank angle can be changed by the learner according to the policy, $\pi$. The sample time, $T$, is determined by the Sampled-Data $Q$-learning algorithm with possible sample times chosen from $T = 0.01$ sec to $T = 0.1$ sec in intervals of $T = 0.01$ sec. The reward shaping used for this simulation was defined by Equation 3.56, where the constant reward was $r_c = -10$ for the boundary, $r_c = 0$ for neutral states, and $r_c = 100$ for the goal.

$$r(T) = \begin{cases} r_c\sqrt{T} & \text{if } 0 < T < 1 \\ r_c T^2 & \text{if } T \geq 1 \end{cases} \tag{3.56}$$

The robot is placed in a grid with the origin placed at the center and extending $\pm 10$m in the x-direction and $\pm 10$m in the y-direction. The goal is for the robot to reach the center of the space at $[0, 0] \pm [1, 1]$m. After 10,000 episodes, the Sampled-Data $Q$-learning algorithm converged to a near-optimal control policy with a maximum-value sample time of $T = 0.07$ sec, as shown in Table 3.4. The

mean sample time value and variance of values over sample times are shown in Equations 3.57 and 3.58. It can be seen that due to the different reward structure and amount of time per episode, the value associated with the best sample time receives far greater reinforcement than that of the inverted pendulum example. The variance in values is greater in this case by several orders of magnitude.

Table 3.4: SDQL Robot Result - Quadratic Reward

| $T(sec)$ | $V_T$ |
|---|---|
| 0.01 | 29.1 |
| 0.02 | 29.9 |
| 0.03 | 2.4 |
| 0.04 | 42.9 |
| 0.05 | 51.4 |
| 0.06 | 73.4 |
| 0.07 | 2271.8 |
| 0.08 | 14.7 |
| 0.09 | 29.2 |
| 0.10 | 120.8 |

$$E[V_T] = 266.56 \tag{3.57}$$

$$Var[V_T] = 4.98 \times 10^5 \tag{3.58}$$

The action-value function itself is far too large to report here, but the mean and covariance over possible actions is of interest. These are shown in Equations 3.59 and 3.60, respectively. In Equation 3.59, it is seen that for the reward structure of this problem, the mean values are all less than 1, and a lower value was given on average to no change in heading angle than the turning actions.

$$E[Q] = \begin{cases} 0.0811 & , \Delta\psi = -45^\circ \\ 0.0745 & , \Delta\psi = 0^\circ \\ 0.0876 & , \Delta\psi = 45^\circ \end{cases} \tag{3.59}$$

$$Cov[Q] = \begin{bmatrix} 0.8771 & 0.0797 & 0.0914 \\ 0.0797 & 0.9442 & 0.0154 \\ 0.0914 & 0.0154 & 1.1810 \end{bmatrix} \tag{3.60}$$

After completing the learning, the Monte Carlo results show the ability to successfully control the robot to the center of the space from each quadrant, as shown in Figures 3.9-3.20. In these figures, the starting point is marked by $\circ$ and the final point is marked by $*$.

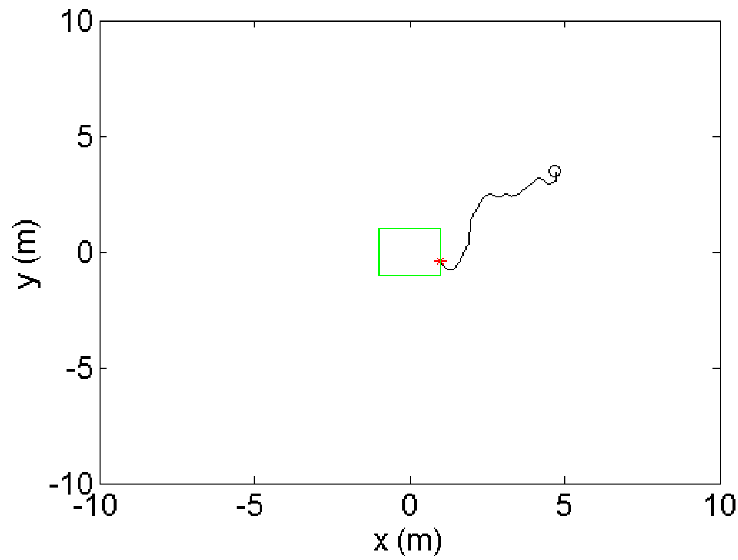


Figure 3.9: Robot Simulation - Quadrant 1

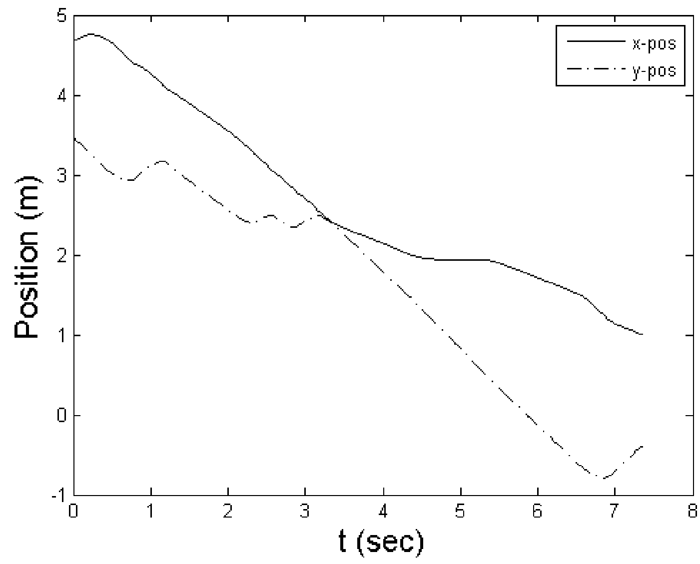


Figure 3.10: State History of Robot - Quadrant 1

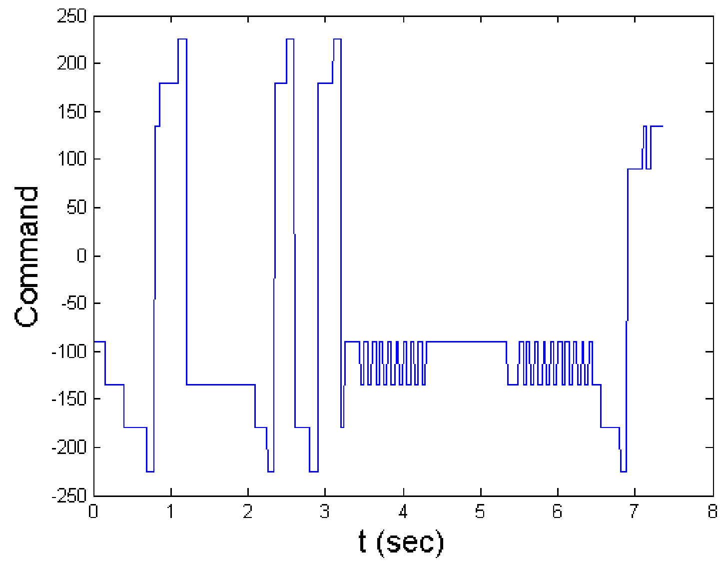


Figure 3.11: Command History of Robot - Quadrant 1

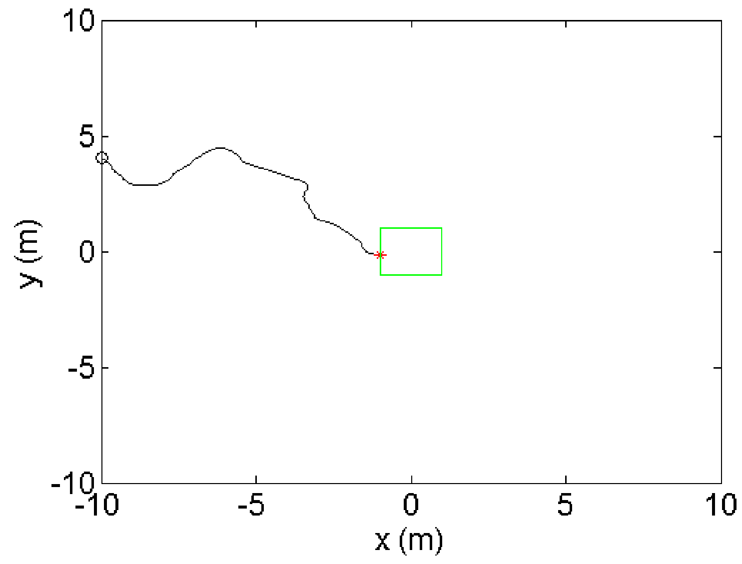

Figure 3.12: Robot Simulation - Quadrant 2

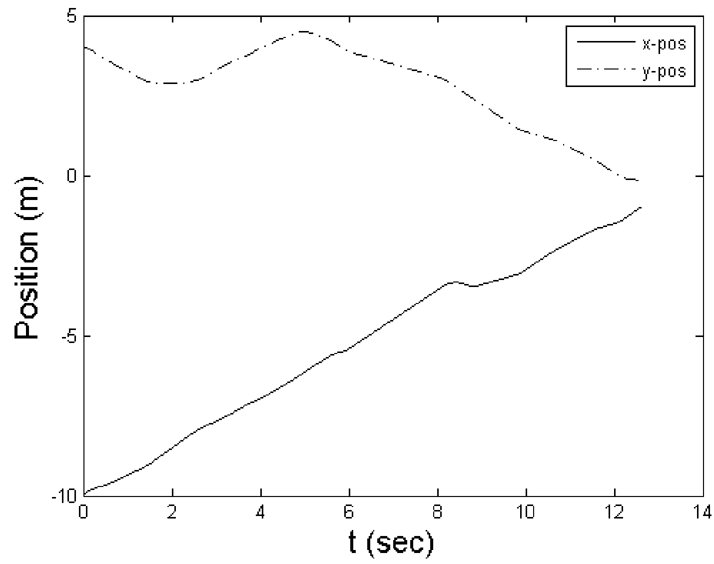

Figure 3.13: State History of Robot - Quadrant 2

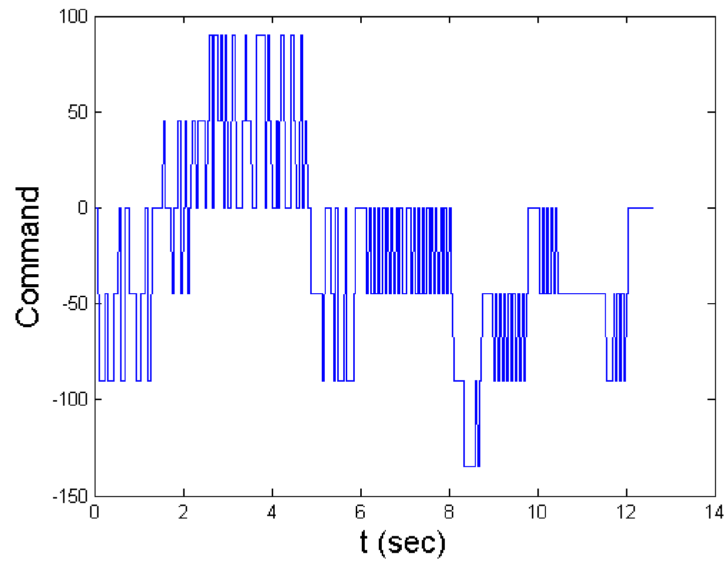

Figure 3.14: Command History of Robot - Quadrant 2

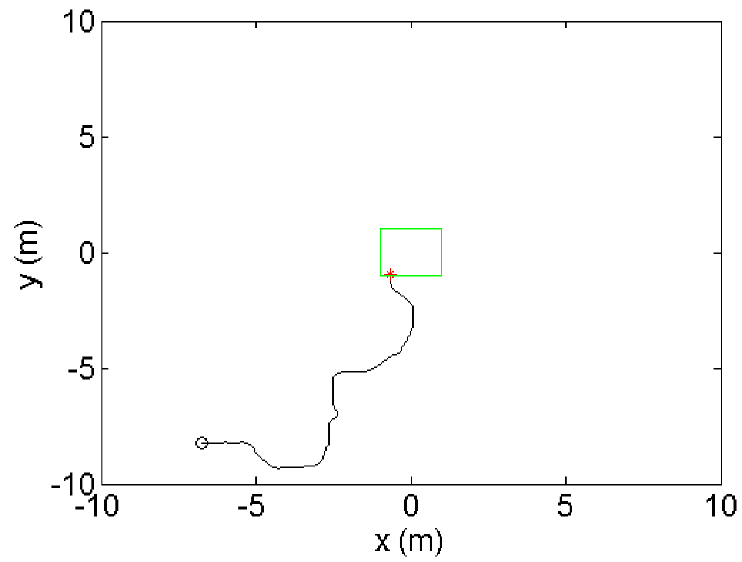

Figure 3.15: Robot Simulation - Quadrant 3

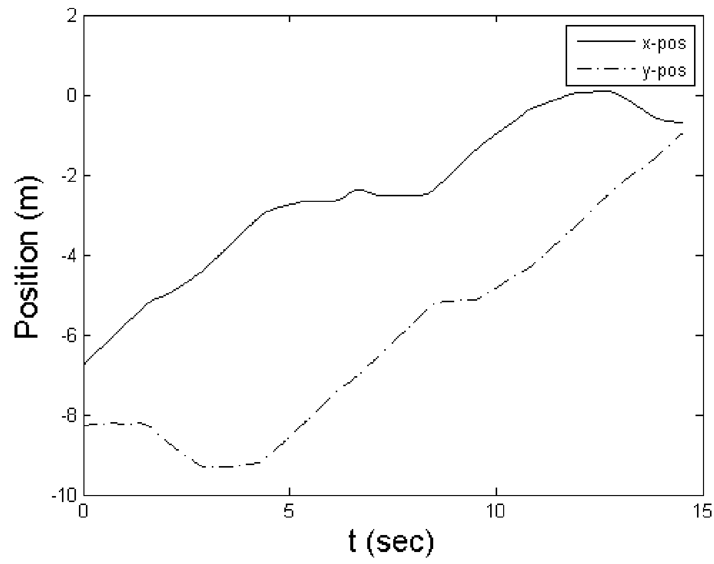


Figure 3.16: State History of Robot - Quadrant 3

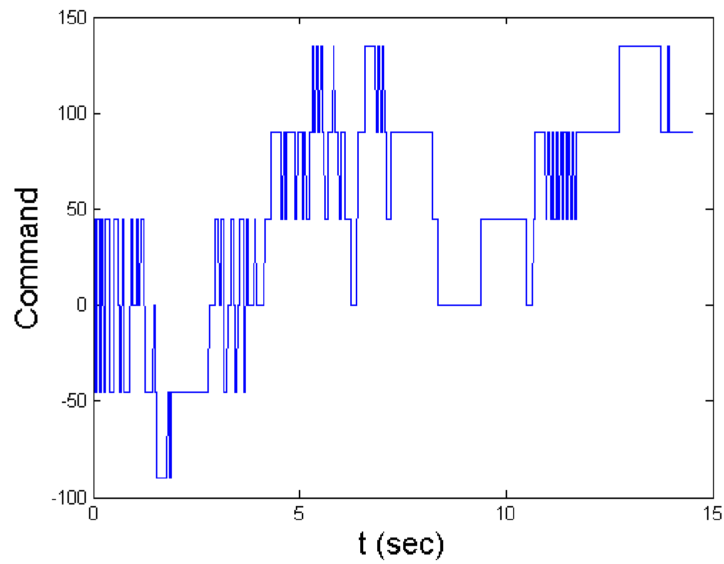


Figure 3.17: Command History of Robot - Quadrant 3

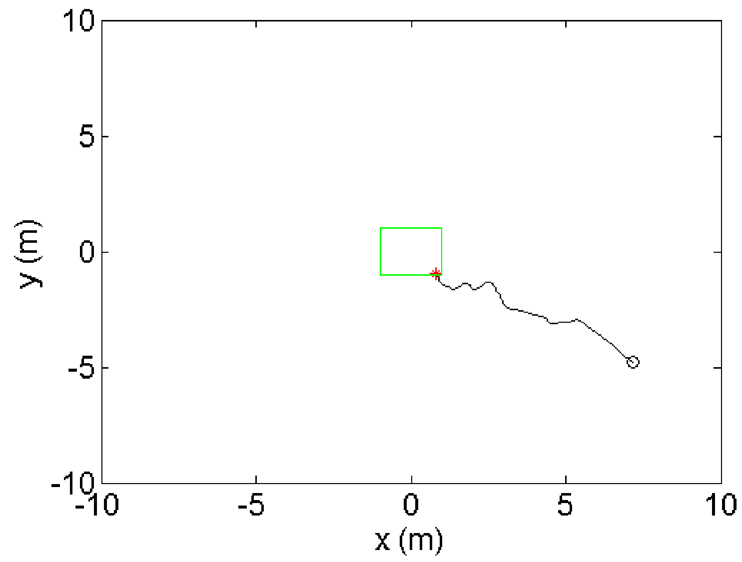


Figure 3.18: Robot Simulation - Quadrant 4

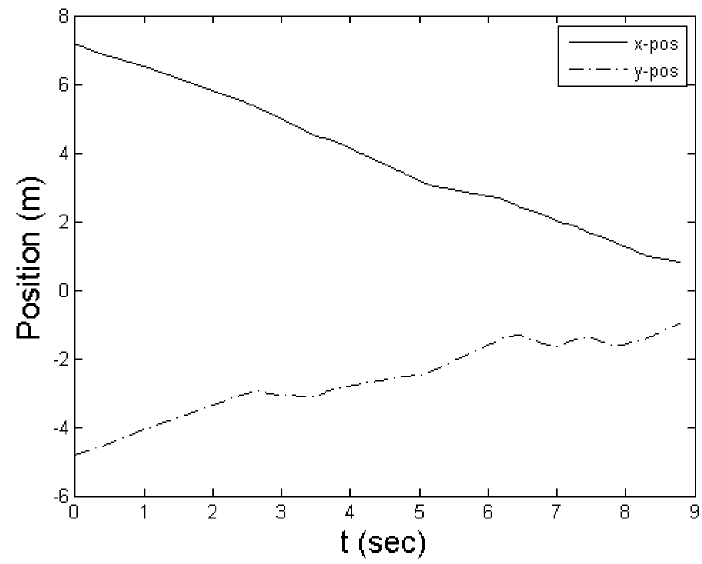


Figure 3.19: State History of Robot - Quadrant 4

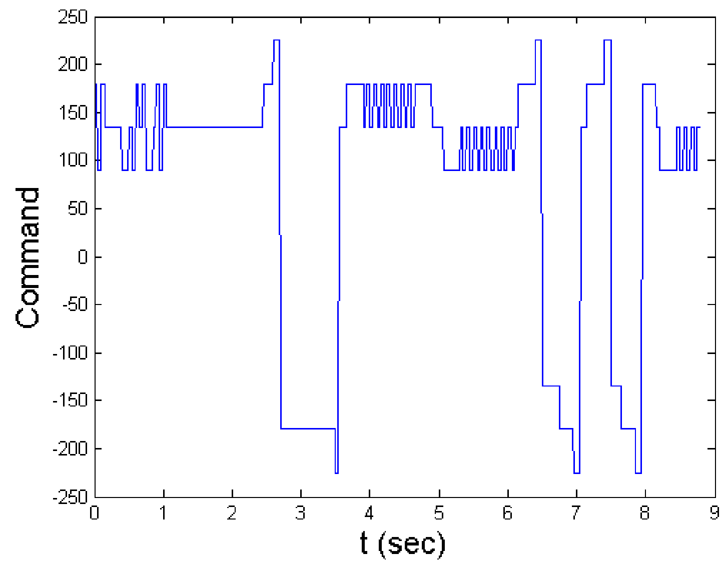


Figure 3.20: Command History of Robot - Quadrant 4

Figures 3.9-3.20 demonstrate that for the value of $T = 0.07$ sec chosen by the Sampled-Data $Q$-learning algorithm, the learner was able to control the robot to reach the goal from all around the environment, and it was able to do so with a success rate of 85%. This shows that the SDQL algorithm was able to successfully learn a non-minimal sampling rate that allows for successful control to the goal.

To compare the results of SDQL to those of $Q$-learning, this rotating robot example was further explored. The basic Watkins' $Q$-learning algorithm was used to learn a control policy for navigating the robot to the goal of $[x, y] = [0, 0]$ with a sampling rate of 0.05 sec. After 10,000 learning episodes, this action value function was then exploited using Monte Carlo simulations. For the Monte Carlo simulations, the environment was first sampled at $T = 0.05$ sec to determine the accuracy of the policy over the sampling rate used. In this case, the robot was able to reach the goal with a success rate of 82%. This sample time was then changed to $T = 0.08$ sec to compare to what SDQL determined was the best rate for this problem. With this change in sampling rate, the accuracy of the action value function fell to 61% as compared to the SDQL accuracy of 85% for $T = 0.08$ sec. This shows that the SDQL algorithm is capable of determining the best sample time and providing good control for that sample time while a basic $Q$-learning approach suffers from lack of accounting for time.

Using RL to determine this sample time and control policy is indeed possible with this algorithm, but it is also of interest to determine an approximation of the agent's dynamics once the sample time is set. Using a similar RL framework to learn agent dynamics approximations is explored in detail in the next chapter.

# 4. DYNAMICS APPROXIMATION LEARNING*

When using $Q$-learning to determine a control policy for dynamical systems, the benefit of not needing to have a model of the dynamics can lead to the user neglecting the dynamics entirely. Learning a control policy for these systems is necessary, but often it is desired to also determine some approximation of the dynamics. This is especially important when dealing with the control of heterogeneous multiagent systems since coordinating the agents requires knowing how the individual agents respond differently in time to similar action inputs [25].*

The fundamental question that is being explored in this chapter is whether or not a method of identifying a system's dynamics can be determined by beginning with a Reinforcement Learning framework. The objective is not necessarily to determine a method of system identification that is better than other current methods, but rather to see if the power of the RL methodology can be utilized to determine system dynamics approximations so it can be incorporated easily into algorithms that already use RL. Ease of implementation for determining the dynamics of a system where episodic learning is already in place is being sought.

Reinforcement Learning techniques are capable of determining the value associated with taking a particular control action given the current state, or similarly the value of being in a particular state. The goal of the investigation in this chapter is to determine if this value function iteration methodology can be used to determine time dynamics information by finding the parameter approximation that maximizes a value function for the current approximation. Determining a simple model of the

*Part of the material reported in this chapter is reproduced with permission of John Wiley & Sons, Inc. from *Reinforcement Learning and Approximate Dynamic Programming for Feedback Control*, eds. Frank L. Lewis and Derong Liu, 2012, John Wiley & Sons, Inc., Hoboken, NJ.
Copyright 2013 by The Institute of Electrical and Electronics Engineers, Inc.

dynamics can be beneficial for many applications, such as model-based control laws, determination of optimal sampling to avoid aliasing, and time-to-goal evaluations of agents.

Evaluating time-to-goal requirements for individual agents is of interest here since it is what will be used by the supervisor in the hierarchical multiagent systems discussed in Chapter 5. One way that a supervisor could evaluate choosing between different agents for a particular goal is to compare their individual action-value functions. These $Q$ functions provide information about expected reward based on current state information, and so act as a cost-to-go function. However, when comparing agents with different dynamics that have different learning experience, the values associated with expected reward-to-go do not necessarily take time-to-goal into account unless time were somehow wrapped into the reward shaping. The amount of time required to achieve the goal is not explicit in the action-value function because the state-transitions probabilities (i.e., the dynamics) are unknown in the cases discussed in this dissertation. It would be more beneficial to have some estimate of the time dynamics for each agent so that a direct comparison of time-to-goal can be considered.

In this chapter, two similar algorithms are investigated for their ability to use a Reinforcement Learning framework to determine an accurate approximation of an agent's state dynamics. The scope being considered here involves modeling individual states from systems with multiple states, and they are considered to be deterministic. The states to be modeled must also be uncoupled from the other states, so the parameters being learned can provide simple uncoupled models of the states themselves.

## 4.1 First-Order Dynamics Learning

The simplest agent dynamics to represent and learn are first-order dynamics. In a first-order dynamical system, the individual state transitions can be described by determining the time constant, $\tau$, from the stable first-order ODE shown in Equation 4.1. This makes learning an approximate time constant all that is needed to determine the time behavior of the agent.

$$\tau \frac{dx}{dt} = -x \tag{4.1}$$

This ODE can be easily solved to get a solution for the state transition from $x_1$ to $x_2$ according to the increment in time, $\Delta t$. The derivation of this equation is shown in Equations 4.2 - 4.5, with the final solution shown in Equation 4.5.

$$\int_{x_1}^{x_2} \frac{dx}{x} = -\int_{t_1}^{t_2} \frac{dt}{\tau} \tag{4.2}$$

$$\ln(x_2) - \ln(x_1) = -\frac{t_2 - t_1}{\tau} = -\frac{\Delta t}{\tau} \tag{4.3}$$

$$\ln\left(\frac{x_2}{x_1}\right) = -\frac{\Delta t}{\tau} \tag{4.4}$$

$$x_2 = x_1 e^{-\Delta t/\tau} \tag{4.5}$$

Equation 4.5 provides the solution to a first-order differential equation that can determine a state $x_2$ given the state $x_1$, $\Delta t$, and the time constant $\tau$. Determining an approximation for each state-to-state transition can be accomplished by learning a time constant associated with each state-action pair. If an action $a$ is taken in

the current state $s$ with a learned time constant $\tau$ for that state-action pair, an approximate next state $s^*$ that is measured after a sampling timestep $T$ can be determined by Equation 4.6.

$$s^* = se^{-T/\tau} \tag{4.6}$$

This development demonstrates the ability to predict a state based on a system with stable dynamics and no command, so the exponential decay is predictable. In systems with control there is a commanded state that is typically non-zero, and consideration must be made for the commanded non-zero setpoint. The dynamics can be solved for a non-zero setpoint similarly to before. The first-order differential equation for a non-zero setpoint is as shown in Equation 4.7, where $x_c$ is the commanded non-zero setpoint.

$$\tau \frac{dx}{dt} = x_c - x \tag{4.7}$$

This can be rearranged and solved in a parallel manner to before by Equations 4.8-4.12.

$$\int_{x_1}^{x_2} \frac{dx}{x - x_c} = -\int_{t_1}^{t_2} \frac{dt}{\tau} \tag{4.8}$$

$$\ln(x_2 - x_c) - \ln(x_1 - x_c) = -\frac{t_2 - t_1}{\tau} = -\frac{\Delta t}{\tau} \tag{4.9}$$

$$\ln\left(\frac{x_2 - x_c}{x_1 - x_c}\right) = -\frac{\Delta t}{\tau} \tag{4.10}$$

$$x_2 = (x_1 - x_c)e^{-\Delta t/\tau} + x_c \quad (4.11)$$

$$x_2 = x_1 e^{-\Delta t/\tau} + (1 - e^{-\Delta t/\tau})x_c \quad (4.12)$$

The solution provided in Equation 4.12 demonstrates that the original zero-point solution in Equation 4.5 can be augmented for non-zero setpoint by the addition of the second term with $x_c$. It can easily be seen how this is translated into the variables used in RL for state transitions by Equation 4.13, where the commanded state is the variable $s_c$.

$$s^* = se^{-T/\tau} + (1 - e^{-T/\tau})s_c \quad (4.13)$$

Equation 4.13 can be used to approximate a state transition for a single 1-dimensional state. If more than one independent state dimension is to be approximated, multiple time constants would be needed. For instance, if this method were applied to a robot traversing a 2-D space with independent first-order dynamics in forward translation and rotation, one might want to know the state transition dynamics of both $\tau_{forward}$ and $\tau_{rotate}$ independently as they would most likely have different dynamics. Time constants for each state-action pair would be determined based on whether the robot were moving forward or rotating.

With the ability to predict the state after the sample time period $T$ using the current approximation of the time constant, $\tau$, a reward function can be created. The reward function used here is based on the error between the predicted next state, $s^*$, and the actual next state, $s'$. To maximize the reward, the error between the predicted and observed next state must be minimized. This can be accomplished by any number of functions involving the error. The most obvious choices would be to

use either the inverse of the error or the negative of the error function. Because the inverse can lead to singularity problems, the negative is only considered here. The $L_2$-norm was chosen as the error reward function because it will always produce a non-negative value, making the opposite always non-positive. This reward function is shown in Equation 4.14.

$$r(s', s^*) = -\|s' - s^*\|_2 = \sqrt{(s' - s^*)^{\mathrm{T}}(s' - s^*)} \tag{4.14}$$

Learning an approximate time constant can be achieved using this reward structure by formulating an update similar to that done in the SDQL algorithm. This new algorithm, called First-Order Dynamics Learning (FODL), is shown in Algorithm 4.1 [25]. The The implementation of this algorithm is shown in the block diagram of Figure 4.1.

In Algorithm 4.1, the global first-order dynamics value function used for determining $\tau$ is denoted $V_1^\pi$ while the local value function for each episode is denoted $V_1$. The local value function is initialized to 0 at the beginning of each episode, and the average value for each time constant is used to update the global value function at the end of each episode. This is done to eliminate bias due to the number of times a particular time constant is visited per episode. Since the chosen reward function always produces a negative reward, choosing the correct value too often in an episode can result in artificially deflating the value. Using the average value over an episode helps to reduce that effect.

The FODL algorithm shown in Algorithm 4.1 represents the utilization of this dynamics learning process after having already determined the control policy and sample time using the SDQL algorithm in Algorithm 3.1. The FODL algorithm can also be combined with SDQL to learn all information at the same time. This full

**Algorithm 4.1** First-Order Dynamics Learning (FODL)

- Determine $T$ and $Q(\tilde{s},a)$ for system (e.g., Sampled-Data $Q$-learning)
- Initialize $V_1^{\pi}(\tau)$ arbitrarily
- Repeat for each episode:
  - Initialize $s$, append $T$ to $\tilde{s}$
  - Initialize $V_1(\tau)$ to 0
  - Repeat for each sample timestep:
    * Choose $a$ from $\tilde{s}$ using greedy policy derived with $Q(\tilde{s},a)$
    * Choose $\tau$ using policy derived from $V_1^{\pi}(\tau)$ (e.g., $\varepsilon$-Greedy)
    * Predict next state, $\tilde{s}^*$, with $\tilde{s}$, $a$, and $\tau$ using first-order approximations
    * Take action $a$, observe actual next state, $\tilde{s}'$
    * Observe $r$ shaped from observed $\tilde{s}'$ and predicted $\tilde{s}^*$
    * $V_1(\tau) \leftarrow V_1(\tau) + \alpha$ [ $r$ - $V_1(\tau)$ ]
    * $\tilde{s} \leftarrow \tilde{s}'$
  - Until $\tilde{s}$ is terminal
  - $V_1^{\pi}(\tau) \leftarrow V_1^{\pi}(\tau) + V_1(\tau)/visits(\tau)$

algorithm is shown in Algorithm 4.2.

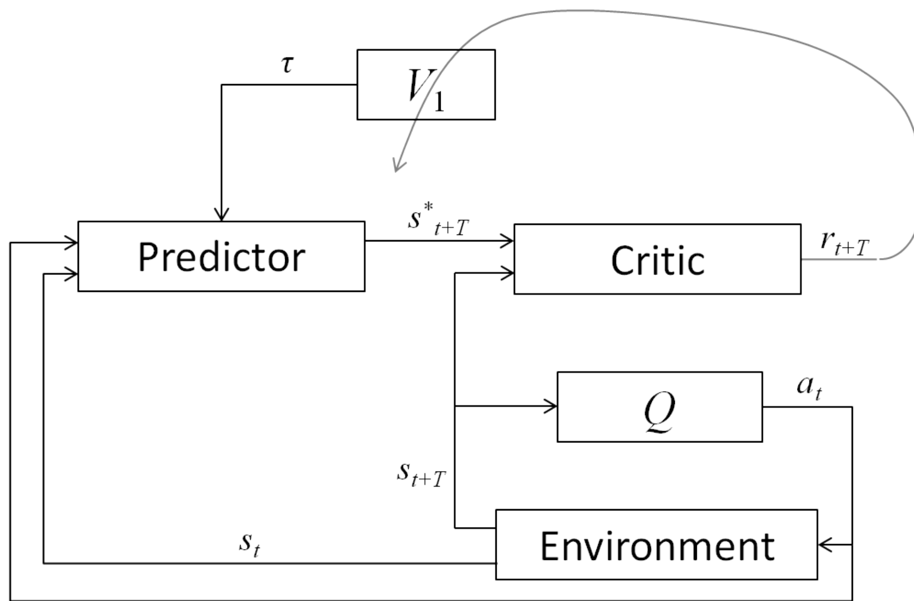


Figure 4.1: FODL Diagram

**Algorithm 4.2** FODL with SDQL

- Initialize $Q(\tilde{s},a)$ arbitrarily
- Initialize $V_T(T)$ arbitrarily
- Initialize $V_1^\pi(\tau)$ arbitrarily
- Repeat for each episode:
  - Choose $T$ using policy derived from $V_T(T)$ (e.g., $\varepsilon$-Greedy)
  - Initialize $R = 0$
  - Initialize $s$, initialize $\tilde{s}$ by appending $T$ to $s$
  - Initialize $V_1(\tau)$ to 0
  - Repeat for each timestep:
    - * Choose $a$ from $\tilde{s}$ using policy derived from $Q(\tilde{s},a)$ (e.g., $\varepsilon$-Greedy)
    - * Choose $\tau$ using policy derived from $V_1^\pi(\tau)$ (e.g., $\varepsilon$-Greedy)
    - * Predict next state, $\tilde{s}^*$, with $\tilde{s}$, $a$, and $\tau$ using first-order approximations
    - * Take action $a$, observe $r$, $\tilde{s}'$
    - * Observe $r_{dyn}$ shaped from observed $\tilde{s}'$ and predicted $\tilde{s}^*$
    - * $V_1(\tau) \leftarrow V_1(\tau) + \alpha$ [ $r_{dyn}$ - $V_1(\tau)$ ]
    - * $Q(\tilde{s},a) \leftarrow Q(\tilde{s},a) + \alpha$ [ $r + \gamma \max_{a'} Q(\tilde{s}',a')$ - $Q(\tilde{s},a)$ ]
    - * $\tilde{s} \leftarrow \tilde{s}'$
  - Until $\tilde{s}$ is terminal
  - $R$ = average($r$)
  - $V_T(T) \leftarrow V_T(T) + \alpha$ [ $R$ - $V_T(T)$ ]
  - $V_1^\pi(\tau) \leftarrow V_1^\pi(\tau) + V_1(\tau)/visits(\tau)$

## 4.2 Second-Order Dynamics Learning

The algorithm for first-order dynamics can be extended to learning approximations of second-order system dynamics. Whereas in the first-order case a systems' dynamics can be described by one parameter (time constant), a second-order system requires learning two parameters (natural frequency and damping ratio). For a second-order system, determining the next state after $T$ is not as simple as for the first-order case. The governing ODE for a system with second-order dynamics with state $s$ and a step command of $s_c$ is as shown in Equation 4.15.

$$\frac{d^2 s}{dt^2} + 2\zeta\omega_n \frac{ds}{dt} + \omega_n^2 s = \omega_n^2 s_c \tag{4.15}$$

This system can be parameterized in state-space form by making new variable declarations. If the state-space is parameterized according to Equations 4.16 and 4.17, then the governing equations of motion become as shown in Equations 4.18 and 4.19.

$$s_1 = s \tag{4.16}$$

$$s_2 = \dot{s} \tag{4.17}$$

$$\dot{s}_1 = s_2 \tag{4.18}$$

$$\dot{s}_2 = -2\zeta\omega_n s_2 - \omega_n^2 (s_1 - s_c) \tag{4.19}$$

Alternatively, the dynamics shown in Equations 4.18 and 4.19 can be put into state-space form, such as shown in Equation 4.20. The state-space form of these equations can be seen in Equation 4.21.

$$\dot{x} = Ax + Bu \tag{4.20}$$

$$\begin{bmatrix} \dot{s}_1 \\ \dot{s}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\omega_n^2 & -2\zeta\omega_n \end{bmatrix} \begin{bmatrix} s_1 \\ s_2 \end{bmatrix} + \begin{bmatrix} 0 \\ \omega_n^2 \end{bmatrix} s_c \tag{4.21}$$

$$x = s \tag{4.22}$$

$$u = s_c \tag{4.23}$$

$$A = \begin{bmatrix} 0 & 1 \\ -\omega_n^2 & -2\zeta\omega_n \end{bmatrix} \tag{4.24}$$

$$B = \begin{bmatrix} 0 \\ \omega_n^2 \end{bmatrix} \tag{4.25}$$

To determine the solution to the second-order system, the characteristic equation must be found. This can be done by taking the Laplace transform of the second-order dynamics defined in Equation 4.15. This is demonstrated in Equations 4.26-4.32.

$$\mathcal{L}\{\ddot{s} + 2\zeta\omega_n\dot{s} + \omega_n^2 s = \omega_n^2 s_c\} \tag{4.26}$$

$$\mathcal{L}\{\ddot{s} + 2\zeta\omega_n\dot{s} + \omega_n^2 s\} = \mathcal{L}\{\omega_n^2 s_c\} \tag{4.27}$$

The Laplace transform of derivatives are as shown in Equations 4.28 and 4.29.

$$\mathcal{L}\{\dot{s}\} = \lambda S(\lambda) - s(0) \tag{4.28}$$

$$\mathcal{L}\{\ddot{s}\} = \lambda^2 S(\lambda) - \lambda s(0) - \dot{s}(0) \tag{4.29}$$

The derivatives can be substituted into Equation 4.27 to continue the derivation with the Laplace transformed variables.

$$\lambda^2 S - \lambda s(0) - \dot{s}(0) + 2\zeta\omega_n(\lambda S - s(0)) + \omega_n^2 S = \omega_n^2 S_c \tag{4.30}$$

$$S(\lambda^2 + 2\zeta\omega_n\lambda + \omega_n^2) = \dot{s}(0) + (\lambda + 2\zeta\omega_n)s(0) + \omega_n^2 S_c \tag{4.31}$$

$$S(\lambda) = \frac{\dot{s}(0) + (\lambda + 2\zeta\omega_n)s(0) + \omega_n^2 S_c(\lambda)}{\lambda^2 + 2\zeta\omega_n\lambda + \omega_n^2} \tag{4.32}$$

The Laplace transform of the commanded state, $s_c$ is as shown in Equation 4.33, providing the frequency domain solution to the second-order differential equation in Equation 4.34.

$$\mathcal{L}\left\{s_c\right\} = S_c(\lambda) = \frac{s_c}{\lambda} \tag{4.33}$$

$$S(\lambda) = \frac{\dot{s}(0) + (\lambda + 2\zeta\omega_n)s(0)}{\lambda^2 + 2\zeta\omega_n\lambda + \omega_n^2} + \frac{\omega_n^2 s_c}{\lambda(\lambda^2 + 2\zeta\omega_n\lambda + \omega_n^2)} \tag{4.34}$$

The Laplace transformed state described by Equation 4.34 can be used to find the poles of the system. The characteristic equation is the denominator of this function as shown in Equation 4.35. This equation can be solved as shown in Equations 4.35-4.38 to find the values of the poles as a function of $\omega_n$ and $\zeta$ in Equation 4.38.

$$\lambda^2 + 2\zeta\omega_n\lambda + \omega_n^2 = 0 \tag{4.35}$$

$$\lambda = \frac{-2\zeta\omega_n \pm \sqrt{4\zeta^2\omega_n^2 - 4\omega_n^2}}{2} \tag{4.36}$$

$$\lambda = \frac{-2\zeta\omega_n \pm 2\omega_n\sqrt{\zeta^2 - 1}}{2} \tag{4.37}$$

$$\lambda = -\zeta\omega_n \pm \mathrm{j}\omega_n\sqrt{1 - \zeta^2} \tag{4.38}$$

Upon examining Equation 4.38, it can be seen that the value of the damping ratios influence the dynamics such that there are 4 main solutions. If $\zeta = 0$ then there is no damping, if $0 < \zeta < 1$ then the system is underdamped, if $\zeta = 1$ it is critically damped, and if $\zeta > 1$ then the system is overdamped. For each of these scenarios, the poles are as listed in Equation 4.39.

$$\lambda = \begin{cases} \pm\mathrm{j}\omega_n & \text{if } \zeta = 0 \\ -\zeta\omega_n \pm \mathrm{j}\omega_n\sqrt{1-\zeta^2} & \text{if } 0 < \zeta < 1 \\ -\omega_n & \text{if } \zeta = 1 \\ -\zeta\omega_n \pm \omega_n\sqrt{\zeta^2 - 1} & \text{if } \zeta > 1 \end{cases} \tag{4.39}$$

The plot of these poles in the complex plane shows the difference these solutions have on the behavior of the system. These plots are shown in Figures 4.2-4.5.

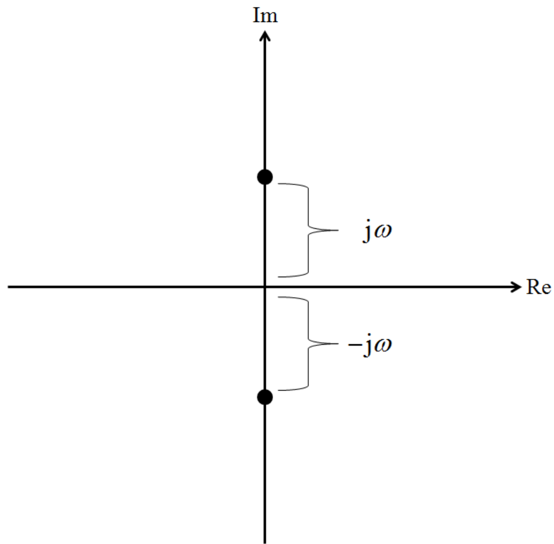

Figure 4.2: Complex Plane - $\zeta = 0$

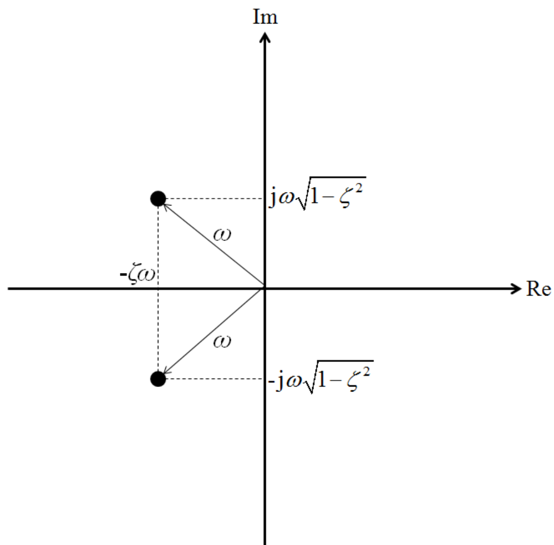

Figure 4.3: Complex Plane - $0 < \zeta < 1$

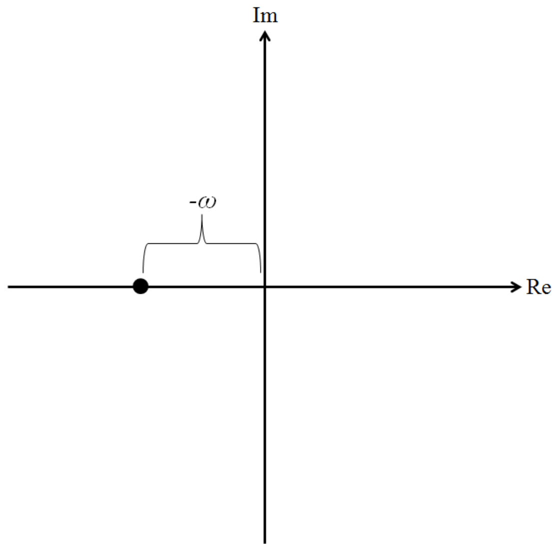


Figure 4.4: Complex Plane - $\zeta = 1$

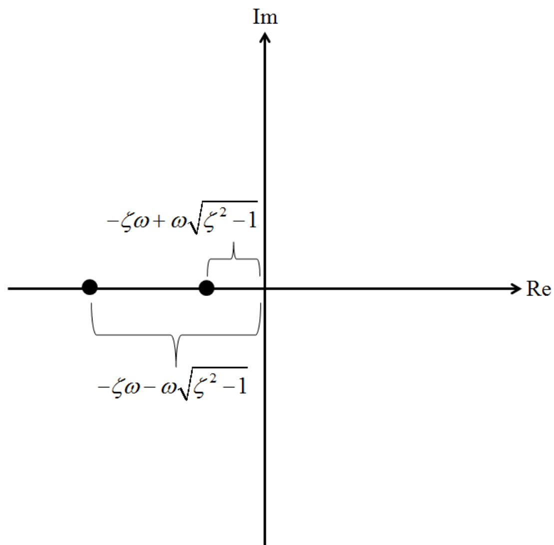


Figure 4.5: Complex Plane - $\zeta > 1$

The solution for the state shown in Equation 4.34 is the frequency domain solution, so finding the time domain solution for the state requires an inverse Laplace transform. Using the inverse Laplace transform to find the time domain solution requires partial fraction expansion over each of the 4 cases of $\zeta$, and is a rather involved process. The full process for obtaining the solution by partial fraction expansion and inverse Laplace transformation can be found in Appendix B. For the case of $\zeta = 0$, the solution is as shown in Equation 4.40.

$$s(t) = s_c + \frac{\dot{s}(0)}{\omega_n}\sin(\omega_n t) + (s(0) - s_c)\cos(\omega_n t) \tag{4.40}$$

For the case of $0 < \zeta < 1$, the solution is as shown in Equation 4.41.

$$\begin{aligned} s(t) = s_c + \frac{\zeta}{\sqrt{1-\zeta^2}} e^{-\zeta\omega_n t}\left(s(0) - s_c + \frac{\dot{s}(0)}{\zeta\omega_n}\right)\sin(\omega_n t\sqrt{1-\zeta^2}) \\ + e^{-\zeta\omega_n t}(s(0) - s_c)\cos(\omega_n t\sqrt{1-\zeta^2}) \end{aligned} \tag{4.41}$$

For the case of $\zeta = 1$, the solution is as shown in Equation 4.42.

$$s(t) = s_c + (s(0) - s_c)e^{-\omega_n t} + (\dot{s}(0) + \omega_n s(0) - \omega_n s_c)te^{-\omega_n t} \tag{4.42}$$

And for the case of $\zeta > 1$, the solution is as shown in Equation 4.43.

$$s(t) = s_c + \left( \frac{\dot{s}(0) + (2\zeta\omega_n - z_1)s(0) - s_c z_2}{z_2 - z_1} \right) e^{-z_1 t} \tag{4.43}$$
$$- \left( \frac{\dot{s}(0) + (2\zeta\omega_n + z_2)s(0) - s_c z_1}{z_2 - z_1} \right) e^{-z_2 t}$$

where

$$z_1 = \omega_n(\zeta - \sqrt{\zeta^2 - 1})$$
$$z_2 = \omega_n(\zeta + \sqrt{\zeta^2 - 1})$$

With the predictive state equations shown above, the algorithm for Second-Order Dynamics Learning is as shown in Algorithm 4.3. The SODL algorithm is very similar to the FODL algorithm, with the difference being in the state-prediction equations and the parameters being determined. Here, the global second-order dynamics value function is denoted $V_2^\pi$ and the local value function is $V_2$. The implementation of this algorithm is shown in the block diagram of Figure 4.6. The comparison of this diagram to Figure 4.1 shows the similarities between these two algorithms.

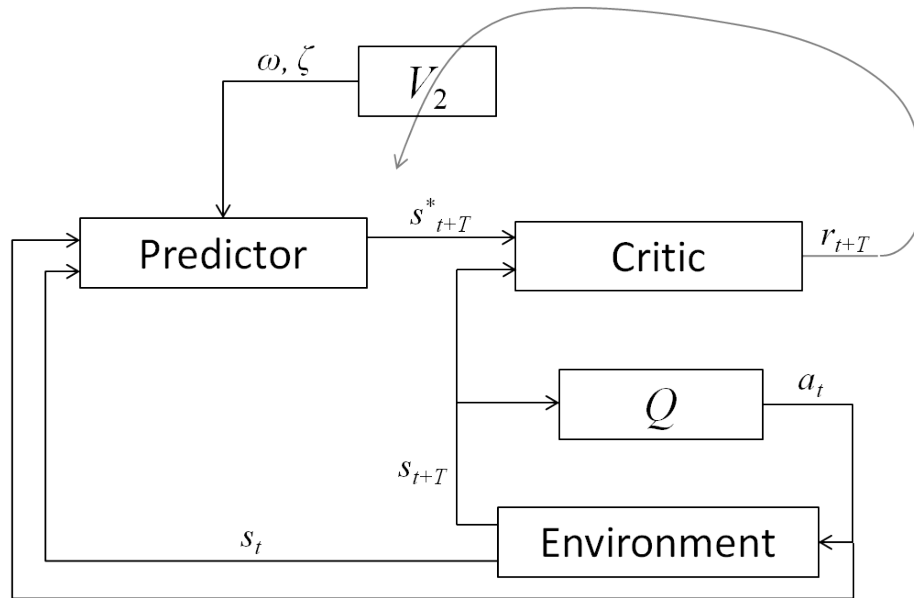

Figure 4.6: SODL Diagram

**Algorithm 4.3** Second-Order Dynamics Learning (SODL)

- Determine $T$ and $Q(\tilde{s},a)$ for system (e.g., Sampled-Data $Q$-learning)
- Initialize $V_2^{\pi}(\omega_n, \zeta)$ arbitrarily
- Repeat for each episode:
  - Initialize $s$, append $T$ to $\tilde{s}$
  - Initialize $V_2(\omega_n, \zeta)$ to 0
  - Repeat for each sample timestep:
    - Choose $a$ from $\tilde{s}$ using greedy policy derived with $Q(\tilde{s},a)$
    - Choose $\omega_n$ and $\zeta$ using policy derived from $V_2^{\pi}(\omega_n, \zeta)$ (e.g., $\varepsilon$-Greedy)
    - Predict next state, $\tilde{s}^*$, with $\tilde{s}$, $a$, $\omega_n$, and $\zeta$ using second-order approximations
    - Take action $a$, observe actual next state, $\tilde{s}'$
    - Observe $r$ shaped from observed $\tilde{s}'$ and predicted $\tilde{s}^*$
    - $V_2(\omega_n, \zeta) \leftarrow V_2(\omega_n, \zeta) + \alpha$ [ $r$ - $V_2(\omega_n, \zeta)$ ]
    - $\tilde{s} \leftarrow \tilde{s}'$
  - Until $\tilde{s}$ is terminal
  - $V_2^{\pi}(\omega_n, \zeta) \leftarrow V_2^{\pi}(\omega_n, \zeta) + V_2(\omega_n, \zeta)/visits(\omega_n, \zeta)$

Just as was shown with FODL, it can be seen that the SODL algorithm as written in Algorithm 4.3 is used after learning the control policy and sample time by SDQL. However, just as before with FODL, the SODL and SDQL algorithms can be combined to learn all information at once. This full algorithm is displayed in Algorithm 4.4.

**Algorithm 4.4** SODL with SDQL

- Initialize $Q(\tilde{s},a)$ arbitrarily
- Initialize $V_T(T)$ arbitrarily
- Initialize $V_2^\pi(\omega_n, \zeta)$ arbitrarily
- Repeat for each episode:
  - Choose $T$ using policy derived from $V_T(T)$ (e.g., $\varepsilon$-Greedy)
  - Initialize $R = 0$
  - Initialize $s$, initialize $\tilde{s}$ by appending $T$ to $s$
  - Initialize $V_2(\omega_n, \zeta)$ to 0
  - Repeat for each timestep:
    * Choose $a$ from $\tilde{s}$ using policy derived from $Q(\tilde{s},a)$ (e.g., $\varepsilon$-Greedy)
    * Choose $\omega_n$ and $\zeta$ using policy derived from $V_2^\pi(\omega_n, \zeta)$ (e.g., $\varepsilon$-Greedy)
    * Predict next state, $\tilde{s}^*$, with $\tilde{s}$, $a$, $\omega_n$, and $\zeta$ using second-order approximations
    * Take action $a$, observe $r$, $\tilde{s}'$
    * Observe $r_{dyn}$ shaped from observed $\tilde{s}'$ and predicted $\tilde{s}^*$
    * $V_2(\omega_n, \zeta) \leftarrow V_2(\omega_n, \zeta) + \alpha$ [ $r_{dyn}$ - $V_2(\omega_n, \zeta)$ ]
    * $Q(\tilde{s},a) \leftarrow Q(\tilde{s},a) + \alpha$ [ $r + \gamma \max_{a'} Q(\tilde{s}',a')$ - $Q(\tilde{s},a)$ ]
    * $\tilde{s} \leftarrow \tilde{s}'$
  - Until $\tilde{s}$ is terminal
  - $R$ = average($r$)
  - $V_T(T) \leftarrow V_T(T) + \alpha$ [ $R$ - $V_T(T)$ ]
  - $V_2^\pi(\omega_n, \zeta) \leftarrow V_2^\pi(\omega_n, \zeta) + V_2(\omega_n, \zeta)/visits(\omega_n, \zeta)$

## 4.3 Dynamics Learning Examples

The learning of approximate dynamics modeling parameters can be very beneficial for problems involving coordination of multiple heterogeneous agents. Learning the time constants for a first-order system or the natural frequency and damping ratio for a second-order system removes uncertainty about the time response behavior of individual agents. Figure 4.7 shows an example of a multiagent system that requires coordination to maximize goal achievement in minimal time. In this system there are 4 agents, although only 3 are pictured. The 3 pictured agents each have differing dynamics that cause them to transition from state-to-state in different amounts of time. The fourth, unseen agent, is the high-level agent that gives commands to the 3 low-level agents.

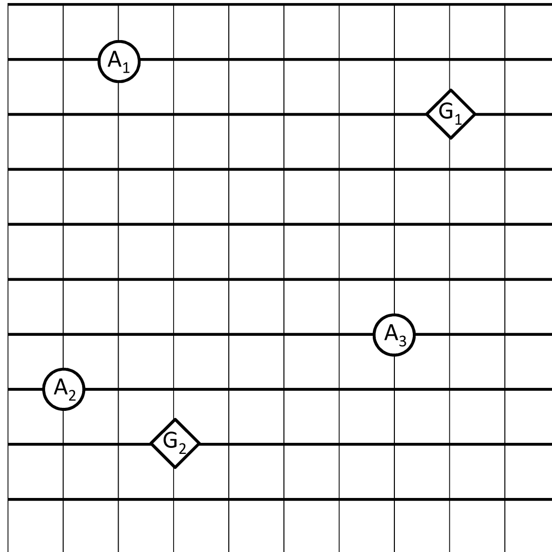


Figure 4.7: Multiagent System

In this hypothetical 2D path planning scenario, there are 2 separate goals that

each must be reached by any 2 of the 3 low-level agents. Minimizing time is considered to be of highest priority, so the high-level agent must take each low-level agent's dynamics into consideration. For instance, agent $A_2$ may be sufficiently close enough to goal $G_2$ to make the decision easy, but the relative positions of $A_1$ and $A_2$ to $G_1$ must be considered before deciding which agent to send. While it may be the simplest approach to just send both agents and see who reaches the goal first, it is considered a waste of valuable resources to send 2 agents to achieve a goal that only takes 1 agent. The decision becomes even more difficult if all 3 agents were to begin at the same state. For this scenario, let us assume that agent $A_1$ is significantly faster than $A_3$, and although $A_3$ is closer in distance it will actually take longer to achieve the goal. This may result in the set of high-level commands shown in Figure 4.8.

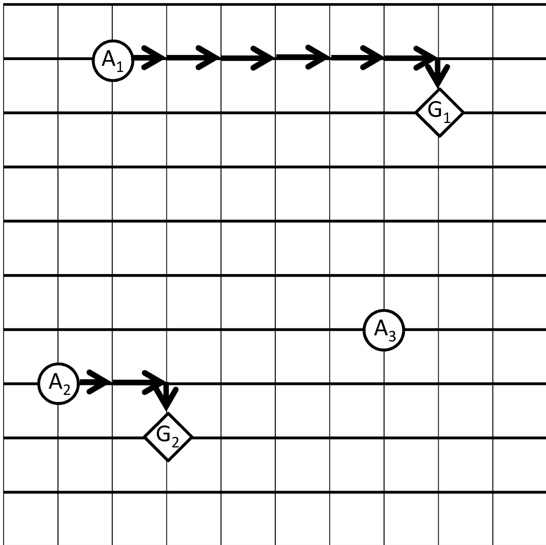


Figure 4.8: High-Level Agent Commands

In this scenario, the high-level agent gives commands to each of the low-level

agents. For this to be possible, each low-level agent must have its own control policy based on the commands given. This would simply require that each agent learn its individual optimal sample time, $T$, using Algorithm 3.1, and learn its dynamical model parameters for performing state-to-state transitions using Algorithms 4.1 and 4.3. These can be done together online using the combined algorithms shown in Algorithms 4.2 and 4.4. The high-level agent simply needs to sample each agent's positions at different intervals, according to the individual $T$ associated with each. It can then use each agent's learned $\tau$ or $\omega_n$ and $\zeta$ to make decisions regarding minimum time goal achievement. An example of how this might affect the decision making of the high-level agent from our thought experiment is shown in Figure 4.9.

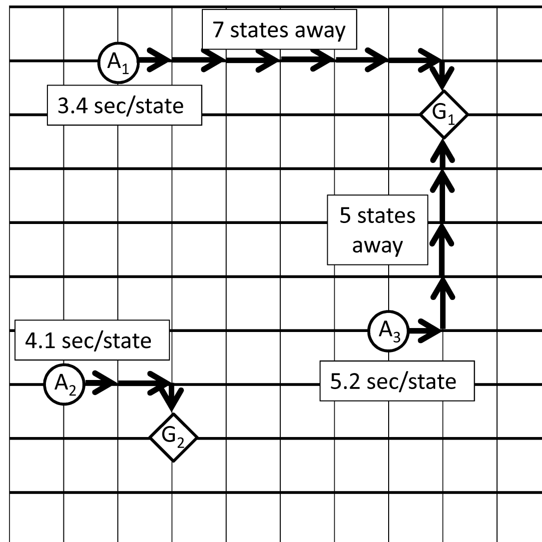


Figure 4.9: High-Level Agent Decision Making

This motivating example is a generalization of systems that need to receive high-level commands, and then use lower-level controllers to achieve those commands. This type of system can consist of any group of spacial agents with time dynamics.

Examples of this include cooperative robots, aircraft, ground vehicles, and any combination of these. The previously used example of the forward-moving robots with rotational command changes provide a general framework for simulating this type of system, and will be used to test these algorithms.

### *4.3.1 First-Order Examples*

Here, the same rotating robot from Section 3.4.2 is used to test the FODL algorithm. In this example, the robot translates forward at a constant speed while rotating to control direction of travel. The heading angle of this robot exhibits first-order dynamics, and the time constant describing these dynamics needs to be determined. The time constant $\tau$ informs the dynamics according to Equation 4.44.

$$\dot{\psi} = (\psi_c - \psi)\tau^{-1} \tag{4.44}$$

In this example, the FODL algorithm was used online with the SDQL algorithm to determine the best approximation of the time constant while also learning the best sample time and control. The time constant for this robot simulation was chosen to be $\tau = 0.2$ sec, so that was the target value that the FODL algorithm was supposed to determine. Equation 4.45 shows the proper equation of motion for the rotation of this robot given the time constant of $\tau = 0.2$ sec. Figures 4.10-4.15 show the evolution of the sampled-data value function over the course of 10,000 episodes.

$$\dot{\psi} = 5\,(\psi_c - \psi) \tag{4.45}$$

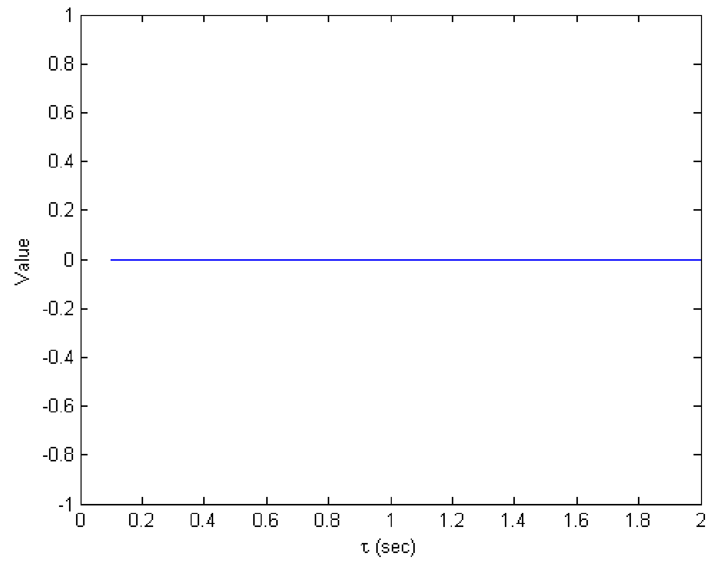


Figure 4.10: $1^{st}$ Order Value Function for $\tau = 0.2$: 0 Episodes

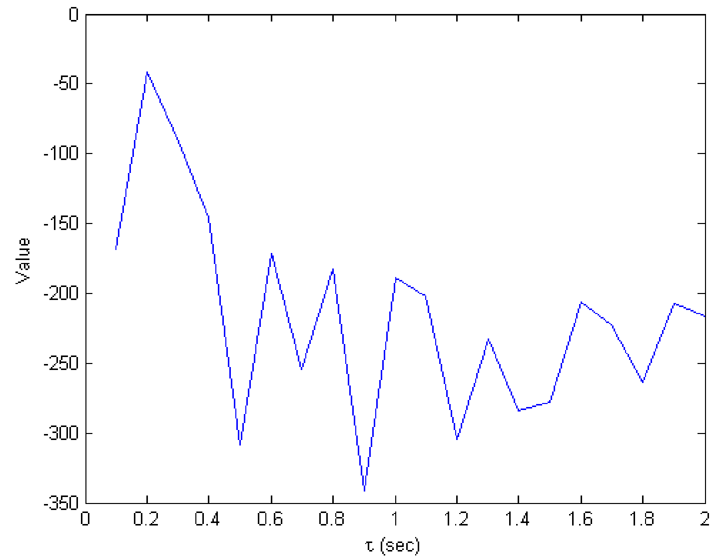


Figure 4.11: $1^{st}$ Order Value Function for $\tau = 0.2$: 200 Episodes

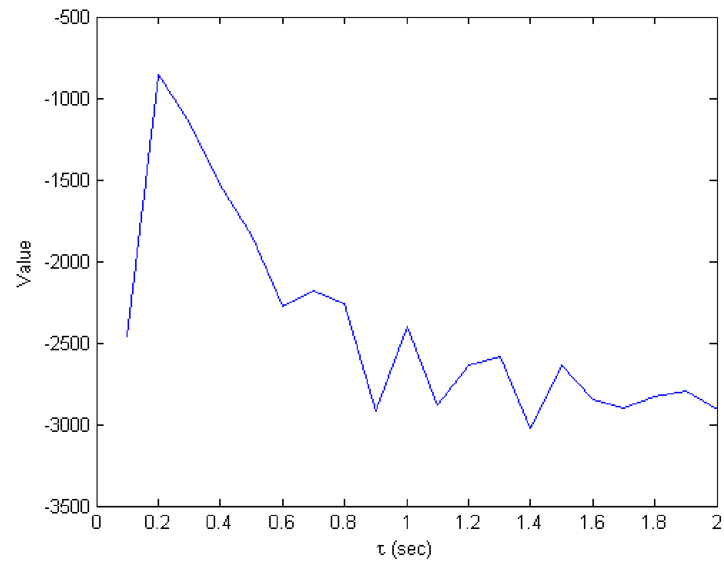


Figure 4.12: $1^{st}$ Order Value Function for $\tau = 0.2$: 1,000 Episodes

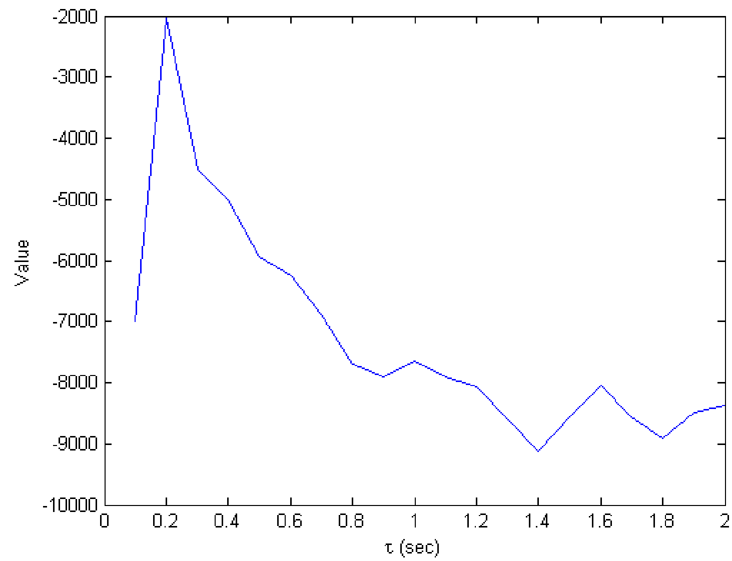


Figure 4.13: $1^{st}$ Order Value Function for $\tau = 0.2$: 5,000 Episodes

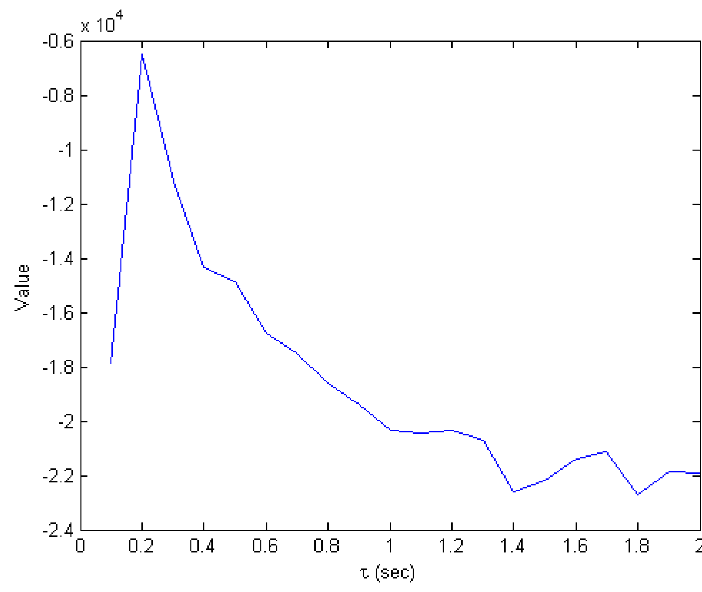


Figure 4.14: $1^{st}$ Order Value Function for $\tau = 0.2$: 10,000 Episodes

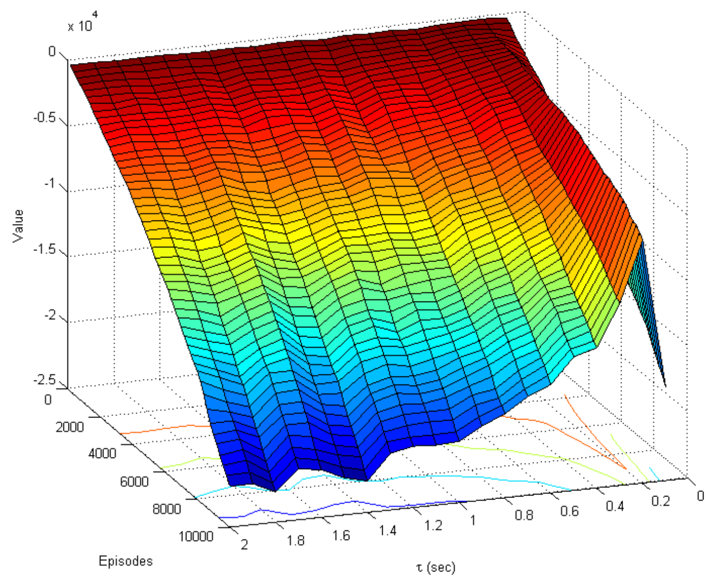


Figure 4.15: $1^{st}$ Order Value Function History for $\tau = 0.2$

As these figures show, the FODL algorithm was able to determine the correct time constant of $\tau = 0.2$ sec early in the learning process, and the continued learning reinforced that knowledge. The final values associated with the various time constants after 10,000 learning episodes are shown in Table 4.1. The mean time constant value and the variance of values over possible time constants are shown in Equations 4.46 and 4.47.

Table 4.1: FODL Robot Value Function: $\tau = 0.2$

| $\tau(sec)$ | $V_1$ |
|---|---|
| 0.1 | $-1.79 \times 10^4$ |
| 0.2 | $-0.65 \times 10^4$ |
| 0.3 | $-1.12 \times 10^4$ |
| 0.4 | $-1.44 \times 10^4$ |
| 0.5 | $-1.49 \times 10^4$ |
| 0.6 | $-1.68 \times 10^4$ |
| 0.7 | $-1.75 \times 10^4$ |
| 0.8 | $-1.86 \times 10^4$ |
| 0.9 | $-1.94 \times 10^4$ |
| 1.0 | $-2.03 \times 10^4$ |
| 1.1 | $-2.05 \times 10^4$ |
| 1.2 | $-2.03 \times 10^4$ |
| 1.3 | $-2.07 \times 10^4$ |
| 1.4 | $-2.26 \times 10^4$ |
| 1.5 | $-2.22 \times 10^4$ |
| 1.6 | $-2.14 \times 10^4$ |
| 1.7 | $-2.11 \times 10^4$ |
| 1.8 | $-2.27 \times 10^4$ |
| 1.9 | $-2.19 \times 10^4$ |
| 2.0 | $-2.19 \times 10^4$ |

$$E[V_1] = -1.85 \times 10^4 \tag{4.46}$$

$$Var[V_1] = 1.78 \times 10^7 \tag{4.47}$$

After successfully demonstrating the capability of the FODL algorithm to converge to the correct time constant for this example, the dynamics were changed so that a different time constant should be learned. Using the same rotating robot example as before, the example was executed with a new time constant of $\tau = 1$ sec. This results in the equation of motion for the robot rotation as shown in Equation 4.48.

$$\dot{\psi} = \psi_c - \psi \tag{4.48}$$

Using this new time constant, the FODL algorithm was utilized. Figures 4.16-4.21 show the convergence of the value function to the proper time constant.

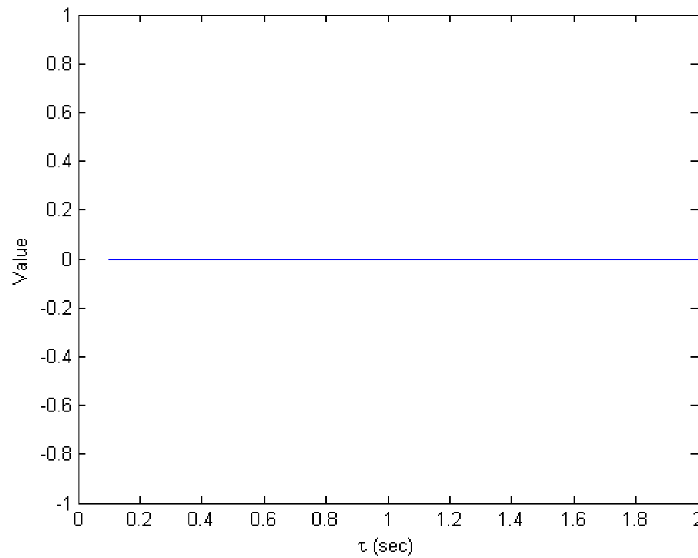


Figure 4.16: $1^{st}$ Order Value Function for $\tau = 1$: 0 Episodes

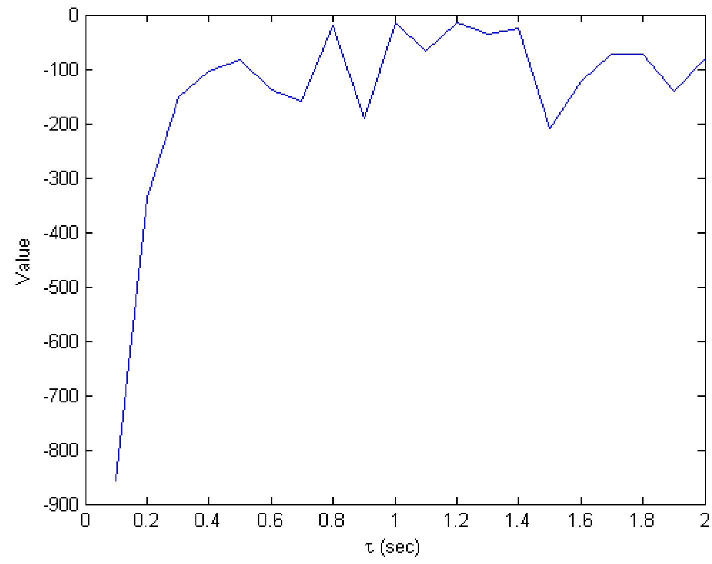


Figure 4.17: $1^{st}$ Order Value Function for $\tau = 1$: 200 Episodes

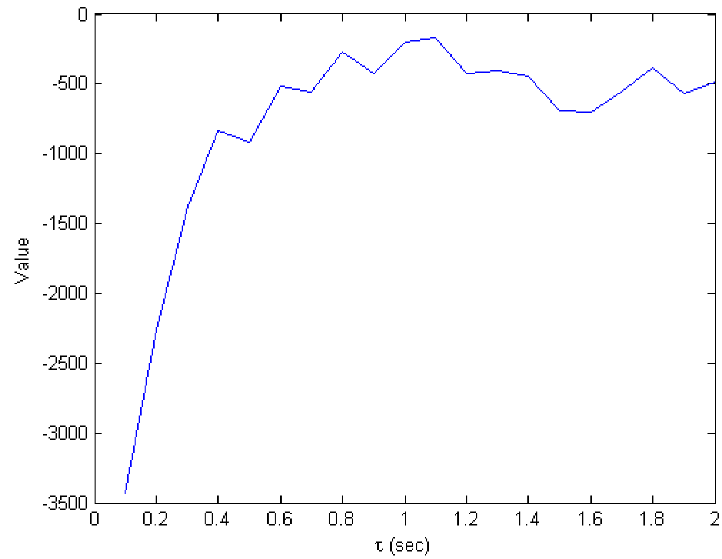


Figure 4.18: $1^{st}$ Order Value Function for $\tau = 1$: 1,000 Episodes

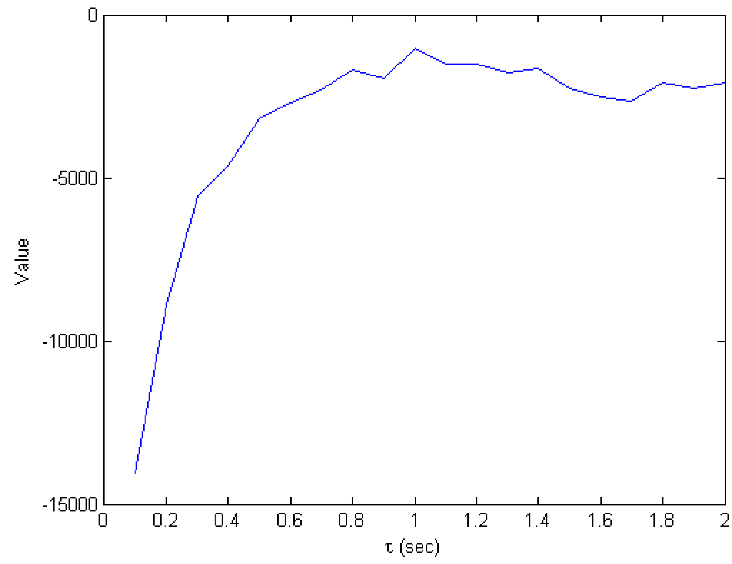

Figure 4.19: $1^{st}$ Order Value Function for $\tau = 1$: 5,000 Episodes

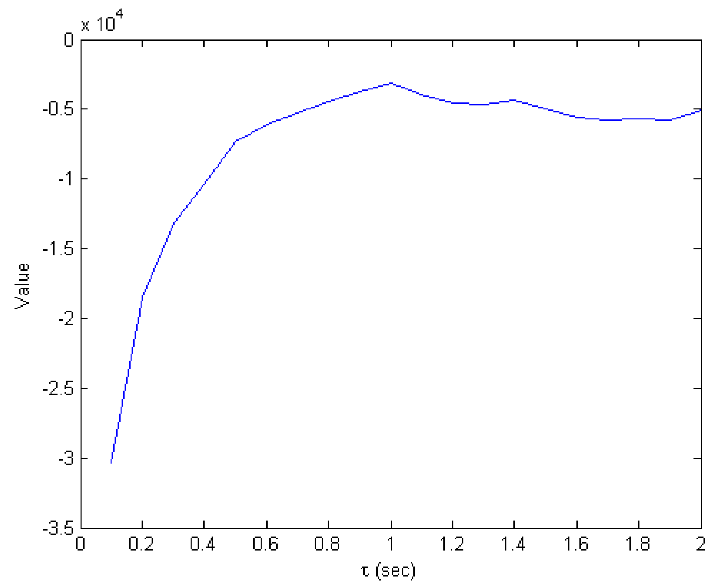

Figure 4.20: $1^{st}$ Order Value Function for $\tau = 1$: 10,000 Episodes

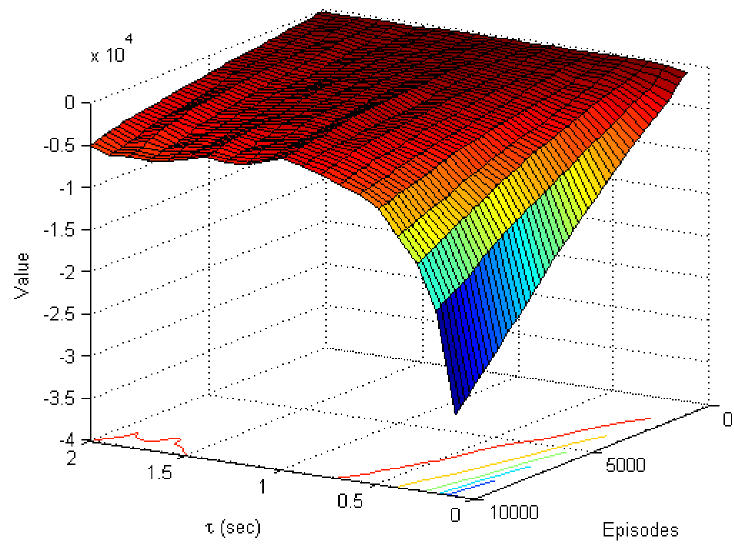


Figure 4.21: $1^{st}$ Order Value Function History for $\tau = 1$

The value convergence history shown in these figures indicates that the FODL algorithm was again successful in determining the correct time constant. In the previous case of $\tau = 0.2$, the value was prominent very early and remained so throughout the learning process. In the case of $\tau = 1$ it can be seen that the value function is much smoother and gradual, but the algorithm still converges to and maintains a maximum value associated with the correct time constant of $\tau = 1$. The final learned values for the time constants after 10,000 learning episodes are shown in Table 4.2. The mean time constant value and the variance of values over possible time constants are shown in Equations 4.49 and 4.50.

Table 4.2: FODL Robot Value Function: $\tau = 1$

| $\tau(sec)$ | $V_1$ |
|---|---|
| 0.1 | $-30.38 \times 10^3$ |
| 0.2 | $-18.48 \times 10^3$ |
| 0.3 | $-13.20 \times 10^3$ |
| 0.4 | $-10.41 \times 10^3$ |
| 0.5 | $-7.36 \times 10^3$ |
| 0.6 | $-6.04 \times 10^3$ |
| 0.7 | $-5.27 \times 10^3$ |
| 0.8 | $-4.42 \times 10^3$ |
| 0.9 | $-3.69 \times 10^3$ |
| 1.0 | $-3.15 \times 10^3$ |
| 1.1 | $-3.90 \times 10^3$ |
| 1.2 | $-4.59 \times 10^3$ |
| 1.3 | $-4.63 \times 10^3$ |
| 1.4 | $-4.33 \times 10^3$ |
| 1.5 | $-4.96 \times 10^3$ |
| 1.6 | $-5.53 \times 10^3$ |
| 1.7 | $-5.78 \times 10^3$ |
| 1.8 | $-5.65 \times 10^3$ |
| 1.9 | $-5.80 \times 10^3$ |
| 2.0 | $-5.08 \times 10^3$ |

$$E[V_1] = -7.63 \times 10^3 \tag{4.49}$$

$$Var[V_1] = 4.21 \times 10^7 \tag{4.50}$$

### *4.3.2 Second-Order Examples*

Now the rotating robot example is used to test the SODL algorithm, so the simulation must be altered to include dynamics of second-order. In this simulation, the robot still translates forward with a constant speed and rotates to control heading direction, but the heading angle exhibits second-order dynamics rather than first-order. Equation 4.51 shows the second-order differential equation driving the dynamics.

$$\ddot{\psi} = -2\zeta\omega_n\dot{\psi} + \omega_n^2(\psi_c - \psi) \tag{4.51}$$

The RL state-space for this problem can be augmented to include $\dot{\psi}$ so that there is full state feedback, but it is not necessarily needed. For the sake of learning time, the second-order cases in these simulations maintain the same state-space as before. It is seen in the simulations that for these problems it is not necessary to include the heading angle rate as convergence is still obtained with high accuracy.

For this simulation, the true natural frequency and damping ratio for the robot were set to $\omega_n = 6$ rad/sec and $\zeta = 0.8$. This provides a robot that is able to rotate its heading angle to the new command with a maximum overshoot of $M_p = 1.52\%$ and a peak time of $t_p = 0.87$ sec. This is a mild overshoot and a reasonably quick peak time. These values are determined according to Equations 4.52 and 4.53.

$$M_p = \exp\left(\frac{-\pi\zeta}{\sqrt{1-\zeta^2}}\right) * 100\% \tag{4.52}$$

$$t_p = \frac{\pi}{\omega_n \sqrt{1 - \zeta^2}} \tag{4.53}$$

With the chosen values of $\omega_n = 6$ rad/sec and $\zeta = 0.8$, the equation of motion for the heading angle becomes as shown in Equation 4.54.

$$\ddot{\psi} = -9.6\dot{\psi} + 36(\psi_c - \psi) \tag{4.54}$$

The robot is then allowed to learn using Sampled-Data $Q$-learning with the SODL algorithm active. This continues for 10,000 episodes just as before, with the SODL algorithm seeking to determine the proper natural frequency and damping ratio for the heading angle command. After 10,000 learning episodes, the SODL algorithm was successfully able to learn the correct natural frequency and damping ratio values of $\omega_n = 6$ and $\zeta = 0.8$. Figures 4.22-4.26 show the evolution of the SODL value function $V_2$ over the course of the 10,000 learning episodes completed.

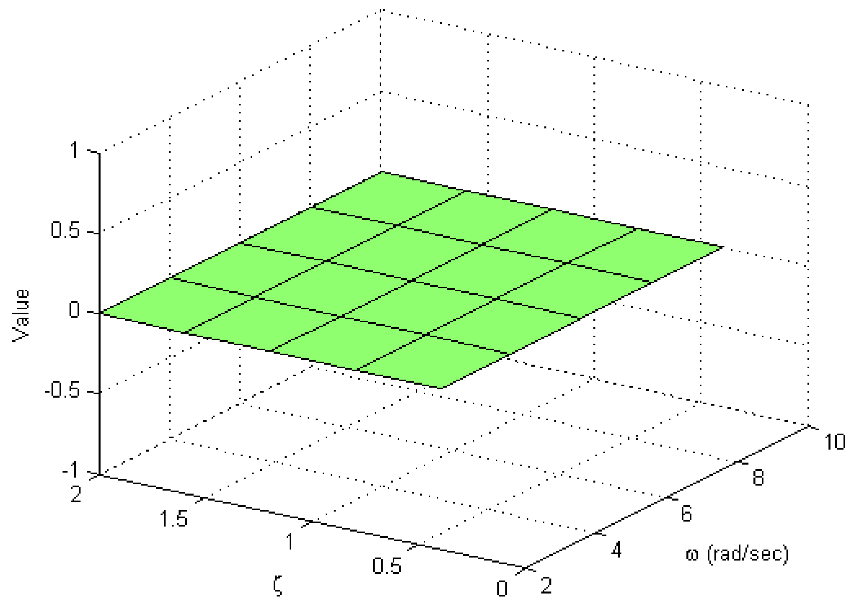


Figure 4.22: $2^{nd}$ Order Value Function for $(\omega_n, \zeta) = (6, 0.8)$: 0 Episodes

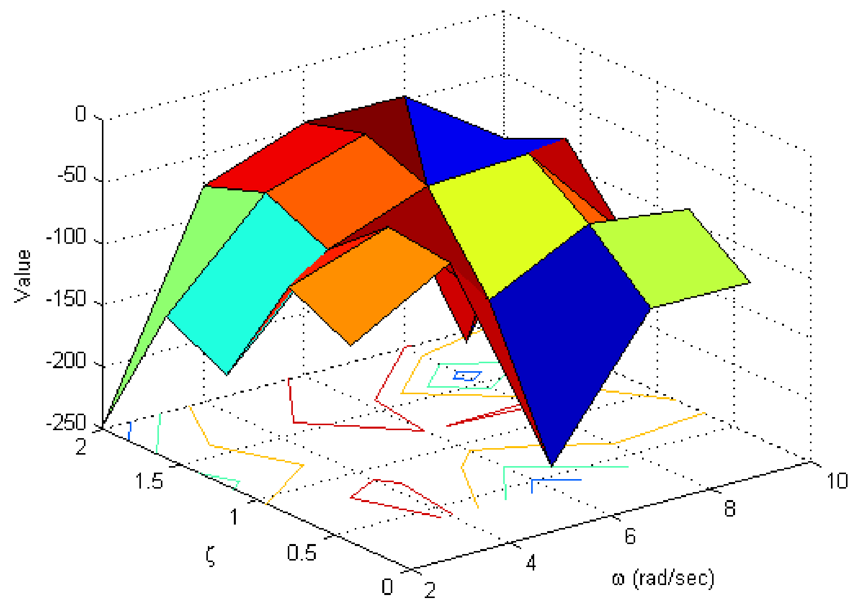


Figure 4.23: $2^{nd}$ Order Value Function for $(\omega_n, \zeta) = (6, 0.8)$: 200 Episodes

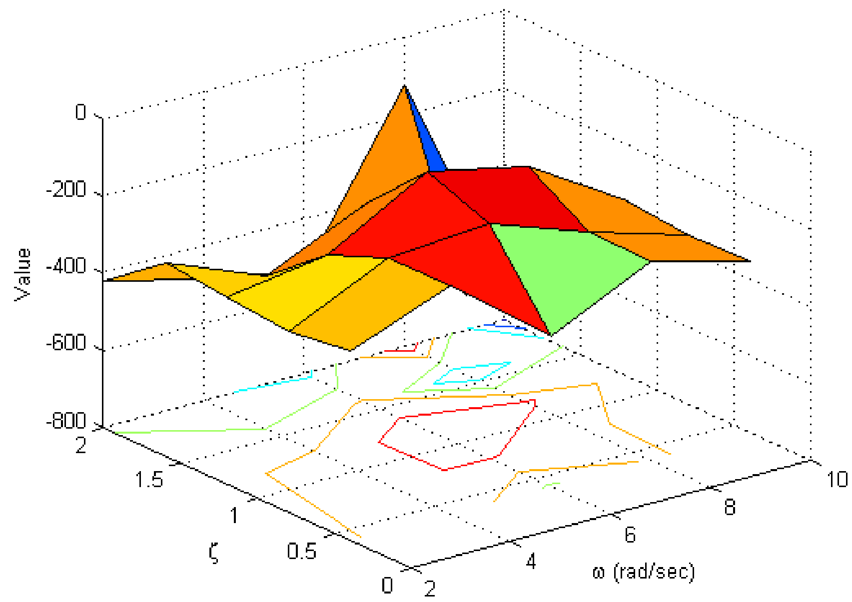


Figure 4.24: $2^{nd}$ Order Value Function for $(\omega_n, \zeta) = (6, 0.8)$: 1,000 Episodes

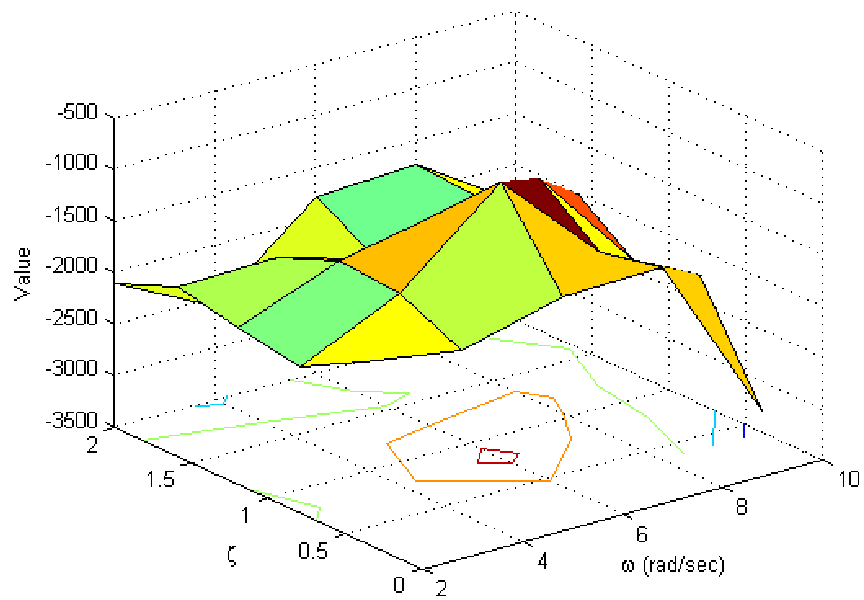


Figure 4.25: $2^{nd}$ Order Value Function for $(\omega_n, \zeta) = (6, 0.8)$: 5,000 Episodes

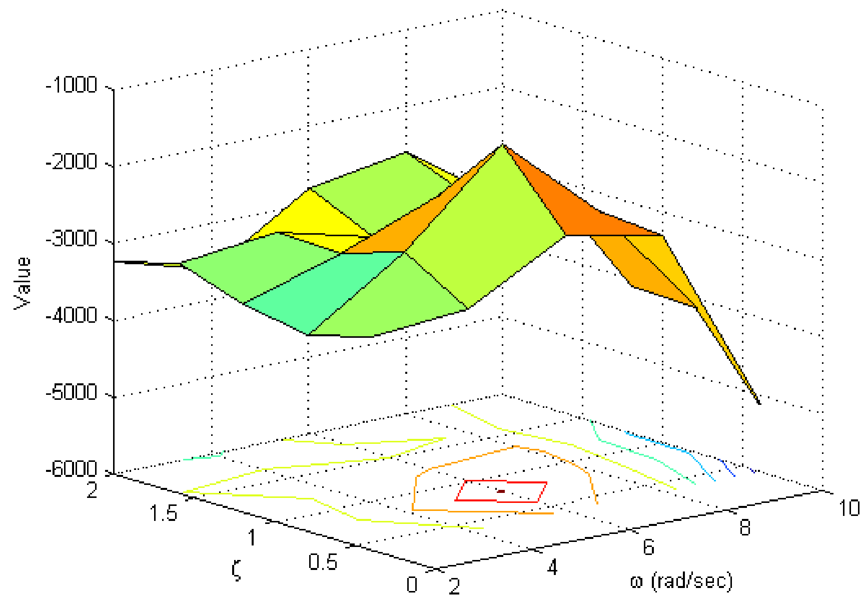


Figure 4.26: $2^{nd}$ Order Value Function for $(\omega_n, \zeta) = (6, 0.8)$: 10,000 Episodes

The way that the $V_2$ function is formulated implies that the value determined is for the *combination* of the variables $\omega_n$ and $\zeta$ rather than designating values for each separately. This is done because it is both of these variables together that determines the behavior of a system, and not either one separately. However, if one were to wish to see how they are valued individually, the average values can be determined. If the value function entries for each instance of a particular $\omega_n$ are averaged together, an estimate of the value for that frequency is determined. Likewise, the same can be done for the damping ratio. Figures 4.27-4.33 show how the average values for the individual parameters evolve over time.

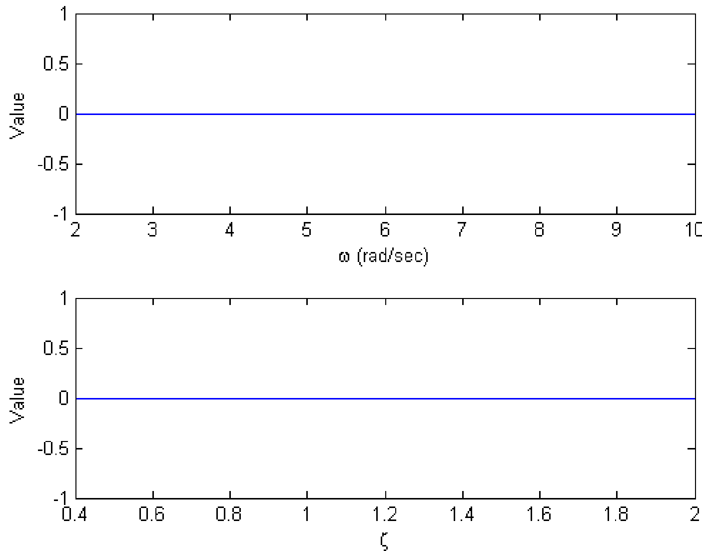


Figure 4.27: Average $2^{nd}$ Order Values for $(\omega_n, \zeta) = (6, 0.8)$: 0 Episodes

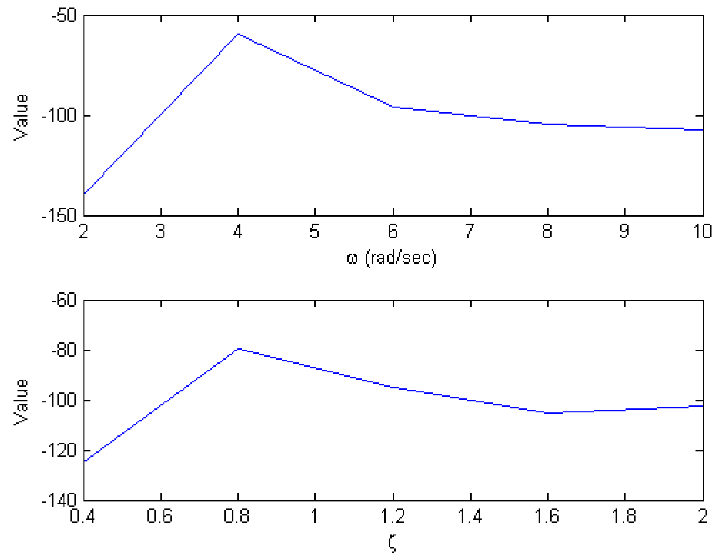


Figure 4.28: Average $2^{nd}$ Order Values for $(\omega_n, \zeta) = (6, 0.8)$: 200 Episodes

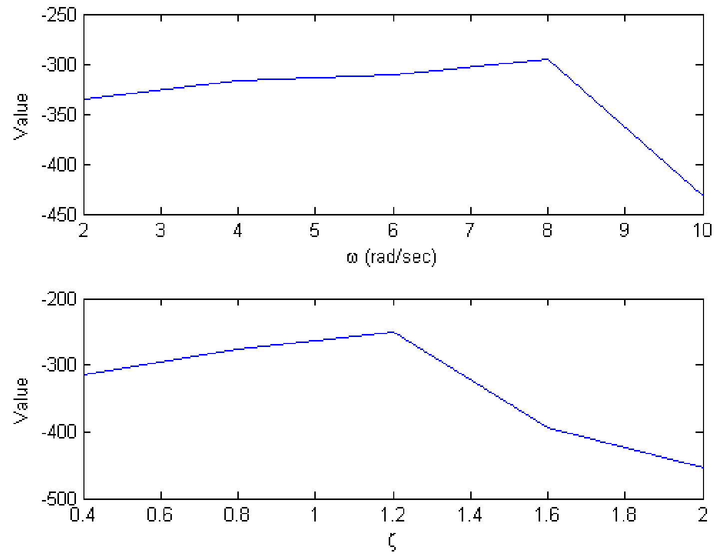


Figure 4.29: Average $2^{nd}$ Order Values for $(\omega_n, \zeta) = (6, 0.8)$: 1,000 Episodes

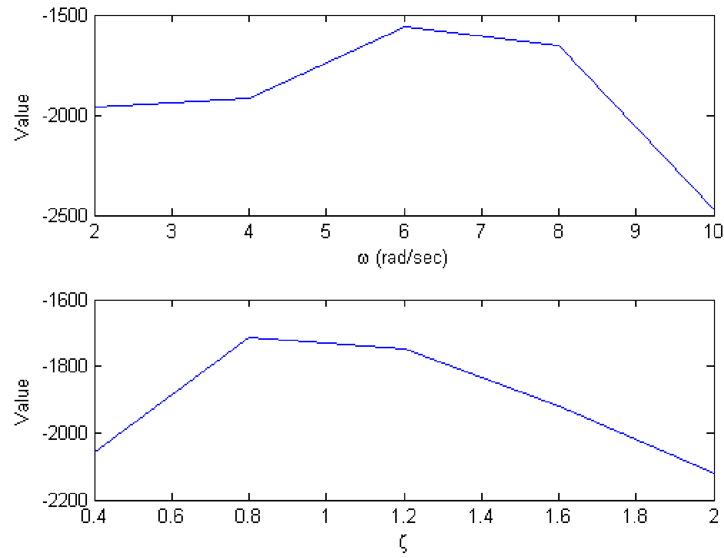


Figure 4.30: Average $2^{nd}$ Order Values for $(\omega_n, \zeta) = (6, 0.8)$: 5,000 Episodes

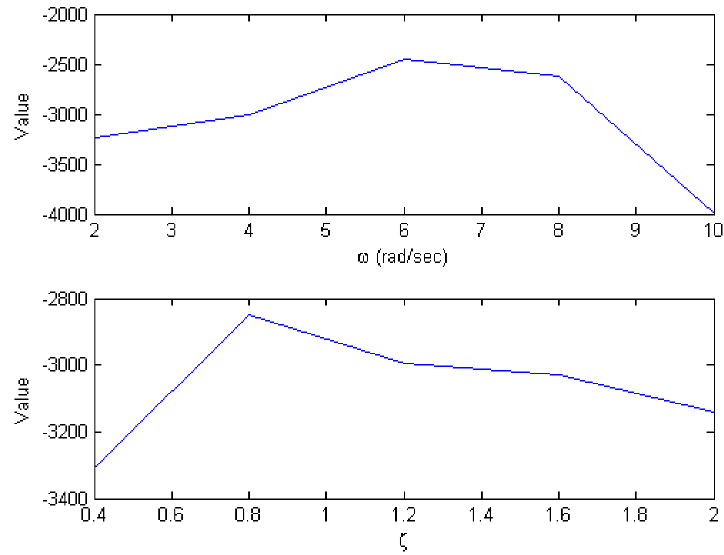


Figure 4.31: Average $2^{nd}$ Order Values for $(\omega_n, \zeta) = (6, 0.8)$: 10,000 Episodes

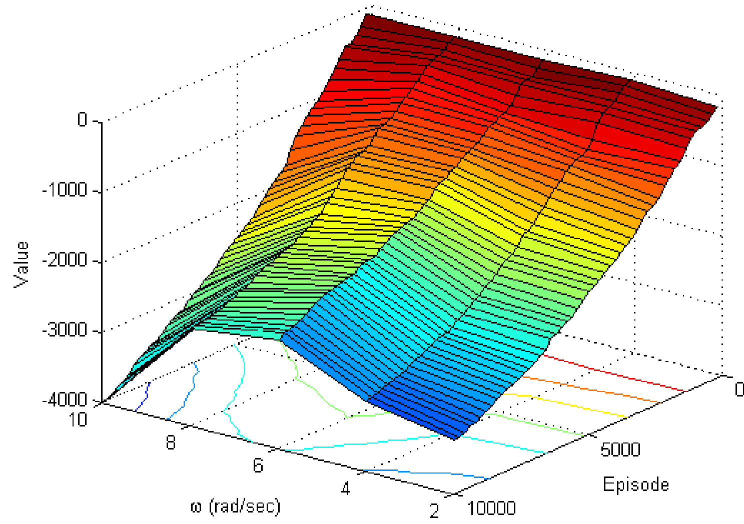


Figure 4.32: Average $\omega_n$ Value History for $(\omega_n, \zeta) = (6, 0.8)$

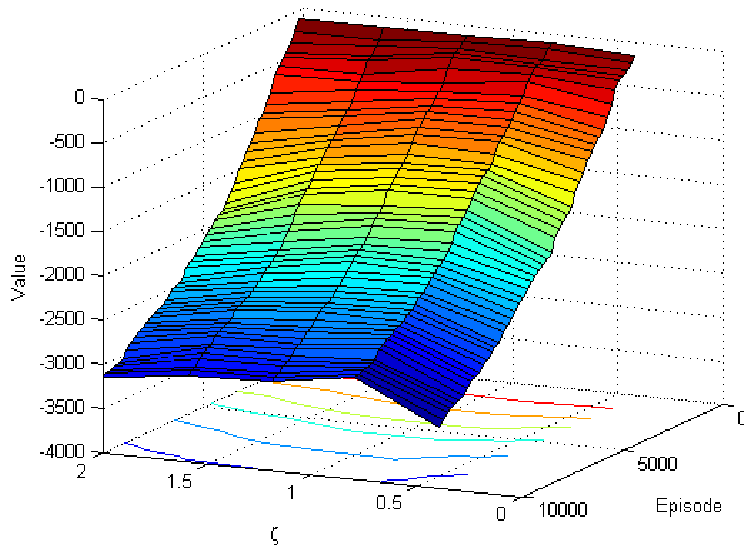


Figure 4.33: Average $\zeta$ Value History for $(\omega_n, \zeta) = (6, 0.8)$

The final version of the average value function for each parameter is shown in Figure 4.31. The final values for $V_2$ are shown in Table 4.3. The mean value and the variance of values over possible combinations of frequency and damping ratio are shown in Equations 4.55 and 4.56.

Table 4.3: $V_2$ after 10,000 Episodes: $(\omega_n, \zeta) = (6, 0.8)$

| | $\omega_n = 2$ | $\omega_n = 4$ | $\omega_n = 6$ | $\omega_n = 8$ | $\omega_n = 10$ |
|---|---|---|---|---|---|
| $\zeta = 0.4$ | $-3220$ | $-3107$ | $-2412$ | $-2671$ | $-5130$ |
| $\zeta = 0.8$ | $-3450$ | $-2610$ | $-1466$ | $-2606$ | $-4110$ |
| $\zeta = 1.2$ | $-3283$ | $-2883$ | $-2393$ | $-2323$ | $-4092$ |
| $\zeta = 1.6$ | $-3008$ | $-2855$ | $-3180$ | $-2940$ | $-3155$ |
| $\zeta = 2.0$ | $-3240$ | $-3602$ | $-2795$ | $-2576$ | $-3484$ |

$$E[V_2] = -3.06 \times 10^3 \tag{4.55}$$

$$Var[V_2] = 5.05 \times 10^5 \tag{4.56}$$

Having shown that the SODL algorithm is capable of accurately learning the natural frequency and damping ratio of a second-order system, the process was repeated with new second-order dynamics. The rotational dynamics of the rotating robot were altered so that the natural frequency was now $\omega_n = 10$ rad/sec and the damping ratio was $\zeta = 1.2$. This system should respond faster than the one before because the natural frequency is faster, but it also is slightly overdamped. The equation of motion governing the rotational dynamics of this system is shown in Equation 4.57.

$$\ddot{\psi} = -24\dot{\psi} + 100(\psi_c - \psi) \tag{4.57}$$

This system was allowed to learn for 10,000 episodes using the SODL algorithm. Over the course of these learning episodes, the SODL algorithm was able to successfully converge to the correct parameter values of $\omega_n = 10$ rad/sec and $\zeta = 1.2$. Figures 4.34-4.38 show the evolution of the SODL value function $V_2$ over the course of the 10,000 learning episodes completed.

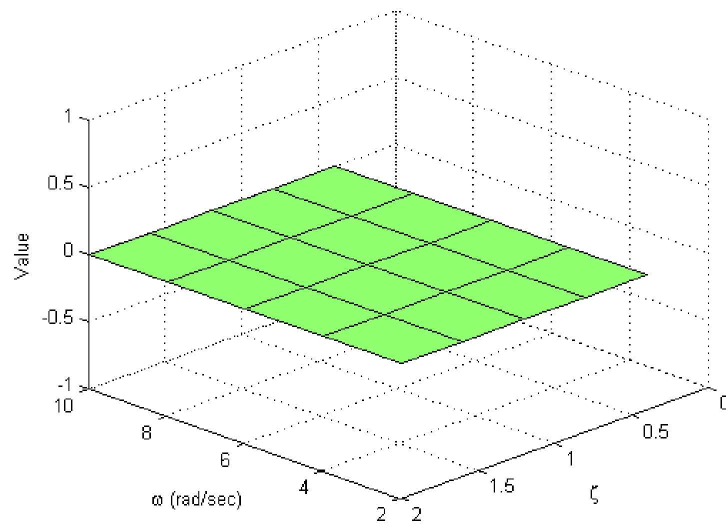


Figure 4.34: $2^{nd}$ Order Value Function for $(\omega_n, \zeta) = (10, 1.2)$: 0 Episodes

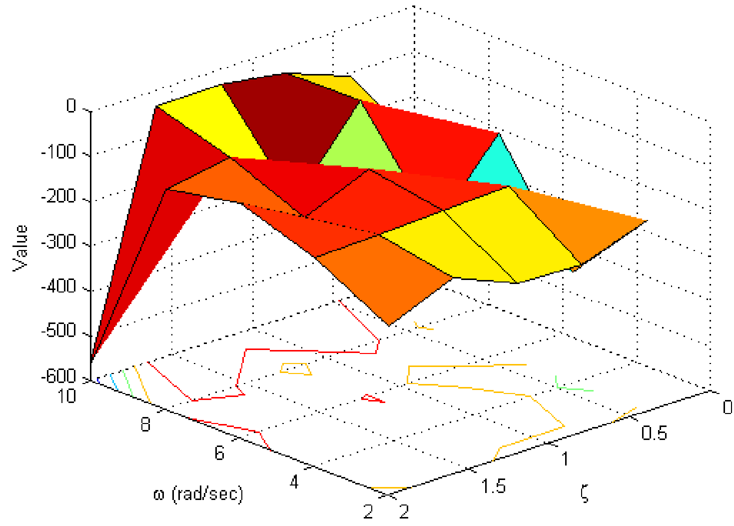


Figure 4.35: $2^{nd}$ Order Value Function for $(\omega_n, \zeta) = (10, 1.2)$: 200 Episodes

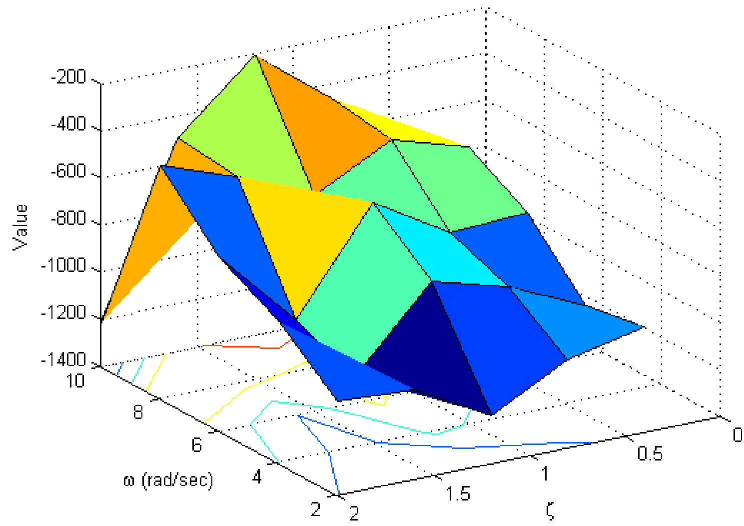


Figure 4.36: $2^{nd}$ Order Value Function for $(\omega_n, \zeta) = (10, 1.2)$: 1,000 Episodes

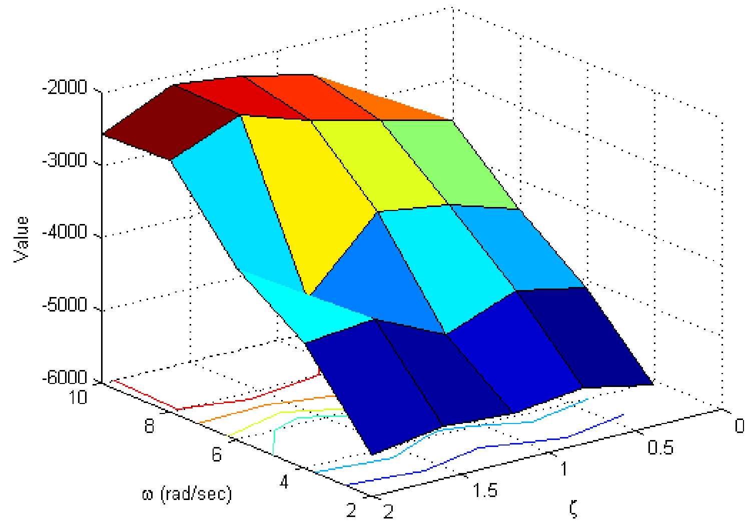


Figure 4.37: $2^{nd}$ Order Value Function for $(\omega_n, \zeta) = (10, 1.2)$: 5,000 Episodes

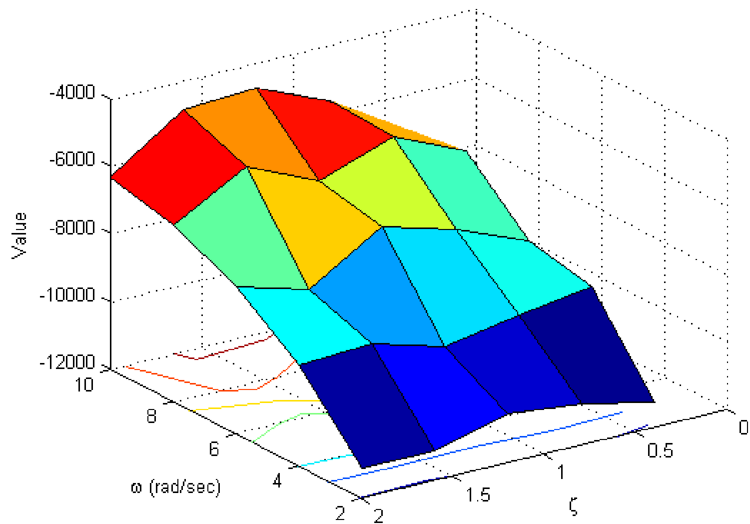


Figure 4.38: $2^{nd}$ Order Value Function for $(\omega_n, \zeta) = (10, 1.2)$: 10,000 Episodes

The average values over the evolution of $V_2$ are able to show the approximate evolution of the individual values for $\omega_n$ and $\zeta$. Figures 4.39-4.43 show the evolution of the individual parameters' average values.

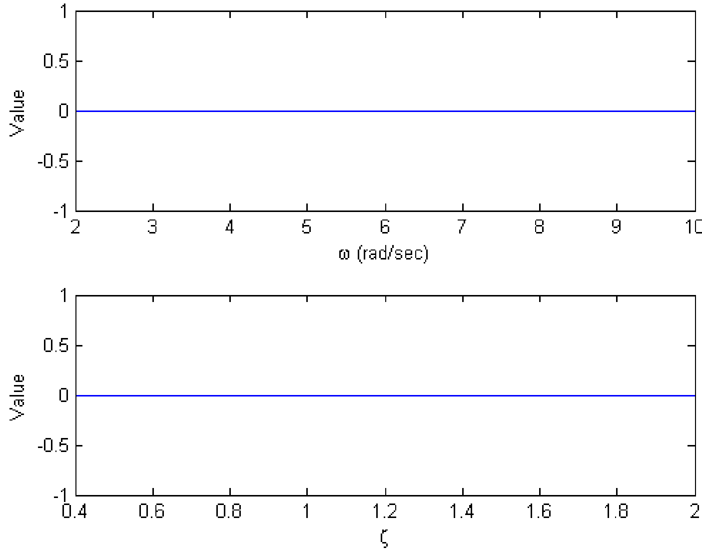


Figure 4.39: Average $2^{nd}$ Order Values for $(\omega_n, \zeta) = (10, 1.2)$: 0 Episodes

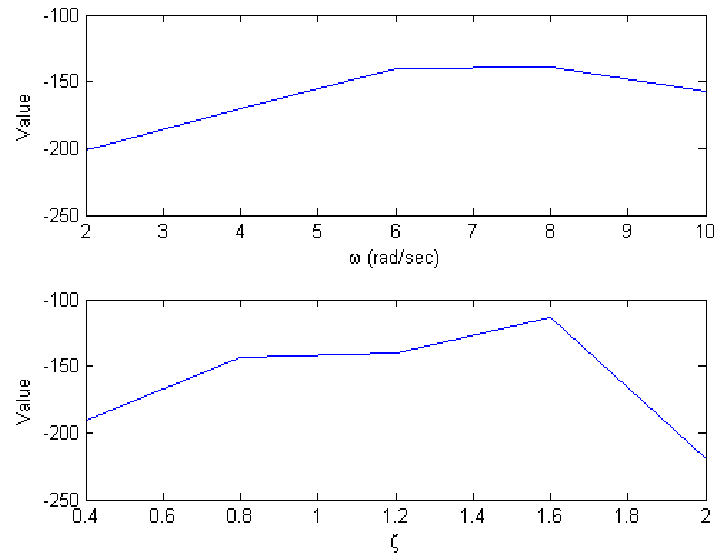


Figure 4.40: Average $2^{nd}$ Order Values for $(\omega_n, \zeta) = (10, 1.2)$: 200 Episodes

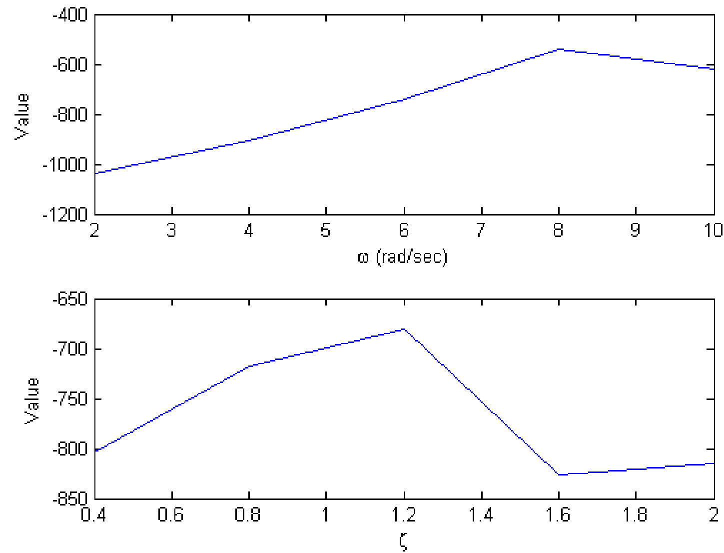


Figure 4.41: Average $2^{nd}$ Order Values for $(\omega_n, \zeta) = (10, 1.2)$: 1,000 Episodes

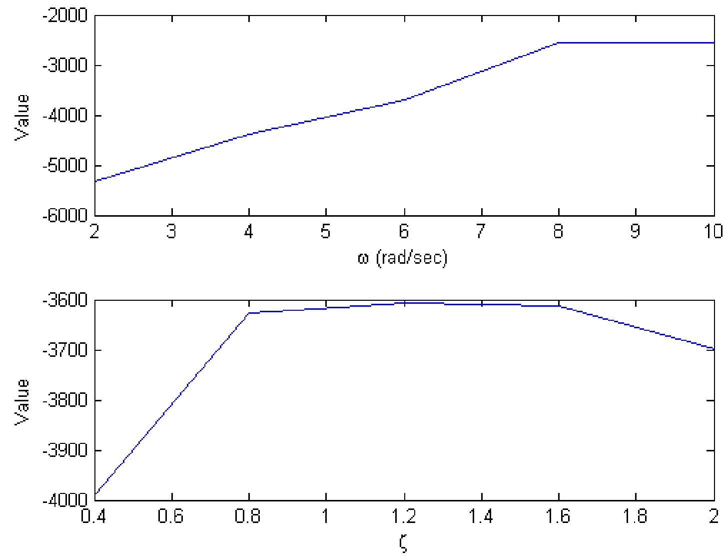


Figure 4.42: Average $2^{nd}$ Order Values for $(\omega_n, \zeta) = (10, 1.2)$: 5,000 Episodes

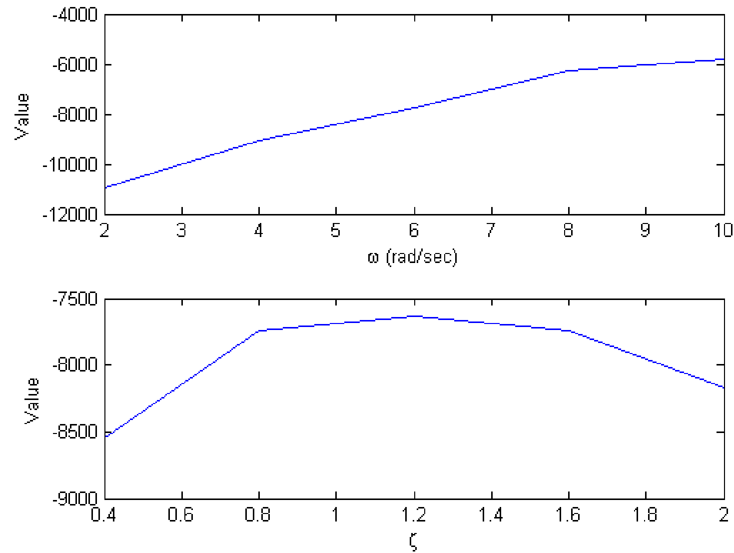


Figure 4.43: Average $2^{nd}$ Order Values for $(\omega_n, \zeta) = (10, 1.2)$: 10,000 Episodes

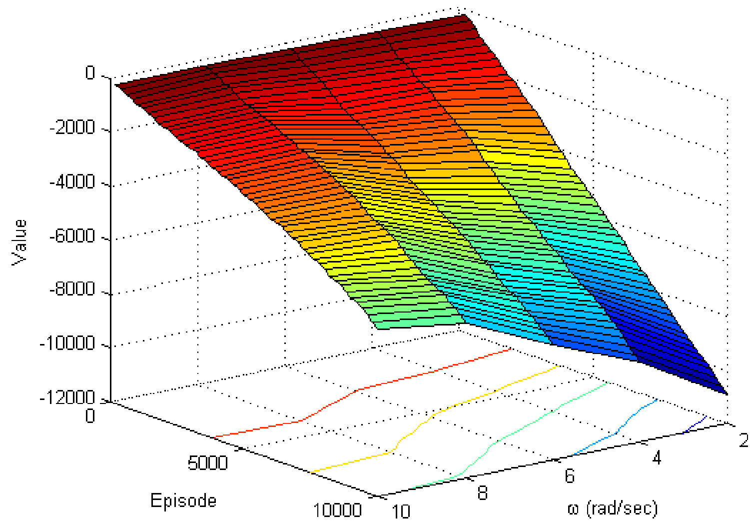


Figure 4.44: Average $\omega_n$ Value History for $(\omega_n, \zeta) = (10, 1.2)$

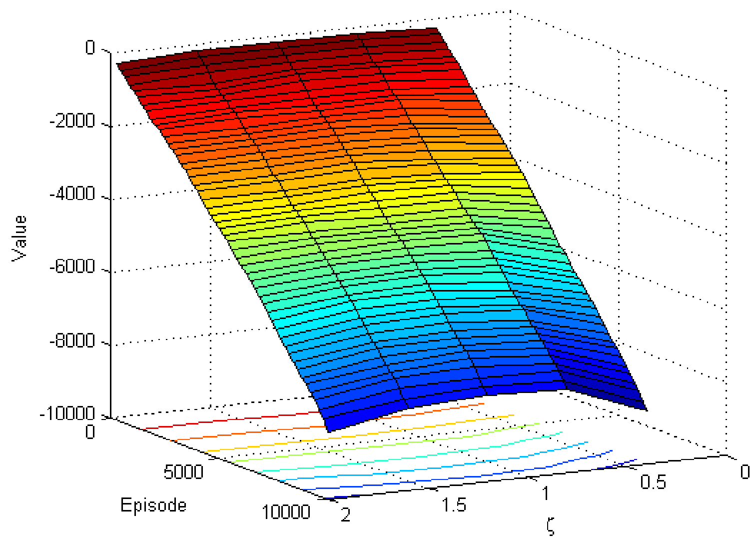


Figure 4.45: Average $\zeta$ Value History for $(\omega_n, \zeta) = (10, 1.2)$

The final version of the average value function for each parameter is shown in Figure 4.43. The final values for $V_2$ are shown in Table 4.4. The mean value and the variance of values over possible combinations of frequency and damping ratio are shown in Equations 4.58 and 4.59.

Table 4.4: $V_2$ after 10,000 Episodes: $(\omega_n, \zeta) = (10, 1.2)$

| | $\omega_n = 2$ | $\omega_n = 4$ | $\omega_n = 6$ | $\omega_n = 8$ | $\omega_n = 10$ |
|---|---|---|---|---|---|
| $\zeta = 0.4$ | $-11247$ | $-8765$ | $-8401$ | $-6698$ | $-7618$ |
| $\zeta = 0.8$ | $-10756$ | $-9082$ | $-7527$ | $-5721$ | $-5615$ |
| $\zeta = 1.2$ | $-10520$ | $-9511$ | $-6902$ | $-6524$ | $-4729$ |
| $\zeta = 1.6$ | $-11128$ | $-8905$ | $-8259$ | $-5583$ | $-4835$ |
| $\zeta = 2.0$ | $-11085$ | $-9018$ | $-7629$ | $-6772$ | $-6321$ |

$$E[V_2] = -8.01 \times 10^3 \tag{4.58}$$

$$Var[V_2] = 3.89 \times 10^6 \tag{4.59}$$

### *4.3.3 Multiple States*

Now that the FODL and SODL algorithms have been shown to successfully learn accurate models of the states, it is of interest to show that more than one state can be modeled from the same agent. In this simulation, the same rotating robot example from before will be explored for the case where the speed changes alongside the heading angle. For this example, the robot begins at rest and is commanded to accelerate to a known maximum speed, $V_{max}$. When the agent reaches its goal region, the robot is commanded to return to rest.

Since the robot cannot overshoot its maximum or minimum speed, it is appropri-

ate to model the dynamics of this state as first-order, and the FODL algorithm will be utilized for determining the time constant ($\tau_V$) that describes this process. The heading angle will also be commanded to change for controlling the robot to its goal, and it is modeled with second-order dynamics. The SODL algorithm is thus used to determine the natural frequency ($\omega_{n,\psi}$) and damping ratio ($\zeta_\psi$) that best model these dynamics. The governing equations of motion are shown in Equations 4.60-4.63.

$$\dot{x} = V \cos \psi \tag{4.60}$$

$$\dot{y} = V \sin \psi \tag{4.61}$$

$$\dot{V} = (V_c - V)\tau_V^{-1} \tag{4.62}$$

$$\dot{\psi} = -2\zeta_\psi \omega_{n,\psi} \dot{\psi} + \omega_{n,\psi}^2 (\psi_c - \psi) \tag{4.63}$$

In the following simulation, the maximum speed of the robot is 2 m/s, and the time constant that describes the speed dynamics is $\tau_V = 1$ sec. The robot begins at rest and accelerates to 2 m/s, maintaining that maximum speed until it reaches the goal zone of $[x, y] = [0, 0] \pm [1, 1]$ m. When the goal has been reached, the robot decelerates back to rest using the same time constant. For the heading angle changes, the natural frequency is $\omega_{n,\psi} = 6$ rad/s and the damping ratio is $\zeta_\psi = 0.8$. Each of these states to be modeled will learn their own dynamics value function using the two appropriate algorithms.

After simulating this robot for 10,000 learning episodes, the FODL and SODL algorithms were successfully able to model these dynamics. The resulting value functions are reported in Tables 4.5 and 4.6. The mean and variance of $V_1$ values for the velocity time constant are shown in Equations 4.64 and 4.65. The mean and variance of $V_2$ values for the heading angle frequency and damping ratio are

shown in Equations 4.66 and 4.67. It can be seen from these tables that the proper values for these dynamics parameters were determined by their respective algorithm. Figures 4.46-4.47 show the individual value functions.

Table 4.5: $V_1$ for Robot Speed: $\tau = 1$

| $\tau_V(sec)$ | $V_1$ |
|---|---|
| 0.1 | $-12.17 \times 10^4$ |
| 0.2 | $-6.99 \times 10^4$ |
| 0.3 | $-4.85 \times 10^4$ |
| 0.4 | $-3.90 \times 10^4$ |
| 0.5 | $-3.45 \times 10^4$ |
| 0.6 | $-3.05 \times 10^4$ |
| 0.7 | $-2.41 \times 10^4$ |
| 0.8 | $-2.42 \times 10^4$ |
| 0.9 | $-1.89 \times 10^4$ |
| 1.0 | $-0.87 \times 10^4$ |
| 1.1 | $-1.70 \times 10^4$ |
| 1.2 | $-1.87 \times 10^4$ |
| 1.3 | $-1.81 \times 10^4$ |
| 1.4 | $-2.06 \times 10^4$ |
| 1.5 | $-2.15 \times 10^4$ |
| 1.6 | $-2.24 \times 10^4$ |
| 1.7 | $-1.87 \times 10^4$ |
| 1.8 | $-2.53 \times 10^4$ |
| 1.9 | $-2.63 \times 10^4$ |
| 2.0 | $-2.33 \times 10^4$ |

$$E[V_1] = -31.6 \times 10^3 \tag{4.64}$$

$$Var[V_1] = 6.27 \times 10^8 \tag{4.65}$$

Table 4.6: $V_2$ for Robot Heading with Velocity Changes: $(\omega_{n,\psi}, \zeta_\psi) = (6, 0.8)$

| | $\omega_{n,\psi} = 2$ | $\omega_{n,\psi} = 4$ | $\omega_{n,\psi} = 6$ | $\omega_{n,\psi} = 8$ | $\omega_{n,\psi} = 10$ |
|---|---|---|---|---|---|
| $\zeta_\psi = 0.4$ | $-6694$ | $-5002$ | $-3931$ | $-6582$ | $-9926$ |
| $\zeta_\psi = 0.8$ | $-5933$ | $-4523$ | $-1750$ | $-5020$ | $-8600$ |
| $\zeta_\psi = 1.2$ | $-6428$ | $-4821$ | $-3551$ | $-4931$ | $-6467$ |
| $\zeta_\psi = 1.6$ | $-5860$ | $-4475$ | $-4060$ | $-5012$ | $-6591$ |
| $\zeta_\psi = 2.0$ | $-6224$ | $-4804$ | $-5218$ | $-5236$ | $-6239$ |

$$E[V_2] = -5.52 \times 10^3 \tag{4.66}$$

$$Var[V_2] = 2.61 \times 10^6 \tag{4.67}$$

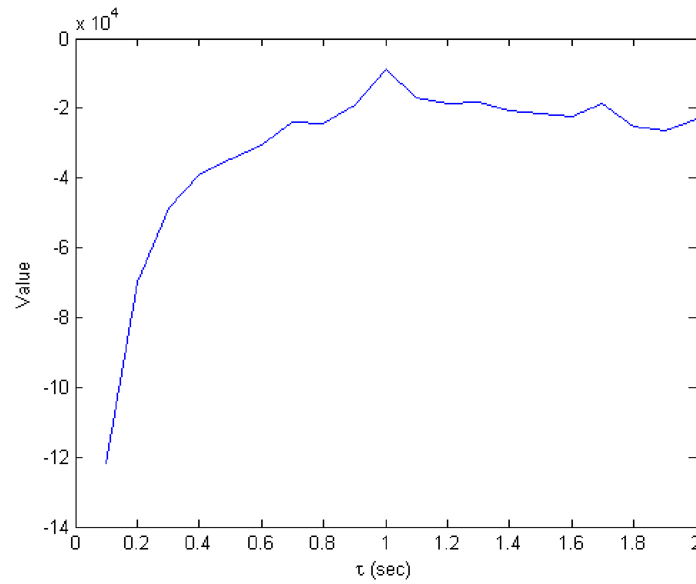


Figure 4.46: $V_1$ for Robot Speed Plot

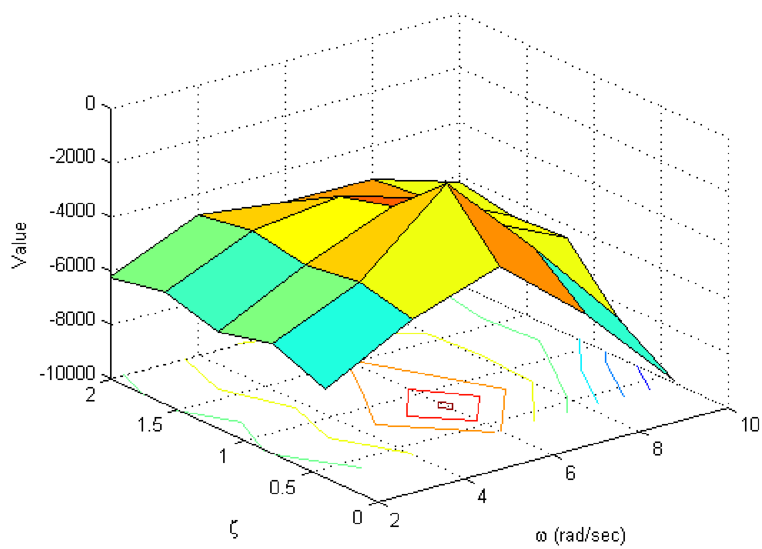


Figure 4.47: $V_2$ for Robot Heading with Velocity Changes Plot

## 4.4 Sample Time Ranges

In this section, the effect of sampling rate on the ability to learn dynamics is explored. This is done by testing the First-Order Dynamics Learning and Second-Order Dynamics Learning algorithms on the rotating robot example under a variety of sampling rates. For the first case, the simulation was performed using a time constant of $\tau = 0.2$, and a large range of sampling rates were used.

The first range that was tested was a range of fast sample times ranging from $T = 0.01$ sec to $T = 0.1$ sec in 0.01 sec intervals. For this range, it was found that the FODL algorithm successfully converged to the proper time constant of $\tau = 0.2$ for all of the sample times tested. These values are shown in Table 4.7. A wider range was then used for longer sample times ranging from $T = 0.05$ sec to $T = 1.0$ sec in 0.05 sec intervals. In this case, the FODL algorithm was able to successfully converge to the correct time constant in all cases except for the last sample time of 1 sec. These values are reported in Table 4.8. This indicates a possible divergence from success for larger sample times, so a final range was tested of $T = 1.0$ sec to $T = 2.0$ sec in 0.1 sec intervals. In this case, as shown by Table 4.9, the FODL begins diverging from the correct time constant. The algorithm determines a time constant that is within 0.1 sec of the correct value for sample times up to 1.5 sec, and then diverges to being within 0.2 sec afterward. This indicates that the ability of the FODL algorithm to converge to the correct time constant is indeed dependent on having a fast sampling rate. For the case of this example, the sampling frequency must be faster than 1 Hz.

Since the shorter sampling rates seem to provide accurate learning results for the FODL algorithm, it is next desired to see if there is a range of sampling rates that are too fast for accurate learning of the dynamics. The final range tested was a set of

Table 4.7: FODL with Sampling Ranges (Short)

| $T(sec)$ | $\tau = \text{argmax}_\tau V_1$ |
|---|---|
| 0.01 | 0.2 |
| 0.02 | 0.2 |
| 0.03 | 0.2 |
| 0.04 | 0.2 |
| 0.05 | 0.2 |
| 0.06 | 0.2 |
| 0.07 | 0.2 |
| 0.08 | 0.2 |
| 0.09 | 0.2 |
| 0.10 | 0.2 |

sample times ranging from $T = 0.001$ sec to $T = 0.01$ sec, as reported in Table 4.10. In this case, after 10,000 episodes it was found that as the sampling rates become too fast, the FODL algorithm has trouble determining the correct time constant. The FODL algorithm was able to determine the correct time constant of $\tau = 0.2$ for sample times that are at least $T = 0.007$ sec. For sample times shorter than this, the algorithm was unable to accurately determine the time constant. The overall range for this example for accurately determining the time constant is thus as shown in Equation 4.68.

Table 4.8: FODL with Sampling Ranges (Medium)

| $T(sec)$ | $\tau = \text{argmax}_\tau V_1$ |
|---|---|
| 0.10 | 0.2 |
| 0.15 | 0.2 |
| 0.20 | 0.2 |
| 0.25 | 0.2 |
| 0.30 | 0.2 |
| 0.35 | 0.2 |
| 0.40 | 0.2 |
| 0.45 | 0.2 |
| 0.50 | 0.2 |
| 0.55 | 0.2 |
| 0.60 | 0.2 |
| 0.65 | 0.2 |
| 0.70 | 0.2 |
| 0.75 | 0.2 |
| 0.80 | 0.2 |
| 0.85 | 0.2 |
| 0.90 | 0.2 |
| 0.95 | 0.2 |
| 1.00 | 0.3 |

Table 4.9: FODL with Sampling Ranges (Long)

| $T(sec)$ | $\tau = \text{argmax}_\tau V_1$ |
|---|---|
| 1.00 | 0.3 |
| 1.10 | 0.3 |
| 1.20 | 0.1 |
| 1.30 | 0.2 |
| 1.40 | 0.1 |
| 1.50 | 0.5 |
| 1.60 | 0.3 |
| 1.70 | 0.2 |
| 1.80 | 0.4 |
| 1.90 | 0.6 |
| 2.00 | 0.6 |

Table 4.10: FODL with Sampling Ranges (Tiny)

| $T(sec)$ | $\tau = \operatorname{argmax}_\tau V_1$ |
|---|---|
| 0.001 | 1.4 |
| 0.002 | 0.3 |
| 0.003 | 1.5 |
| 0.004 | 0.3 |
| 0.005 | 1.2 |
| 0.006 | 1.5 |
| 0.007 | 0.2 |
| 0.008 | 0.2 |
| 0.009 | 0.2 |
| 0.01 | 0.2 |

$$0.006 < T < 1 \tag{4.68}$$

Equation 4.68 indicates the accuracy with which the time constant, $\tau$, can be determined based on the sample time for the rotating robot example. It has been discovered that the prediction of the time constant is dependent on a particular range of $T$ that is bounded both above and below. For accurate determination of the time constant for this case, the sample time must be between 0.006 sec and 1.0 sec.

The same analysis was then completed for the SODL algorithm, with the case of $\omega_n = 6$ rad/s and $\zeta = 1.2$. For this case, it was determined that the range of $T$ required to learn was slightly more restrictive. Tables 4.11 and 4.12 show the results of this analysis.

Table 4.11: SODL with Sampling Ranges (Short)

| $T(sec)$ | $\omega_n = \text{argmax}_{\omega_n} V_2$ | $\zeta = \text{argmax}_{\zeta} V_2$ |
|---|---|---|
| 0.01 | 8 | 1.6 |
| 0.02 | 6 | 1.2 |
| 0.03 | 6 | 1.2 |
| 0.04 | 6 | 1.2 |
| 0.05 | 6 | 1.2 |
| 0.06 | 6 | 1.2 |
| 0.07 | 6 | 1.2 |
| 0.08 | 6 | 1.2 |
| 0.09 | 6 | 1.2 |
| 0.10 | 6 | 1.2 |

As Tables 4.11-4.12 show, the range of sample times that allows for successful learning of the natural frequency and damping ratio for this problem is smaller than for the first-order case. The range allowed for this particular example is as shown in Equation 4.69.

Table 4.12: SODL with Sampling Ranges (Long)

| $T(sec)$ | $\omega_n = \text{argmax}_{\omega_n} V_2$ | $\zeta = \text{argmax}_{\zeta} V_2$ |
|---|---|---|
| 0.10 | 6 | 1.2 |
| 0.20 | 6 | 1.2 |
| 0.30 | 6 | 1.2 |
| 0.40 | 6 | 1.2 |
| 0.50 | 6 | 1.2 |
| 0.60 | 6 | 1.2 |
| 0.70 | 6 | 1.2 |
| 0.80 | 6 | 1.2 |
| 0.90 | 4 | 0.8 |
| 1.00 | 8 | 1.6 |

$$0.01 < T < 0.9 \tag{4.69}$$

The results presented in this chapter indicate that the algorithms FODL and SODL are capable of accurately determining a model of an agent's individual state dynamics for first- and second-order systems, respectively. Determining this information is especially of interest in the case of a multiagent system with agents of unknown dynamics. The use of these 2 algorithms, along with SDQL, in the case of commanding agents in a hierarchical multiagent system is explored in the following chapter.

# 5. MULTIAGENT SYSTEMS

In systems that involve only one independent agent, determining proper high-level commands for goal achievement is typically trivial. If a goal must be achieved and there is only one agent to command, there is no decision to be made. The agent must be the one to achieve the goal. However, in systems with multiple independent agents there is a consideration that must be made for hierarchical decision making. If there are multiple agents in a system and only one is needed to achieve a new goal, which agent should be tasked? In this chapter, the idea of utilizing the previously discussed learned dynamics to make hierarchical decisions for goal assignment is explored.

## 5.1 Learning in Multiagent Systems

The idea of utilizing learning techniques in the control of multiagent systems has been investigated for a variety of system structures, and for a variety of learning methods [35, 4]. Multiagent systems can include both cooperative agents and independent agents [29, 47, 43]. They can be handled with a single homogeneous level or with hierarchical commands between higher- and lower-level agents [30, 17, 49]. The structure of the multiagent system effects the manner in which the learned information is utilized between separate agents. This section discusses research that has been done in RL for multiagent systems for the purpose of determining which type of multiagent system solution is useful for this research.

### *5.1.1 Single-Level Multiagent Learning*

In work by , a $Q$-learning-based algorithm called Hyper-Q learning is introduced for systems involving multiple agents [47]. Hyper-Q learning learns values of mixed

strategies rather than base actions, and Bayesian inference is used to estimate other agent's strategies from observed actions. The $Q$-matrix of this technique involves the joint mixed strategy of all other agents. Convergence of this algorithm is dependent on both the agents' learning dynamics and the function approximation technique utilized. Here, agents can estimate the strategies of other agents by means of either model-based or model-free methods. Model-free estimation methods are explored, and Bayesian inference is chosen for estimating agent strategies based on the history of observed actions. This performs well for the simple rock-paper-scissors game examples shown in this paper, but it has not been shown for more complex systems.

The concept of independent vs. cooperative agent learning was investigated in research performed by [43]. In this work, the author uses hunter-prey MDP games to determine how the level of information sharing affects the convergence behavior for multiple agents. The hunter-prey games can be considered to be multiagent gridworld navigation with moving goals, and in this case information can be shared between agents that includes sensory information, policies learned, and solution episodes experienced. This examination was performed solely on these hunter-prey style games, and the results of this research demonstrate that there is a difference in payoff that must be considered between independent agents and cooperative agents. The paper concludes that sharing information and learning cooperatively improves the convergence time and allows for joint task achievement in these single-level scenarios. However, it takes a much larger state-space to learn cooperative behavior for joint tasks, and the sharing of knowledge or episodes comes with a cost in communication.

The idea of using MDP games, or general stochastic games, for exploring multiagent Reinforcement Learning methods was further investigated by . In this work, multiagent systems were examined by using minimax criterion to determine optimal policies for zero-sum stochastic games involving 2-players. This minimax criterion

allows the agent to converge to a fixed strategy in winning the game by determining a policy capable of performing as well as possible against the worst possible opponent. This limited view of multiagent system learning is further expanded later in work done by , who explored the application of Reinforcement Learning to general-sum stochastic games. This paper uses the Nash equilibria conditions to design the learning agent for the purpose of non-cooperative Reinforcement Learning. In these non-cooperative games, none of the agents are aware of each other or consider each other in their decision making. It is shown that the algorithm proposed converges, but it is entirely dependent on certain restrictions to the games.

These research areas involving multiple agents include the investigation of games where one agent is attempting to defeat another, but in many practical applications it is more helpful to consider the case of multiagent systems where the goal is cooperation to achieve a mutual goal. In research performed by , multiagent Reinforcement Learning in cooperative coordination games is considered. These games are considered for both the cases of independent learners and joint-action learners. For the independent case, each agent has a separate $Q$-learner that is unaware of any other agents. In the joint-action learner case, each agent has a learner that updates and makes decisions based on the joint action of all observable agents. The authors found that both cases converge in finite time and the joint-action learner converges faster, but both cases converge on a similar time scale. It was also found that while the joint-action learner converged slightly faster, it is limited in the fact that it takes a lot of contrary experience to learn to choose a path that has already been heavily reinforced. This indicates that in scenarios where multiple agents are used to achieve a goal rather than combat each other, using independent learners for each agent provides comparable convergence time with a simpler implementation and more influential later learning.

In work done by , a distributed Reinforcement Learning approach was investigated for determining an optimal policy for a cooperative multiagent system [29]. The systems considered in this work consists of individual agents that learn independently and have no information communicating between individual agents. After learning, The agents in this system must learn optimal policies and agree on a single policy that leads to optimal behavior for the joint team effort. The combined MDP is handled by treating them collectively as a single MDP where the action vector is composed of an augmented vector of all possible actions for all agents. This is accomplished by creating a central $Q$-matrix with state-action information for all agents, but each agent only updates and selects from the subset of the $Q$-matrix that represents the individual agent's elementary actions. Each agent assumes that all agents will act optimally in this scenario. This research reveals that even if each agent chooses their individual action optimally, the joint action selection can still be suboptimal. This indicates that the agents must have some level of coordination between them, whether it be mutually achieved or through some hierarchical command structure.

The concept of coordination in cooperative multiagent systems was further explored by the works of [19]. This research involves cooperative multiagent systems where joint actions are chosen but the agents do not have the ability to observe their mutual actions. The agents must choose their joint actions using environmental feedback. The action selection used is called the Boltzmann strategy, which uses a probability function to determine an action's usefulness. This usefulness probability is used by each agent when choosing their individual actions, and an action selection strategy is proposed called the Frequency Maximum Q-Value heuristic. This gives the estimated value of choosing an action by a linear combination of the $Q$-table value and the possible reward weighted by the probability of receiving that reward. The resulting value is then used in the Boltzmann strategy for action selection. This

was tested on the same cooperative games used by  and , and was shown to outperform their strategies by converging to optimal actions in each game, although it proved to perform equally poorly in truly stochastic games.

In each of these scenarios explored, Reinforcement Learning-based techniques were explored for multiagent systems where agents operate at the same level with no hierarchy. The results presented in these papers indicate that there is promise to further exploration of these methods, but perhaps some level of supervision is necessary to achieve the kind of cooperation required for more complex systems.

#### *5.1.2 Hierarchical Multiagent Learning*

In work by , cooperative hierarchical Reinforcement Learning algorithms are discussed for learning in multiagent systems with hierarchical policies [13, 14]. In this research, hierarchical methods are investigated where the communication level between individual agents are sorted into a hierarchy dependent on the level of cooperation required. In non-cooperative subtasks, joint actions are not required since the agents act completely independently of each other. Each individual agent can be modeled as individual SMDPs, and do not require communication to accomplish the subtasks. In cooperative subtasks, the system is modeled as a joint-action learner where the actions taken are determined by the union of actions between agents. This is done at a high-level of abstraction and does not involve low-level subtasks. The cooperation levels are defined by the user, and if there are subtasks that would benefit from cooperation but are defined as non-cooperative they will be suboptimal. However, this methodology has the benefit of making cooperation more efficient since cooperation subtasks take less time to complete due to infrequent communication. The complexity of the learner is also greatly reduced because the only information that must be stored is local agent information. The authors conclude that dividing

tasks into hierarchy reduces learning time and time to achieve tasks.

In fact, in a survey of multiagent Reinforcement Learning published by  it is determined by the authors that after examining the multiagent Reinforcement Learning approaches to date, it would be helpful to future approaches to have some division of task structure into hierarchical methodology. This idea of hierarchical reinforcement learning is further expanded upon by the works of  in [53] and [54]. In these papers, the idea of hierarchical multiagent Reinforcement Learning is solidified into a series of multi-level automated supervisors. The agents that perform tasks are called "workers" and the supervisors evaluate the worker agents' policies and give suggestions of actions to perform. Tasks are allocated to workers based on the evaluation of the supervisor agents, and the worker agents do not have to carry around extra baggage regarding the state of other agents in the system. The automated supervisor is able to evaluate where tasks should be allocated based on information that the supervisor maintains about the lower-level worker agents. This automated supervisory approach to multiagent Reinforcement Learning provides faster convergence, accurate policies, and low need for communication.

It is this most recent approach to multiagent Reinforcement Learning that will be exploited in the examples shown in this chapter. By using a hierarchical approach to multiagent systems, a high-level automated supervisor can be used to allocate tasks to lower-level agents. These agents only need to know their own individual policies for achieving the task given to them, and do not need to be concerned with the other agents in the system. The high-level supervisor will provide information regarding the goal that each agent in the system should achieve, and the allocation of goals to agents will be determined by minimum time-to-goal for the total achievement of all goals in the system. Each agent in the system will learn their own action-value function, sample time value function, and dynamics value functions using the

algorithms presented in Chapters 3 and 4. The learned dynamics will be used by the supervisor agent to evaluate how long it will take each agent to achieve each goal, and then will allocate agents to goals by a minimum time consideration. The process for learning and implementing the supervisor in this multiagent system scenario is illustrated in Figure 5.1.

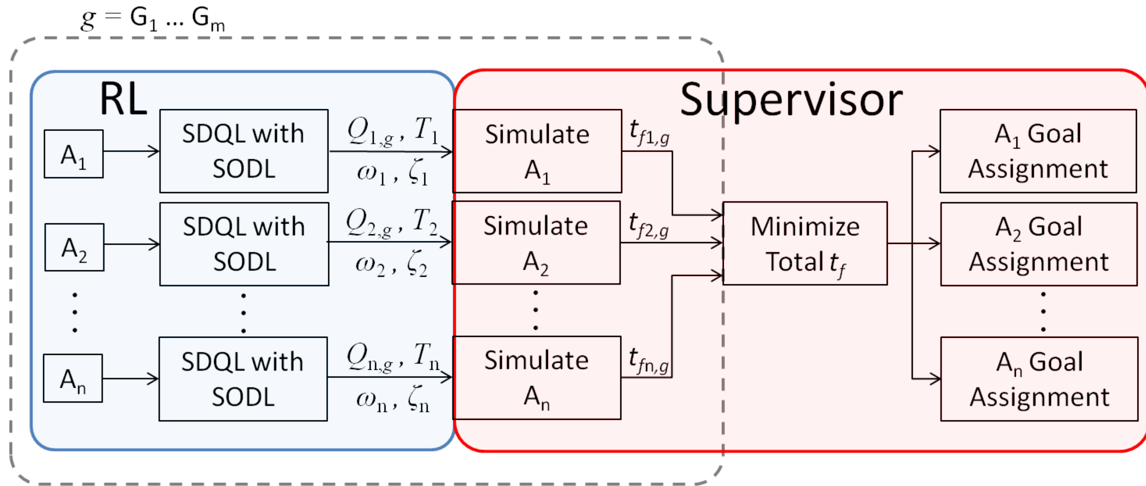


Figure 5.1: Hierarchical Multiagent System Diagram

## 5.2 Homogeneous Agents Examples

In this section, the case of a homogeneous multiagent system is explored. In this sense, a homogeneous multiagent system is a multiagent system where all of the agents have identical governing equations of motion. Since the dynamics of the agents in this system are all the same, they can each be controlled using the same policies. The high-level supervisor that allocates tasks to agents does so by a minimum total time consideration. In the case of agents with identical dynamics, the minimum distance and minimum time scenarios should be the same. The agents

in this example are all rotating robots such as are described in Section 3.4.2.

This system is set up with 1 high-level supervisor agent, 3 low-level rotating robot agents, and 3 goals to be reached. This system is illustrated in Figure 5.2. The high-level supervisor agent, $A_s$, is able to "see" state information about all 3 agents and all 3 goals. The supervisor is also able to communicate with the 3 agents, as shown by the dashed lines. This is how goal allocation is determined during the simulation.

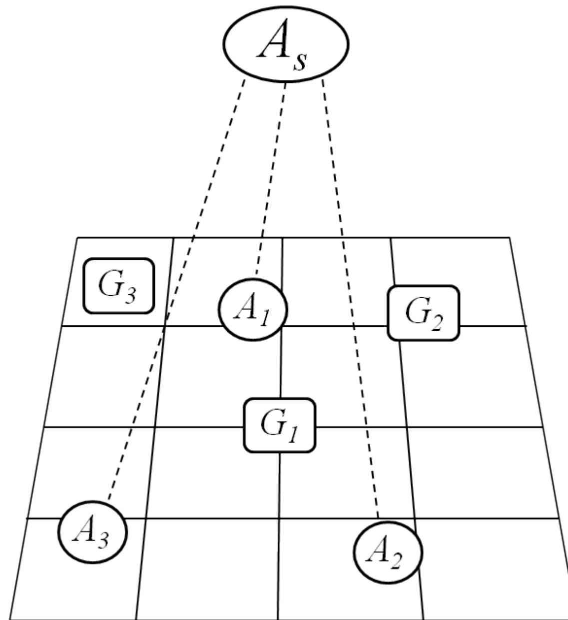


Figure 5.2: Hierarchical Multiagent System

At the beginning of simulation, the supervisor checks the initial conditions of each agent and compares them to the goals that need to be achieved. The dynamics value functions, $V_1$ or $V_2$, are checked by the supervisor for each agent for determining the best approximation of the agent dynamics. This information is used by the supervisor to determine which agents will be commanded to which goals for achieving all goals in minimum time. Each agent controls itself to its specified goal, so the sample time

for each agent will be determined individually using its respective sampled-data value function, $V_T$.

In these simulations, all of the agents will be identical in their dynamics, so they will all have the same governing equations of motion. The agents simulated here will be based on the rotating robot example from the previous chapters, and for simplicity these initial simulations will have first-order dynamics in their rotation. All agents begin at rest and accelerate to the same maximum forward speed of $V = 1$ m/s with a time constant of $\tau_V = 1$ sec, and they have a heading angle time constant of $\tau_\psi = 0.2$ sec. When they reach their goal region, they will be commanded to stop, and will decelerate back to $V = 0$ m/s with the same speed dynamics. The governing equations of motion for these agents are as shown in Equations 5.1-5.4.

$$\dot{x} = \cos\psi \tag{5.1}$$

$$\dot{y} = \sin\psi \tag{5.2}$$

$$\dot{V} = V_c - V \tag{5.3}$$

$$\dot{\psi} = 5(\psi_c - \psi) \tag{5.4}$$

### *5.2.1 Equal Number of Agents and Goals*

In this simulation, the 3 goals are designated $G_1$, $G_2$, and $G_3$, and are as shown in Table 5.1. Two different cases will be tested here. The first will involve each of the agents beginning at different initial conditions. This should result in the agents being commanded to go to the goal that is closest to them in space. In the second case, all of the agents will begin with the same initial conditions, so the supervisor will have to allocate goals randomly.

Table 5.1: Multiagent Goals

| Goal | Coordinates |
|---|---|
| $G_1$ | $[x,y]$=$[0,0]$ |
| $G_2$ | $[x,y]$=$[3,6]$ |
| $G_3$ | $[x,y]$=$[-5,8]$ |

For the first case, the initial conditions of each agent will each be different from one another. They all begin with a speed of 0 m/s, and they all accelerate to 1 m/s with a velocity time constant of $\tau_V = 1$ sec. The other states begin with the values shown in Table 5.2.

Table 5.2: Homogeneous Multiagent System ICs - Sim 1

| Agent | $x_0$ (m) | $y_0$ (m) | $\psi_0$ (deg) |
|---|---|---|---|
| $A_1$ | $-5$ | $-5$ | 0 |
| $A_2$ | 0 | 4 | 90 |
| $A_3$ | 4 | $-6$ | 180 |

With these initial conditions, the supervisor agent $A_s$ used the learned dynamics to determine the proper agent-to-goal assignment for minimum time. The agents were then able to successfully navigate to their goals using the control policies and sample times learned previously. This is shown in Figures 5.3-5.6, which demonstrate that for homogeneous agents the supervisor chose the closest goal in space to each agent. The goal assignment determined by the supervisor is shown in Table 5.3.

Table 5.3: Homogeneous Multiagent Goal Assignment - Sim 1

| Goal | Agent |
|---|---|
| $G_1$ | $A_3$ |
| $G_2$ | $A_2$ |
| $G_3$ | $A_1$ |

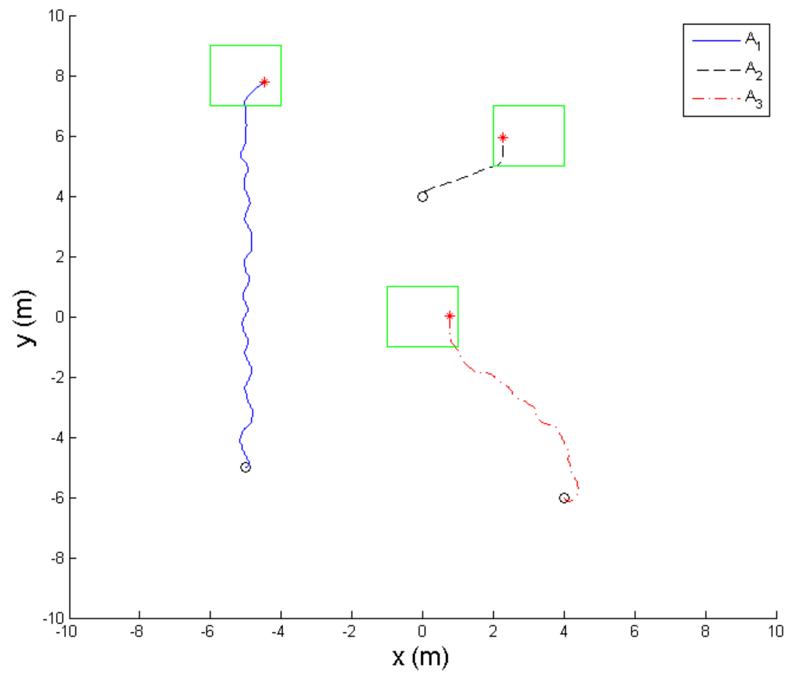


Figure 5.3: Homogeneous Multiagent Robot Paths - Sim 1

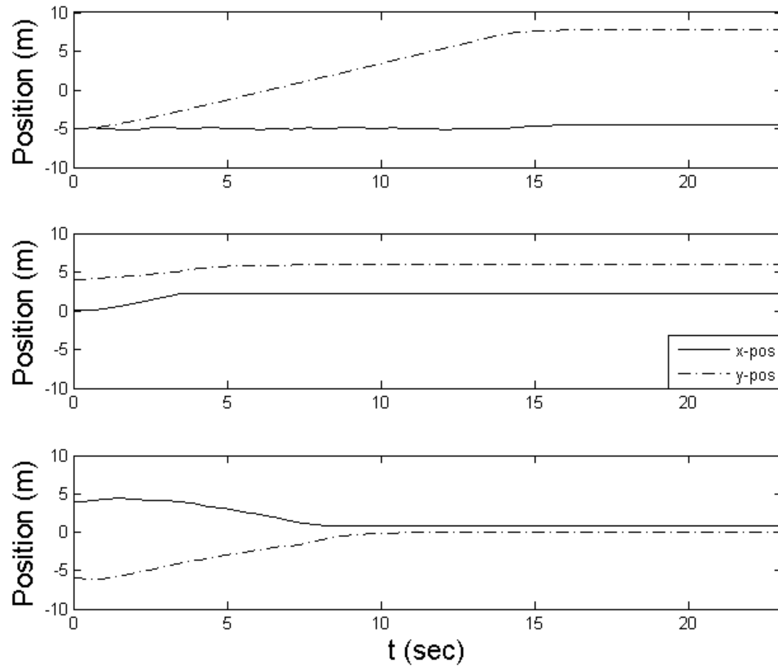


Figure 5.4: Homogeneous Multiagent Robot States - Sim 1

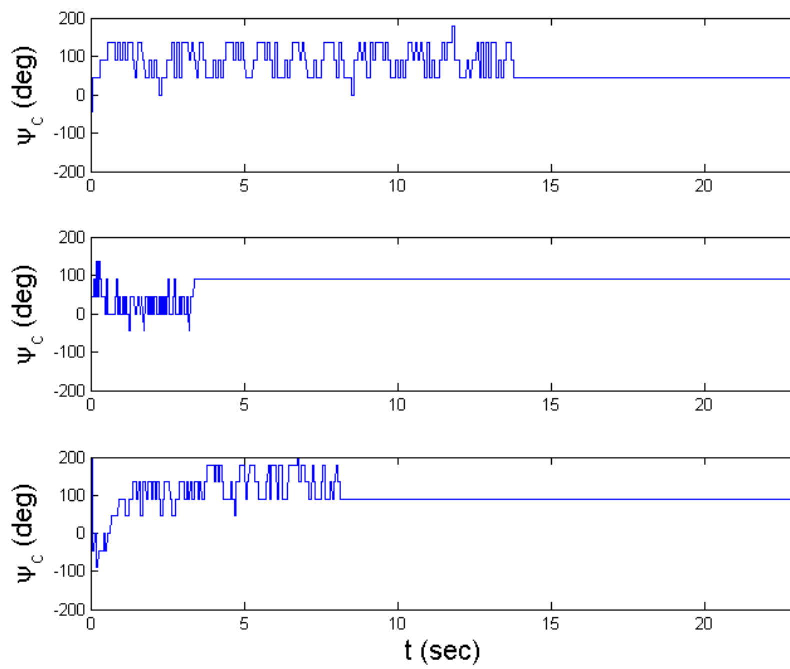


Figure 5.5: Homogeneous Multiagent Robot Command - Sim 1

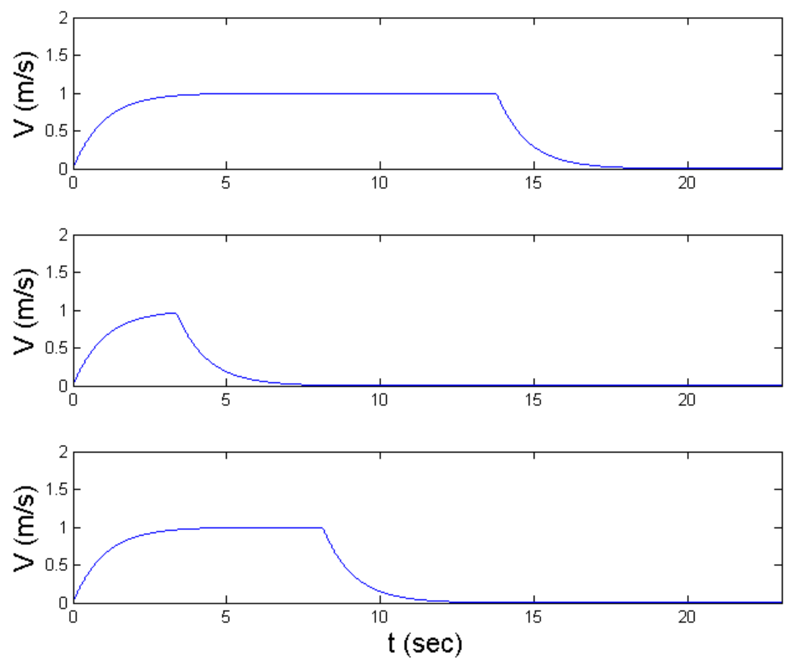


Figure 5.6: Homogeneous Multiagent Robot Speed - Sim 1

Figure 5.3 indicates that all agents were able to reach their designated goals within the 30 second time period allowed. Figure 5.4 shows that the agent that took longest to reach its goal was $A_1$, which took 15 seconds to reach goal $G_3$, so the full task achievement took 15 seconds. Figure 5.5 shows the heading angle command history for each of the agents, and it can be seen that they do not all have the same sampling rate. Agent $A_1$ had a sample time of $T = 0.08$ sec, $A_2$ sampled at an interval of $T = 0.04$ sec, and $A_3$ sampled every $T = 0.06$ sec. The speed time histories for each agent are shown in Figure 5.6, and show the first-order response of the robots to the commands of maximum speed and stopping.

Next, the same simulation was performed where all agents began with the exact same initial conditions. The initial conditions are as shown in Table 5.4. In this case, the supervisor should be forced to assign goals randomly, and that is indeed the case in practice. Figures 5.7-5.10 show how the simulation turned out for this case, and the goal assignment from $A_s$ is shown in Table 5.5.

Table 5.4: Homogeneous Multiagent System ICs - Sim 2

| Agent | $x_0$ (m) | $y_0$ (m) | $\psi_0$ (deg) |
|---|---|---|---|
| $A_1$ | $-5$ | $-5$ | 180 |
| $A_2$ | $-5$ | $-5$ | 180 |
| $A_3$ | $-5$ | $-5$ | 180 |

Table 5.5: Homogeneous Multiagent Goal Assignment - Sim 2

| Goal | Agent |
|---|---|
| $G_1$ | $A_2$ |
| $G_2$ | $A_3$ |
| $G_3$ | $A_1$ |

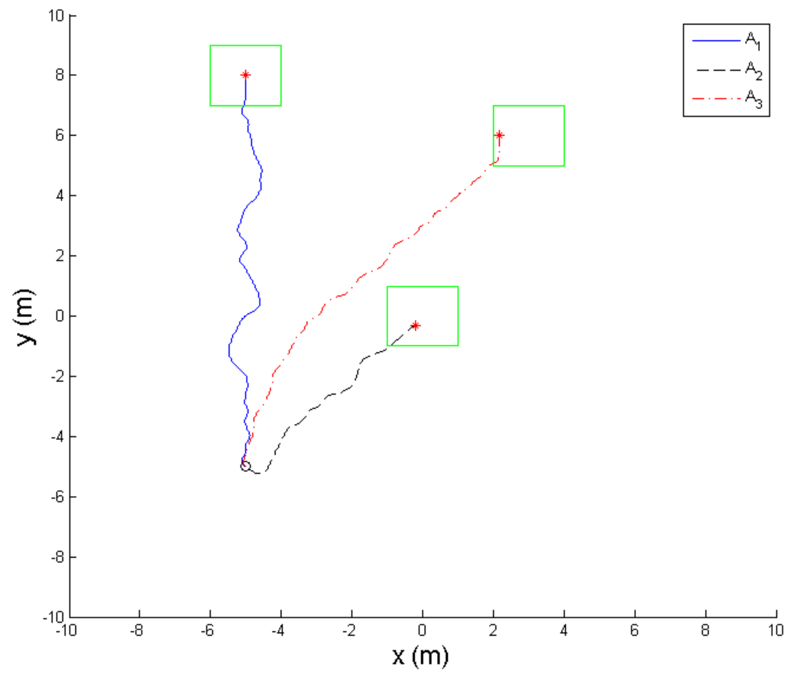


Figure 5.7: Homogeneous Multiagent Robot Paths - Sim 2

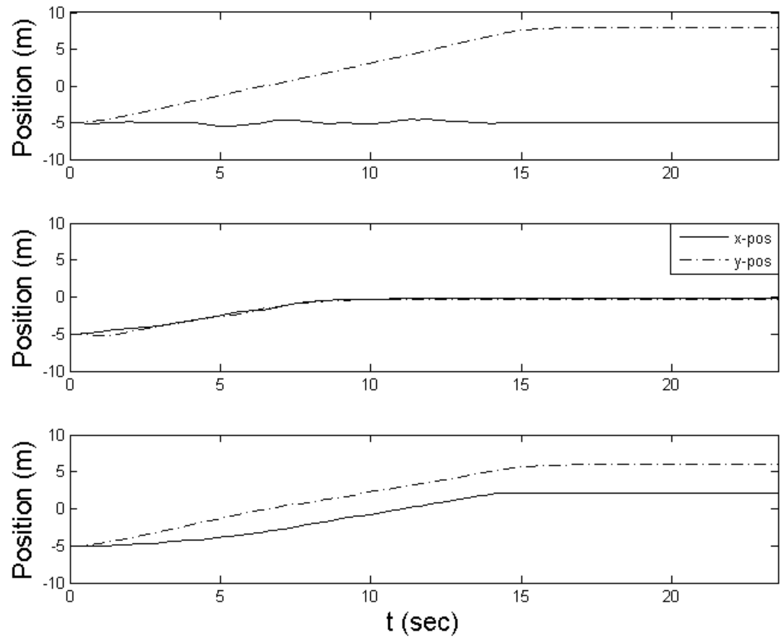


Figure 5.8: Homogeneous Multiagent Robot States - Sim 2

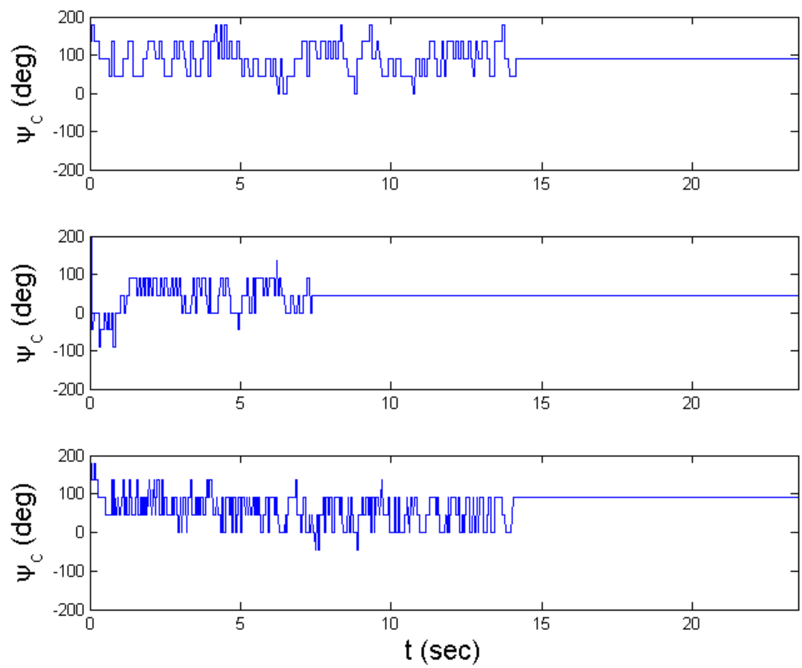


Figure 5.9: Homogeneous Multiagent Robot Command - Sim 2

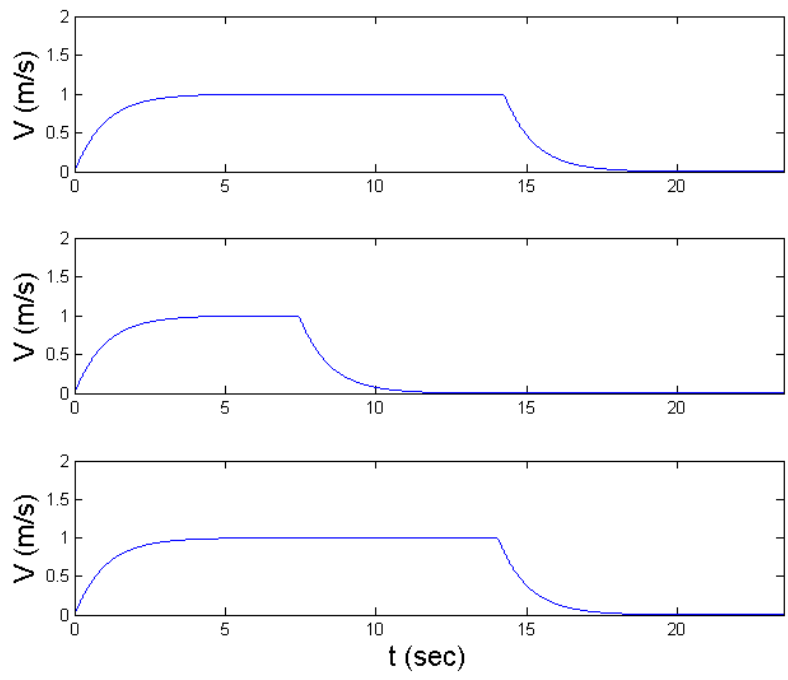


Figure 5.10: Homogeneous Multiagent Robot Speed - Sim 2

### *5.2.2 Fewer Goals than Agents*

This simulation shows how the supervisor can command an agent to stay in its initial state in the event that there are fewer goals than agents. Here, there are 3 agents but only 2 goals to achieve. Only 1 agent per goal is needed, so the supervisor determines which agents should be chosen and allocates the proper goals to them. The agent that is not chosen for a goal will be commanded to remain still in its initial state.

For this case, the two goals that are being investigated are $G_1$ and $G_2$ from Table 5.1. As before, two different test cases will be considered. The first case involves having each of the agents beginning with different initial conditions. The initial conditions are as shown in Table 5.6.

Table 5.6: Homogeneous Multiagent System ICs - Sim 3

| Agent | $x_0$ (m) | $y_0$ (m) | $\psi_0$ (deg) |
|---|---|---|---|
| $A_1$ | 0 | 4 | 90 |
| $A_2$ | $-6$ | 2 | $-90$ |
| $A_3$ | 6 | $-2$ | 0 |

Under the initial conditions in Table 5.6, the supervisor determined that the best task allocation for minimum time was to assign each agent to the closest goal, leaving agent $A_3$ without an assignment. Agent $A_3$ is tasked to wait as a result while agents $A_1$ and $A_2$ achieve their assigned goals, which are shown in Table 5.7. This is demonstrated in Figures 5.11-5.14.

Table 5.7: Homogeneous Multiagent Goal Assignment - Sim 3

| Goal | Agent |
|---|---|
| $G_1$ | $A_2$ |
| $G_2$ | $A_1$ |

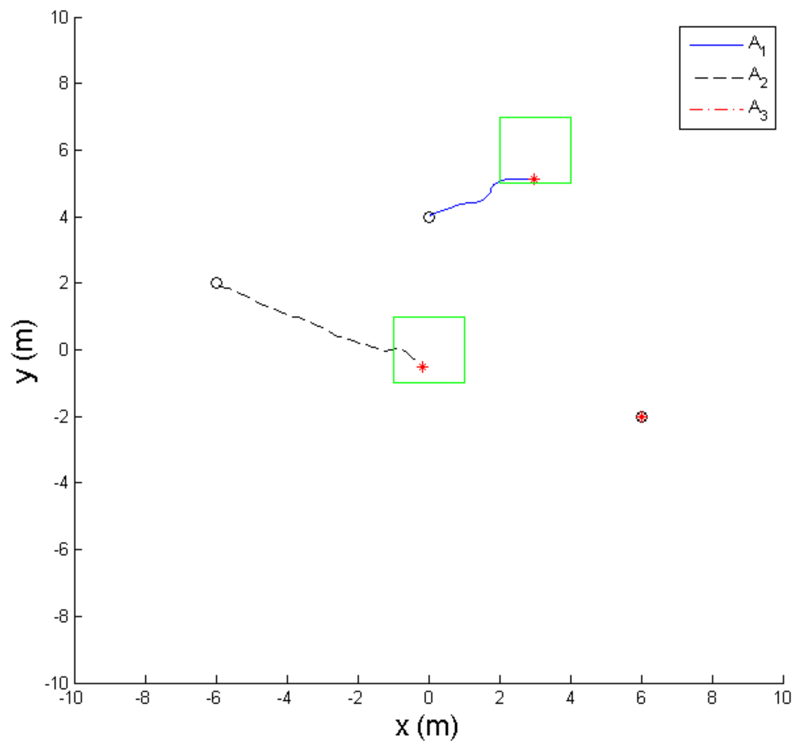


Figure 5.11: Homogeneous Multiagent Robot Paths - Sim 3

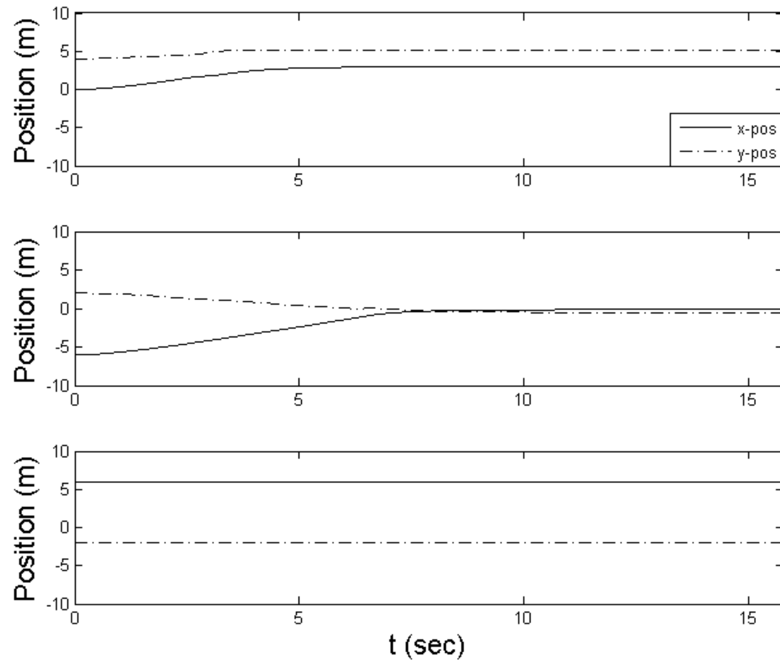


Figure 5.12: Homogeneous Multiagent Robot States - Sim 3

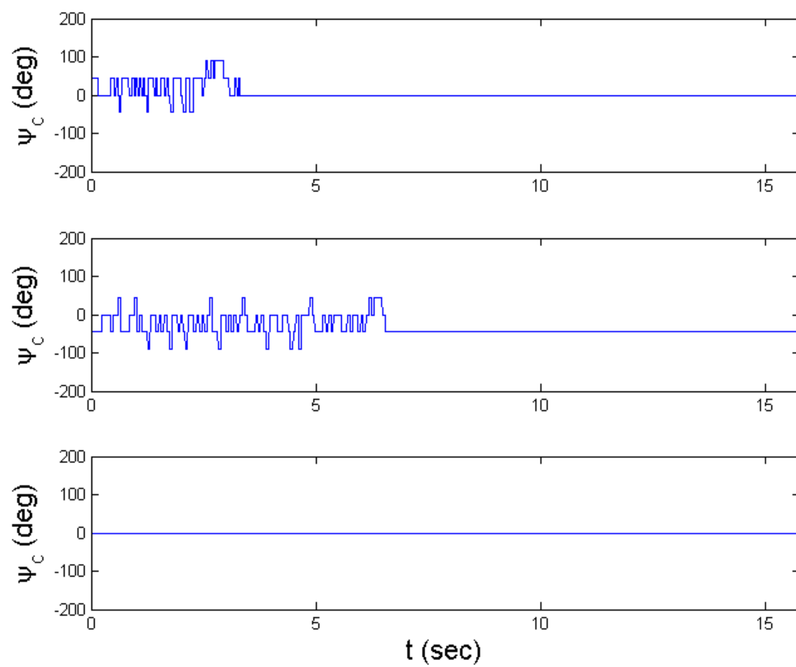


Figure 5.13: Homogeneous Multiagent Robot Command - Sim 3

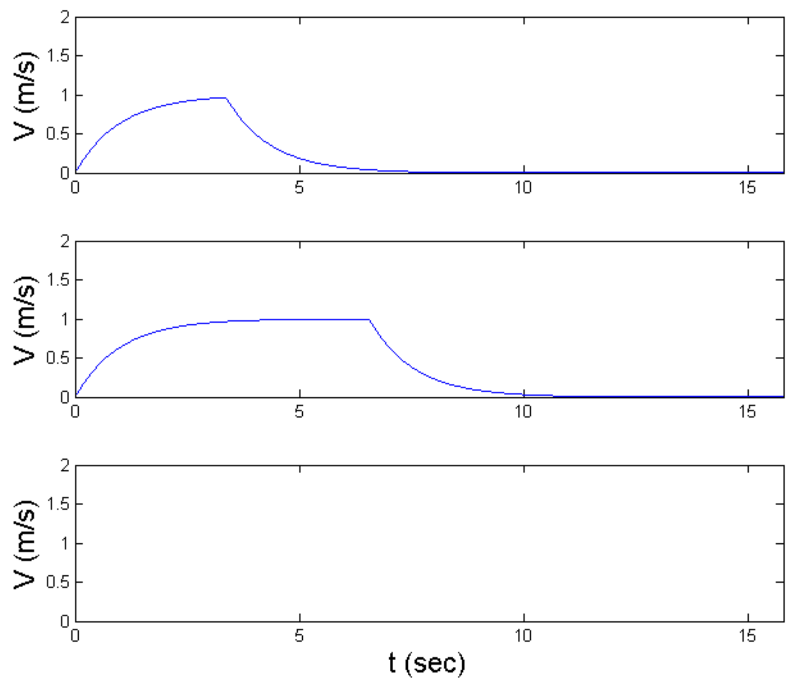


Figure 5.14: Homogeneous Multiagent Robot Speed - Sim 3

For the second case, all 3 agents began in the same initial conditions. The initial conditions for these agents are reported in Table 5.8. As before, since all agents have identical dynamics the supervisor chooses goals randomly, and the goal assignments are shown in Table 5.9. Figures 5.15-5.18 show the agents achieving the specified goals.

Table 5.8: Homogeneous Multiagent System ICs - Sim 4

| Agent | $x_0$ (m) | $y_0$ (m) | $\psi_0$ (deg) |
|---|---|---|---|
| $A_1$ | $-6$ | $-4$ | 180 |
| $A_2$ | $-6$ | $-4$ | 180 |
| $A_3$ | $-6$ | $-4$ | 180 |

Table 5.9: Homogeneous Multiagent Goal Assignment - Sim 4

| Goal | Agent |
|---|---|
| $G_1$ | $A_3$ |
| $G_2$ | $A_1$ |

These examples show that the supervisor is capable of assigning agents to goals, and for the case of having all agents be identical the minimum-time decisions are equivalent to the minimum-distance decisions. The individual sample times and control policies for each agent are used to achieve the particular goal assigned to them, and they are all capable of doing so successfully in each case. It is now necessary to consider the scenario where the agents do not have the same dynamics, and see how the learned dynamics for each agent affects the supervisor's goal allocation decisions.

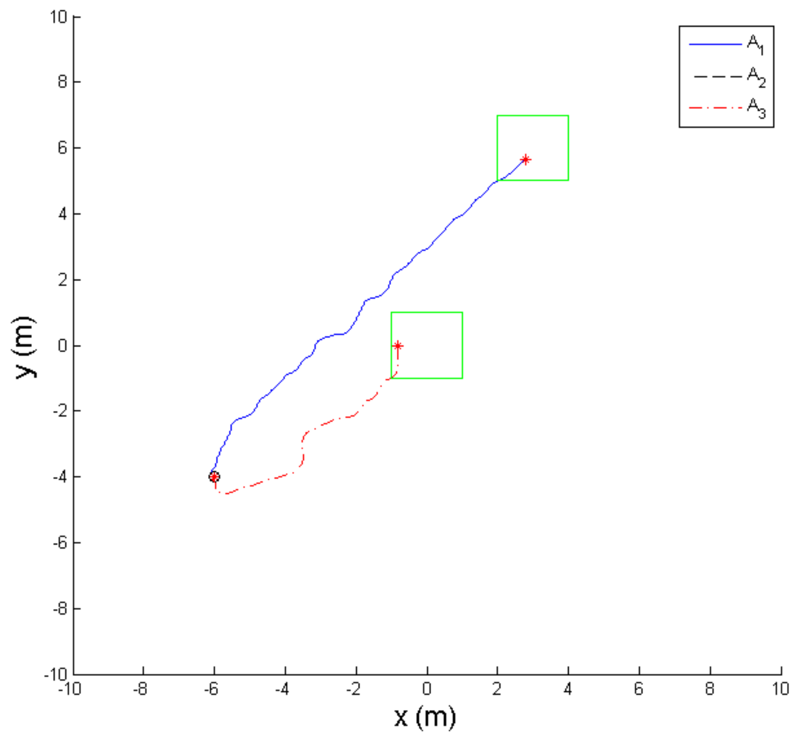


Figure 5.15: Homogeneous Multiagent Robot Paths - Sim 4

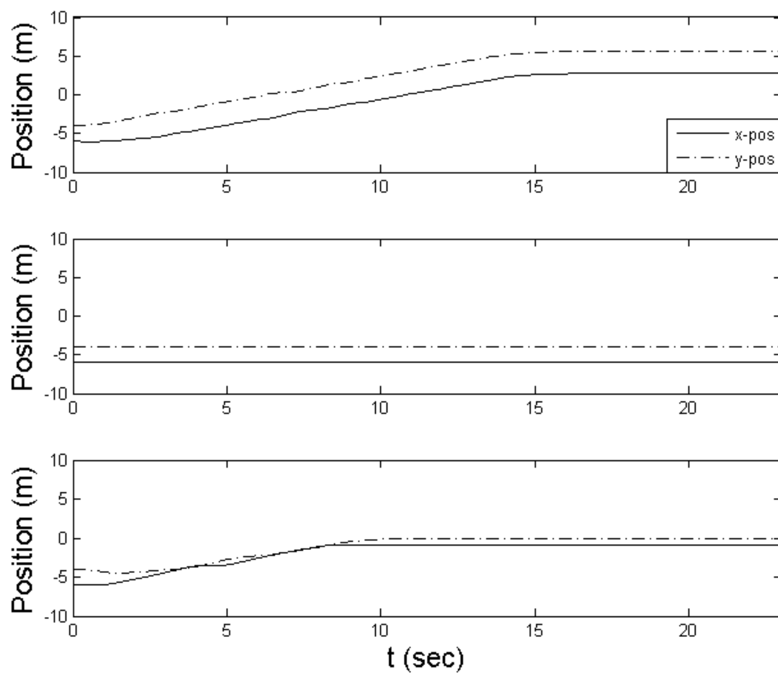


Figure 5.16: Homogeneous Multiagent Robot States - Sim 4

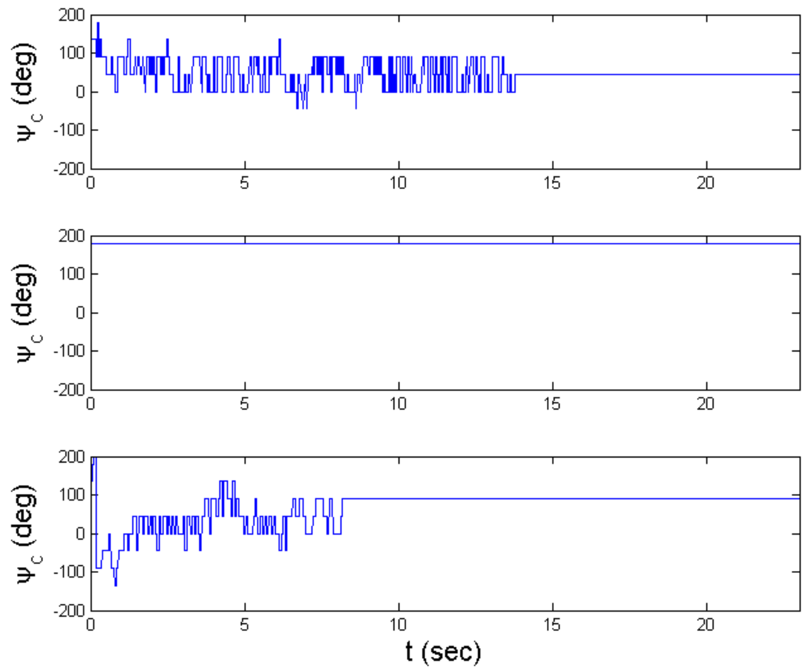


Figure 5.17: Homogeneous Multiagent Robot Command - Sim 4

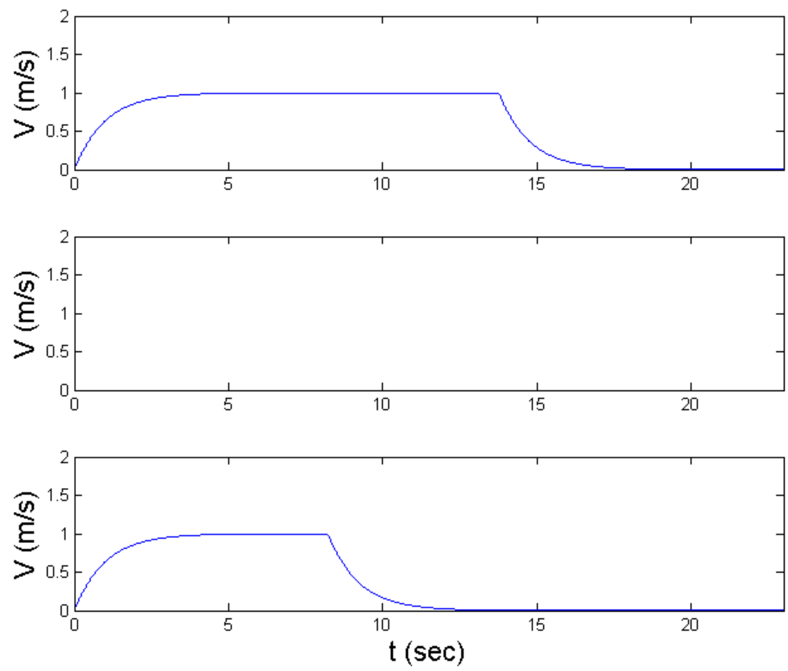


Figure 5.18: Homogeneous Multiagent Robot Speed - Sim 4

## 5.3 Heterogeneous Agents Examples

In this section, the case of a multiagent system with heterogeneous dynamics is considered. Just as before, there are 3 different agents with 3 different goals. The agents each have different dynamics, so the task allocation from the supervisor agent will be based on the dynamics of each individual agent considered. In this multiagent system, the 3 agents have different dynamics that exhibit either first- or second-order behavior.

In these examples, each agent begins at rest and accelerates to the maximum speed with first-order dynamics governing the speed changes. Once an agent reaches the goal region, it is commanded to decelerate to rest. All agents in this simulation have a speed time constant of $\tau_V = 1$ sec that is unknown to the learner, but they each have different maximum speeds which are already known. The rotational dynamics for each agent are varied. It is known in advance whether the dynamics are first- or second-order, but the parameters governing the rotational equations of motion are unknown to the learner. The general equations of motion for the first-order rotation are shown in Equations 5.5-5.8.

$$\dot{x} = V \cos\psi \tag{5.5}$$

$$\dot{y} = V \sin\psi \tag{5.6}$$

$$\dot{V} = (V_c - V)\tau_V^{-1} \tag{5.7}$$

$$\dot{\psi} = (\psi_c - \psi)\tau_\psi^{-1} \tag{5.8}$$

For the case of agents with second-order dynamics in the rotation, the heading angle equation of motion shown in Equation 5.8 must be replaced with the second

derivative. This is shown in Equation 5.9.

$$\ddot{\psi} = -2\zeta_{\psi}\omega_{n,\psi}\dot{\psi} + \omega_{n,\psi}^2(\psi_c - \psi) \tag{5.9}$$

The 3 different agents discussed will be labeled as agents $A_1$-$A_3$. $A_1$ is a robot with first-order dynamics in rotation that has a time constant of $\tau_{\psi} = 0.2$ sec. $A_1$ reaches a maximum speed of 1 m/s from rest according to the first-order time constant of $\tau_V = 1$ sec. The resulting equations of motion governing $A_1$ are shown in Equations 5.10-5.13.

$$\dot{x} = V\cos\psi \tag{5.10}$$

$$\dot{y} = V\sin\psi \tag{5.11}$$

$$\dot{V} = V_c - V \tag{5.12}$$

$$\dot{\psi} = 5(\psi_c - \psi) \tag{5.13}$$

$A_2$ has similar speed dynamics with the same first-order time constant of $\tau_V = 1$ sec, but the maximum speed of $A_2$ is faster with $V_{max} = 2$ m/s. $A_2$ also differs in rotation with second-order heading angle dynamics. The rotational dynamics are governed by a natural frequency of $\omega_{n,\psi} = 6$ rad/s and an underdamped damping ratio of $\zeta_{\psi} = 0.8$. The equations of motion for $A_2$ are shown in Equations 5.14-5.17.

$$\dot{x} = V \cos \psi \tag{5.14}$$

$$\dot{y} = V \sin \psi \tag{5.15}$$

$$\dot{V} = V_c - V \tag{5.16}$$

$$\ddot{\psi} = -9.6\dot{\psi} + 36(\psi_c - \psi) \tag{5.17}$$

$A_3$ also has a first-order speed time constant of $\tau_V = 1$ sec, but it has the slowest maximum speed with $V_{max} = 0.5$ m/s. $A_3$ has second-order rotational dynamics similar to $A_2$, but the natural frequency is $\omega_{n,\psi} = 10$ rad/s and it is overdamped with $\zeta_\psi = 1.2$. The equations of motion for $A_2$ are shown in Equations 5.18-5.21.

$$\dot{x} = V \cos \psi \tag{5.18}$$

$$\dot{y} = V \sin \psi \tag{5.19}$$

$$\dot{V} = V_c - V \tag{5.20}$$

$$\ddot{\psi} = -24\dot{\psi} + 100(\psi_c - \psi) \tag{5.21}$$

The SDQL algorithm was used on all 3 agents to determine the control policy for each agent to reach the specified goals, and to determine the best sampling rate for doing so by maximizing the quadratic sample time reward. The agents also experienced the dynamics learning algorithms, using FODL on $A_1$ and SODL on $A_2$ and $A_3$. For all of the agents, these algorithms were used online during the learning process, according to Algorithms 4.2 and 4.4. After running these algorithms for each of the agents, the determined sample times and dynamics approximations for each agent were determined to be as reported in Tables 5.10-5.12.

Using the dynamics learned as shown in Tables 5.10-5.12, the multiagent system

Table 5.10: Agent 1 Learned Dynamics

| Parameter | $G_1$ | $G_2$ | $G_3$ |
|---|---|---|---|
| $T(sec)$ | 0.08 | 0.04 | 0.06 |
| $\tau_V(sec)$ | 1.0 | 1.0 | 1.0 |
| $\tau_\psi(sec)$ | 0.2 | 0.2 | 0.2 |

Table 5.11: Agent 2 Learned Dynamics

| Parameter | $G_1$ | $G_2$ | $G_3$ |
|---|---|---|---|
| $T(sec)$ | 0.10 | 0.08 | 0.04 |
| $\tau_V(sec)$ | 1.0 | 1.0 | 1.0 |
| $\omega_{n,\psi}(rad/s)$ | 6 | 6 | 6 |
| $\zeta_\psi$ | 0.8 | 0.8 | 0.8 |

can be simulated. The system is hierarchical with a single supervisory agent that utilizes the dynamics information in allocating goals to agents according to Figure 5.1. The low-level agents, designated Agents $A_1$-$A_3$, use the learned control policies and sample times to sample the environment and direct themselves to their designated goals.

### *5.3.1 Equal Number of Agents and Goals*

In the first scenario, there will be a goal for every agent simulated. The first case involves having each agent begin at a different initial condition. Since the agents have different dynamics now, the supervisor will have to consider the learned dynamics

Table 5.12: Agent 3 Learned Dynamics

| Parameter | $G_1$ | $G_2$ | $G_3$ |
|---|---|---|---|
| $T(sec)$ | 0.06 | 0.08 | 0.02 |
| $\tau_V(sec)$ | 1.0 | 1.0 | 1.0 |
| $\omega_{n,\psi}(rad/s)$ | 10 | 10 | 10 |
| $\zeta_\psi$ | 1.2 | 1.2 | 1.2 |

in determining which agent to allocate to each task, and minimum distance will not necessarily be the same assignment as minimum time. The initial conditions for this first case are shown in Table 5.13. The goals are labeled as shown in Table 5.1, and the agent-to-goal task assignment determined by the supervisory agent is shown in Table 5.14.

Table 5.13: Heterogeneous Multiagent System ICs - Sim 1

| Agent | $x_0$ (m) | $y_0$ (m) | $\psi_0$ (deg) |
|---|---|---|---|
| $A_1$ | $-6$ | $-7$ | $-90$ |
| $A_2$ | 0 | 5 | 90 |
| $A_3$ | 5 | $-3$ | 0 |

Table 5.14: Heterogeneous Multiagent Goal Assignment - Sim 1

| Goal | Agent |
|---|---|
| $G_1$ | $A_3$ |
| $G_2$ | $A_2$ |
| $G_3$ | $A_1$ |

Once the supervisor agent completed its task allocation, the agents in the system were simulated using all of the learned information to reach their goals. The results of this simulation shown in Figures 5.19-5.22 indicate that all algorithms were successful in contributing to the multiagent system control problem.

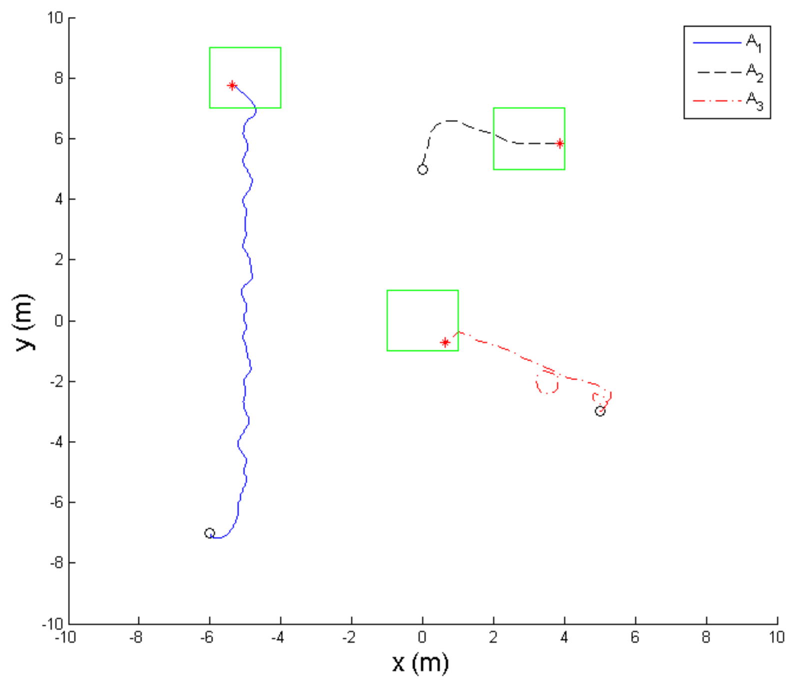


Figure 5.19: Heterogeneous Multiagent Robot Paths - Sim 1

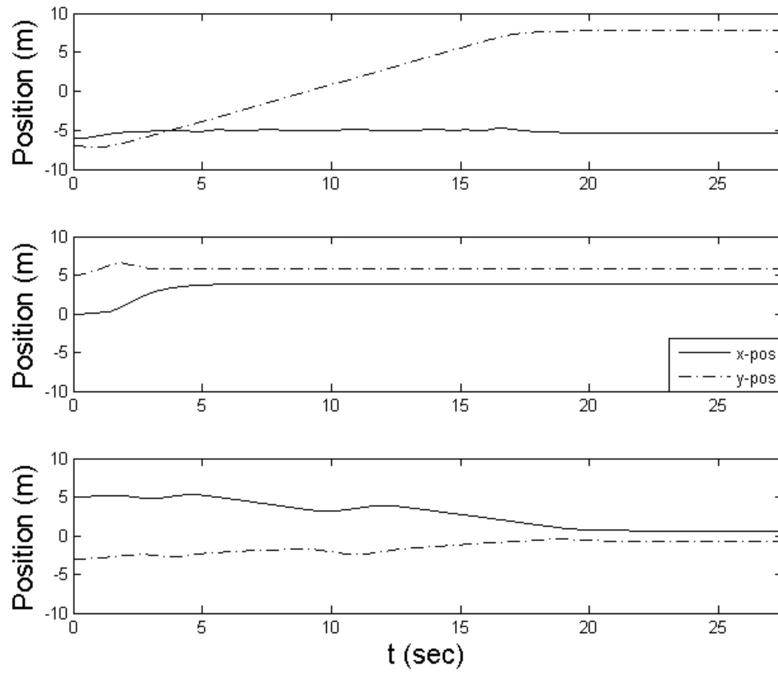


Figure 5.20: Heterogeneous Multiagent Robot States - Sim 1

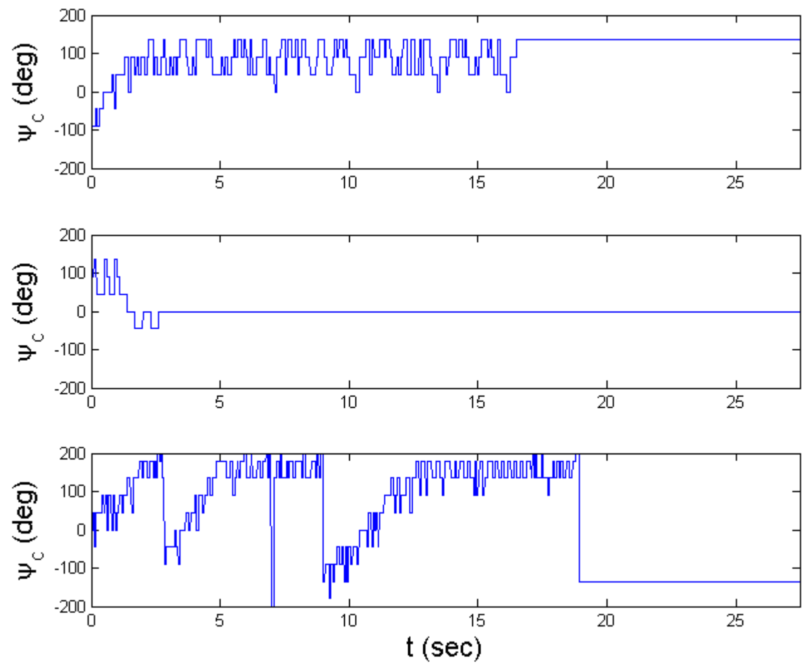


Figure 5.21: Heterogeneous Multiagent Robot Command - Sim 1

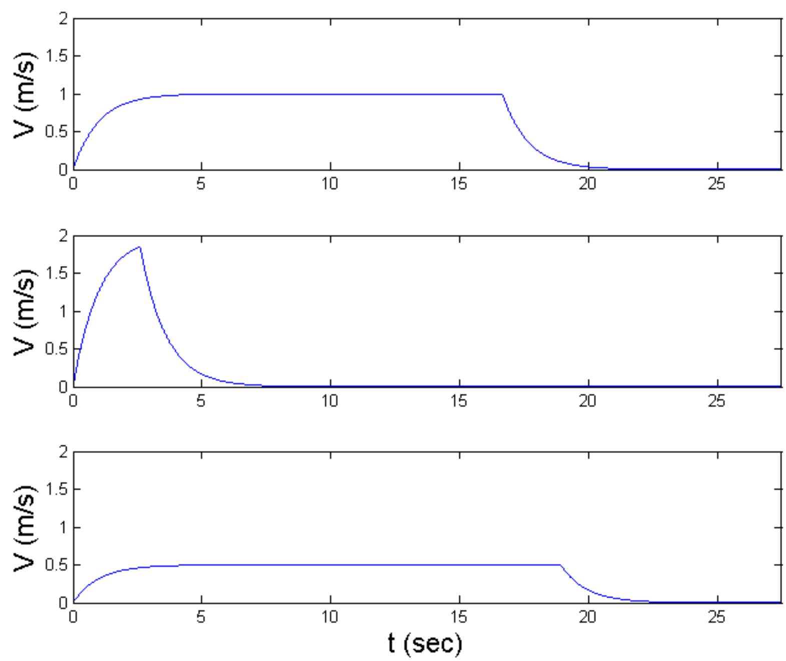


Figure 5.22: Heterogeneous Multiagent Robot Speed - Sim 1

Figure 5.19 indicates that all agents were able to reach their designated goals within the 30 second time period allowed. Figure 5.20 shows that the agent that took longest to reach its goal was $A_1$, which took 16 seconds to reach goal $G_3$, so the full task achievement took 16 seconds. Figure 5.21 shows the heading angle command history for each of the agents, and it can be seen that they do not all have the same sampling rate. Agent $A_1$ had a sample time of $T = 0.08$ sec, $A_2$ sampled at an interval of $T = 0.10$ sec, and $A_3$ sampled every $T = 0.06$ sec. The speed time histories for each agent are shown in Figure 5.22, and show the first-order response of the robots to the commands of maximum and minimum speed.

Upon examining Figure 5.19, it can be seen that agent $A_3$ reinforced undesirable behavior in the form of loops. This is an indication that this was an early path that received positive reinforcement, and with further learning this learned behavior may be overcome. To demonstrate this, agent $A_3$ received an additional 10,000 learning episodes to see if it could learn a better path. Figures 5.23-5.26 show that this is indeed the case. Agents $A_1$ and $A_2$ retain the same learning and therefore follow the same path as before, but agent $A_3$ has learned to follow a better path to its goal.

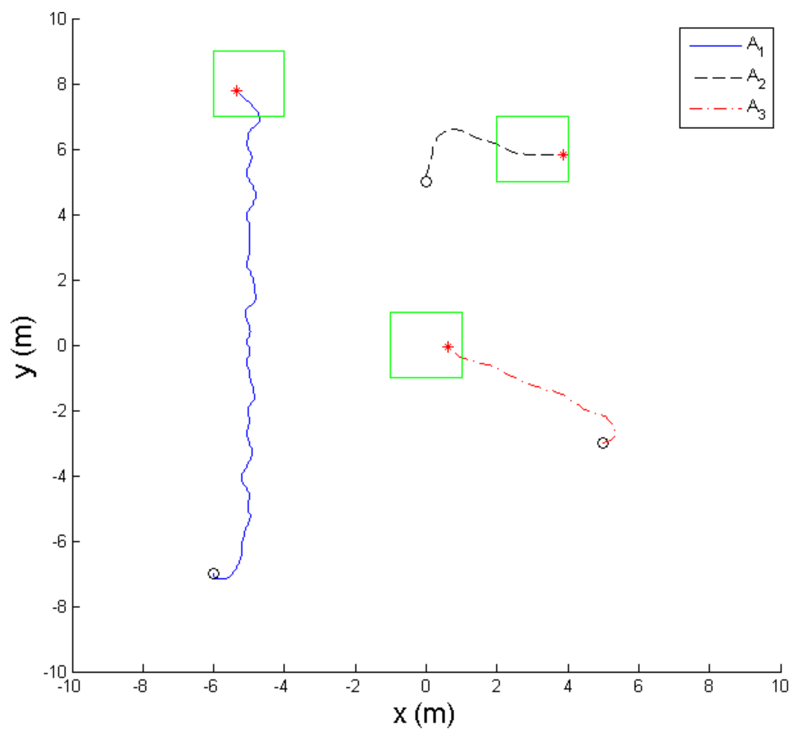


Figure 5.23: Heterogeneous Robot Paths - Sim 1 (More Learning)

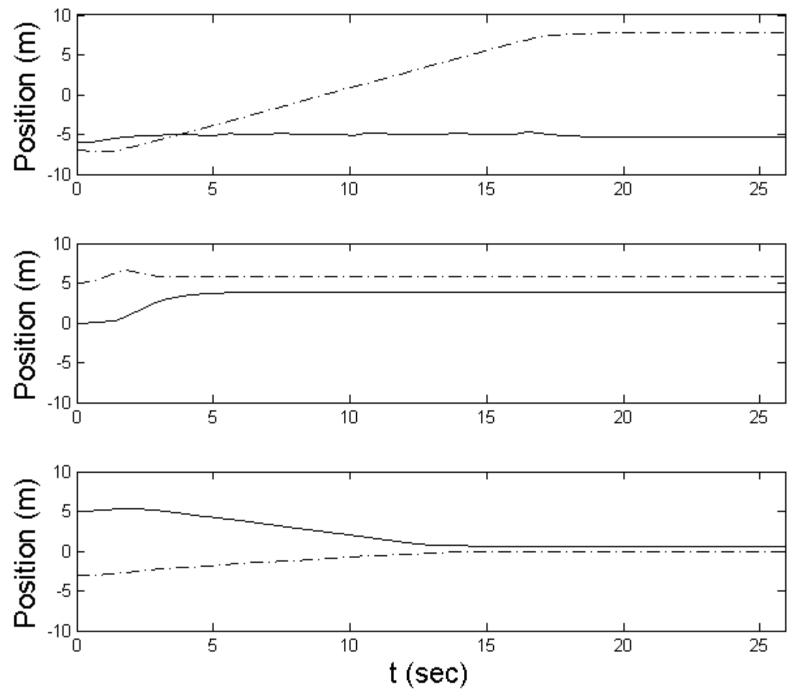


Figure 5.24: Heterogeneous Robot States - Sim 1 (More Learning)

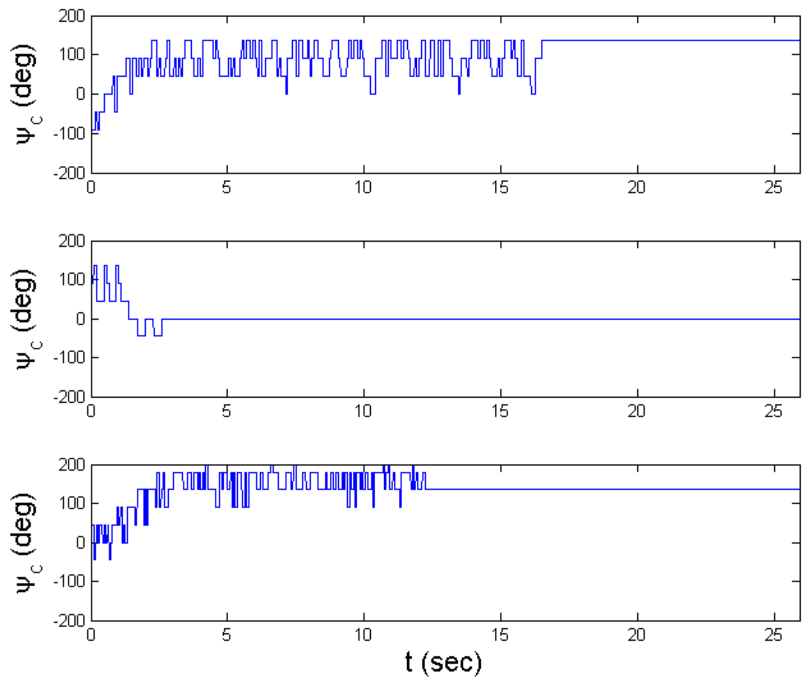


Figure 5.25: Heterogeneous Robot Command - Sim 1 (More Learning)

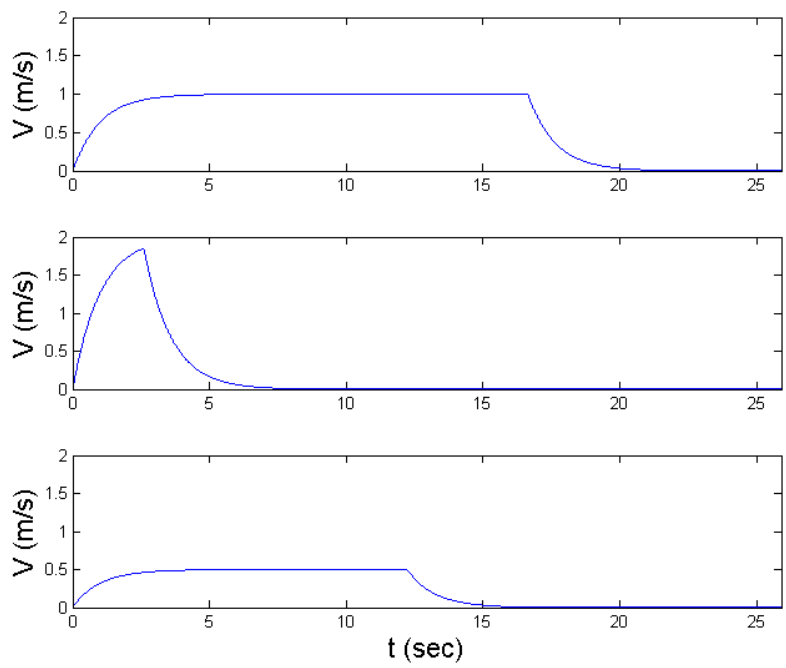


Figure 5.26: Heterogeneous Robot Speed - Sim 1 (More Learning)

The next case involves having all 3 agents begin at the same initial conditions. In this scenario, the different dynamics will make a very big difference in how goals are allocated to agents. The initial conditions for all of the agents are shown in Table 5.15, and the resulting goal assignments from agent $A_s$ are shown in Table 5.16.

Table 5.15: Heterogeneous Multiagent System ICs - Sim 2

| Agent | $x_0$ (m) | $y_0$ (m) | $\psi_0$ (deg) |
|---|---|---|---|
| $A_1$ | 0 | 4 | 90 |
| $A_2$ | 0 | 4 | 90 |
| $A_3$ | 0 | 4 | 90 |

Table 5.16: Heterogeneous Multiagent Goal Assignment - Sim 2

| Goal | Agent |
|---|---|
| $G_1$ | $A_2$ |
| $G_2$ | $A_3$ |
| $G_3$ | $A_1$ |

In this case, the slowest agent, $A_3$, was assigned to the easiest target, $G_2$. For the other 2 goals, due to the initial orientation it was deemed that goal $G_1$ would need to be reached by the fastest agent, $A_2$. Agent $A_1$ was then left with the task of achieving goal $G_3$. Figures 5.30-5.27 show the successful simulation of this test case.

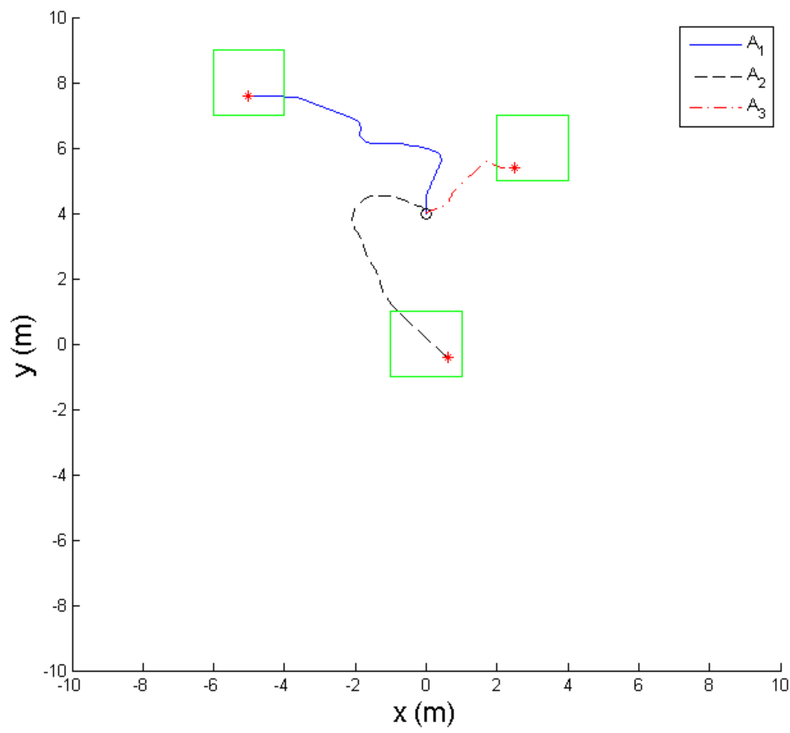


Figure 5.27: Heterogeneous Multiagent Robot Paths - Sim 2

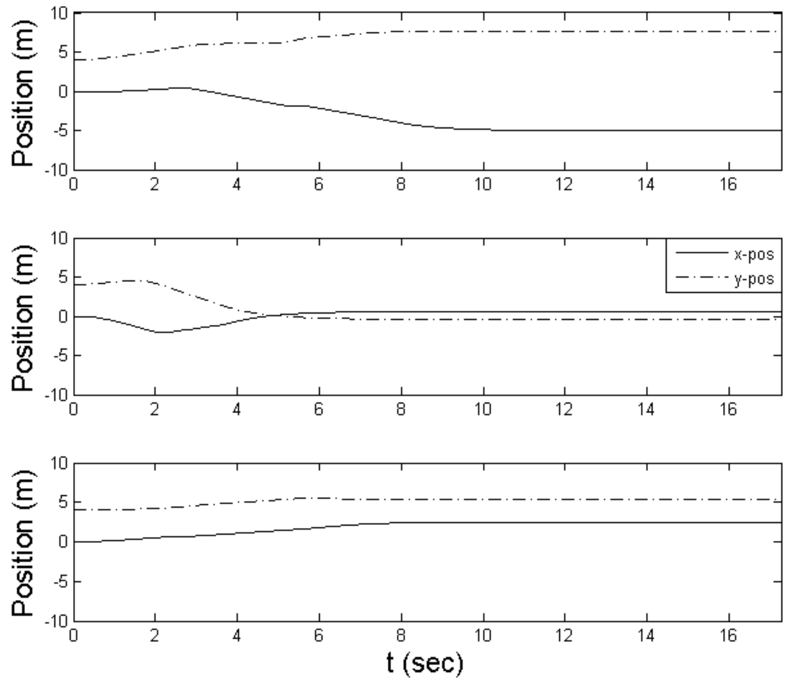


Figure 5.28: Heterogeneous Multiagent Robot States - Sim 2

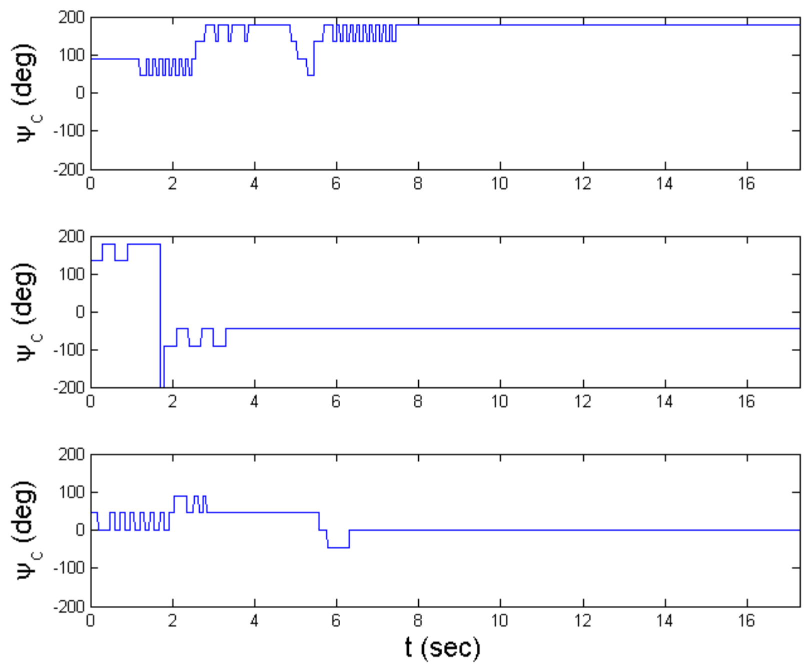


Figure 5.29: Heterogeneous Multiagent Robot Command - Sim 2

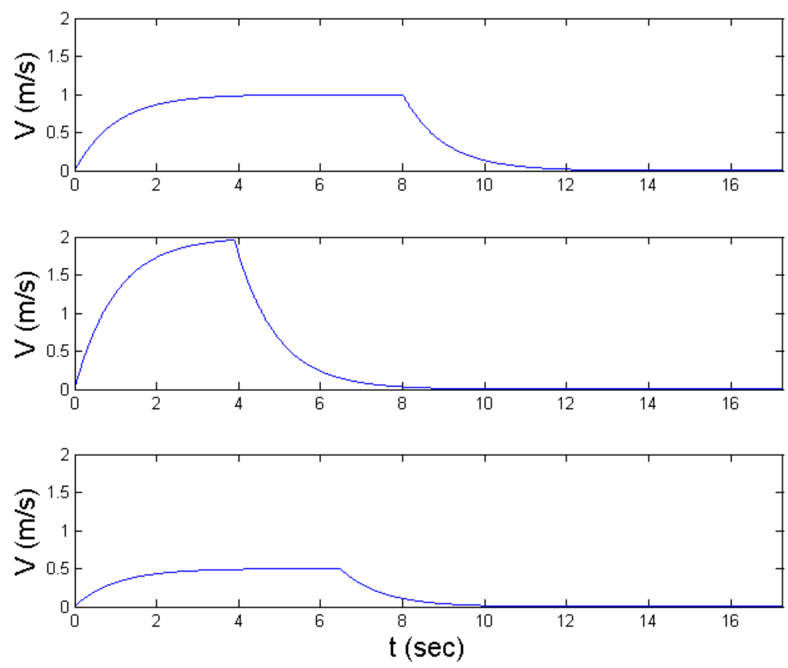


Figure 5.30: Heterogeneous Multiagent Robot Speed - Sim 2

### *5.3.2 Fewer Goals than Agents*

Now the case of assigning goals to agents when there is a surplus of agents must be considered. In these cases, there will be the same 3 agents and only goals $G_1$ and $G_2$ as before. The first case will demonstrate the task allocation in the situation where initial conditions are not the same. Since not every agent must go to a goal, the learned dynamics will need to be considered by agent $A_s$ to ensure minimum time, not minimum distance. The initial conditions for this case are shown in Table 5.17, and the goal assignments determined by the supervisor, $A_s$, are shown in Table 5.18.

Table 5.17: Heterogeneous Multiagent System ICs - Sim 3

| Agent | $x_0$ (m) | $y_0$ (m) | $\psi_0$ (deg) |
|---|---|---|---|
| $A_1$ | $-5$ | $-5$ | $-90$ |
| $A_2$ | 0 | 5 | 180 |
| $A_3$ | 3 | $-4$ | 0 |

Table 5.18: Heterogeneous Multiagent Goal Assignment - Sim 3

| Goal | Agent |
|---|---|
| $G_1$ | $A_1$ |
| $G_2$ | $A_2$ |

In this case, it is seen that although agent $A_3$ is closer to goal $G_1$ than agent $A_1$, it is actually agent $A_1$ that gets the goal assignment due to faster dynamics. Figures 5.34-5.31 show the successful navigation of each agent to its goal for this case.

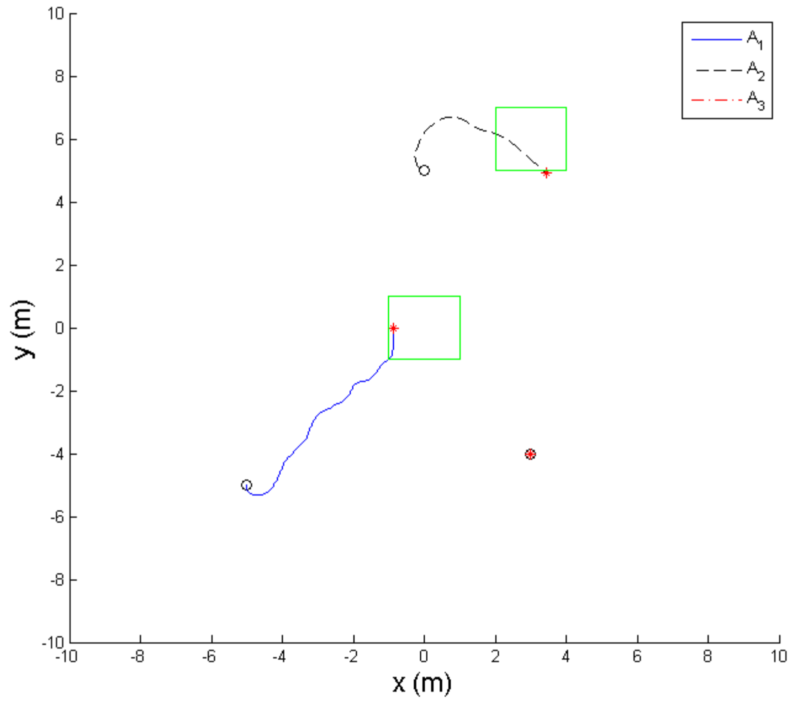

Figure 5.31: Heterogeneous Multiagent Robot Paths - Sim 3

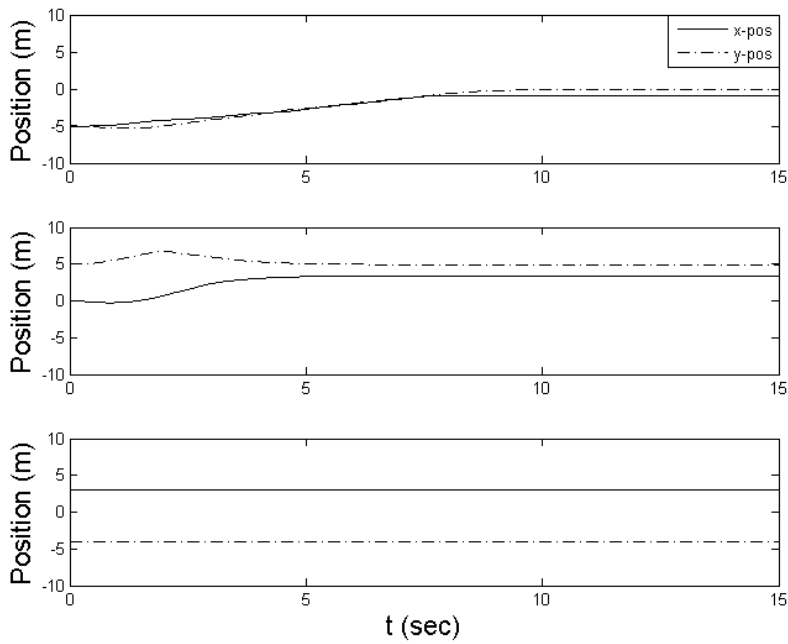

Figure 5.32: Heterogeneous Multiagent Robot States - Sim 3

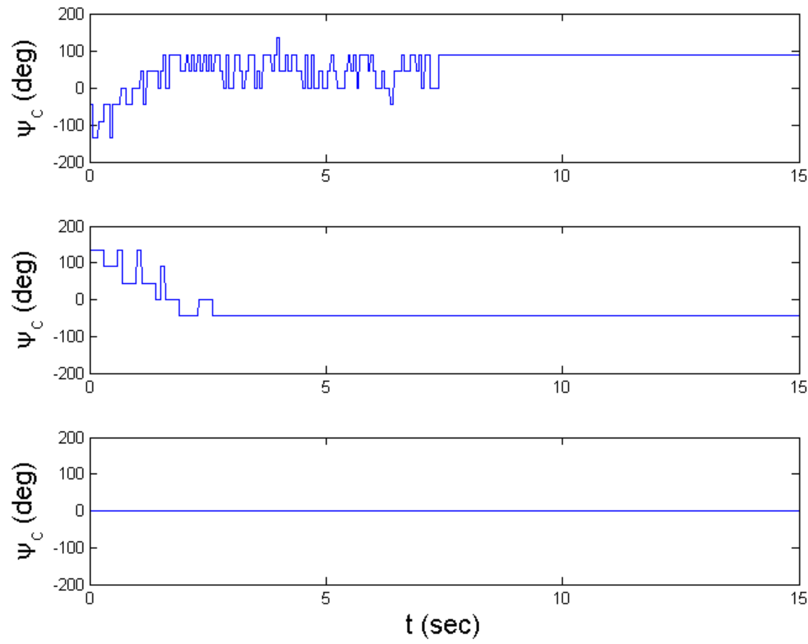


Figure 5.33: Heterogeneous Multiagent Robot Command - Sim 3

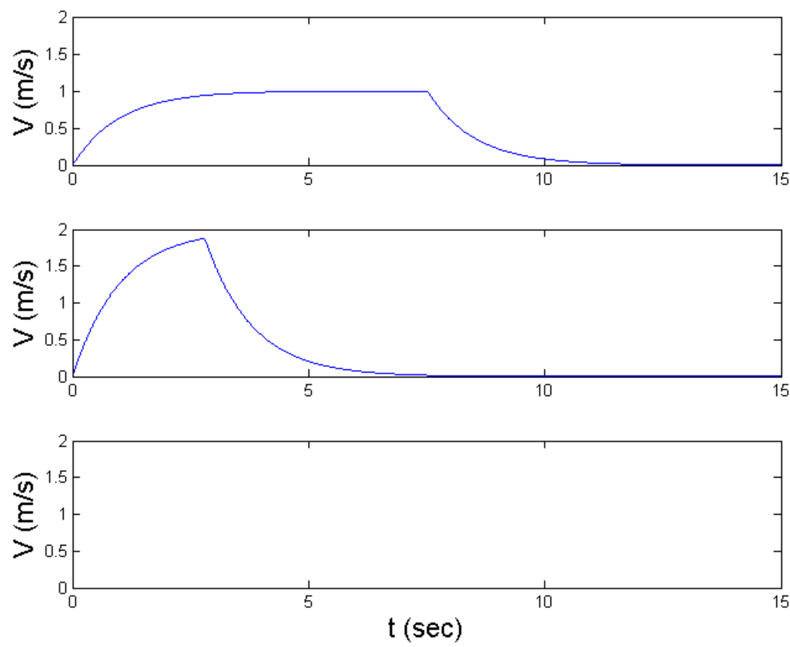


Figure 5.34: Heterogeneous Multiagent Robot Speed - Sim 3

For the next case, the scenario of having all 3 agents begin with the same initial conditions but only have 2 possible goals will be investigated. Here, all of the agents begin with the initial conditions shown in Table 5.19. The supervisor assigns goals based on their learned dynamics, and this task allocation is shown in Table 5.20.

Table 5.19: Heterogeneous Multiagent System ICs - Sim 4

| Agent | $x_0$ (m) | $y_0$ (m) | $\psi_0$ (deg) |
|---|---|---|---|
| $A_1$ | $-5$ | $-5$ | 180 |
| $A_2$ | $-5$ | $-5$ | 180 |
| $A_3$ | $-5$ | $-5$ | 180 |

Table 5.20: Heterogeneous Multiagent Goal Assignment - Sim 4

| Goal | Agent |
|---|---|
| $G_1$ | $A_1$ |
| $G_2$ | $A_2$ |

Supervisor $A_s$ then uses the dynamics to determine which agents will be assigned to which goals. Intuitively, the farthest goal should receive the fastest agent and the slowest agent should have no assignment. Indeed, it is verified that this is the case. Figures 5.35-5.38 show the simulation of these agents as they achieve their designated tasks. These figures show that each of the agents are able to achieve their designated goal successfully.

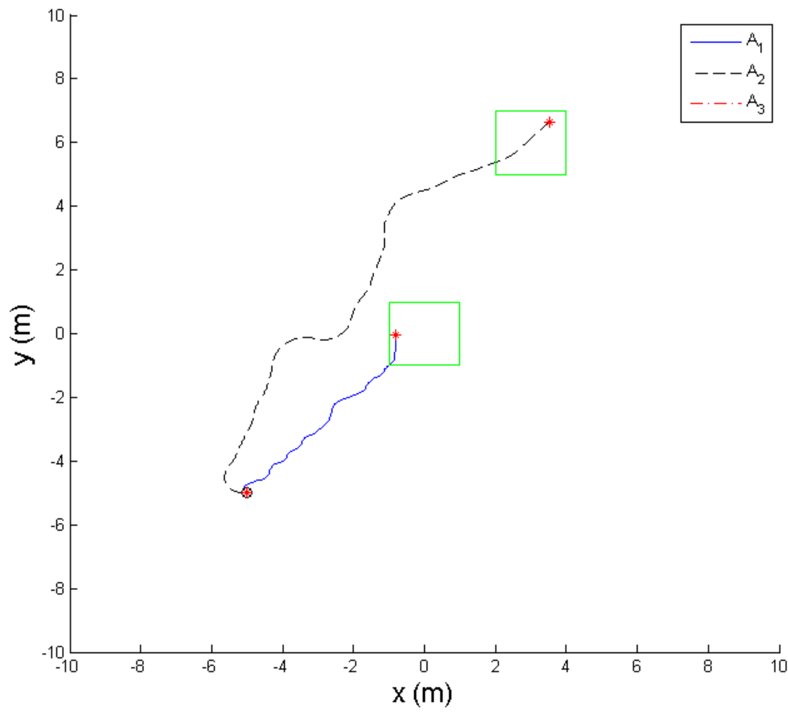


Figure 5.35: Heterogeneous Multiagent Robot Paths - Sim 4

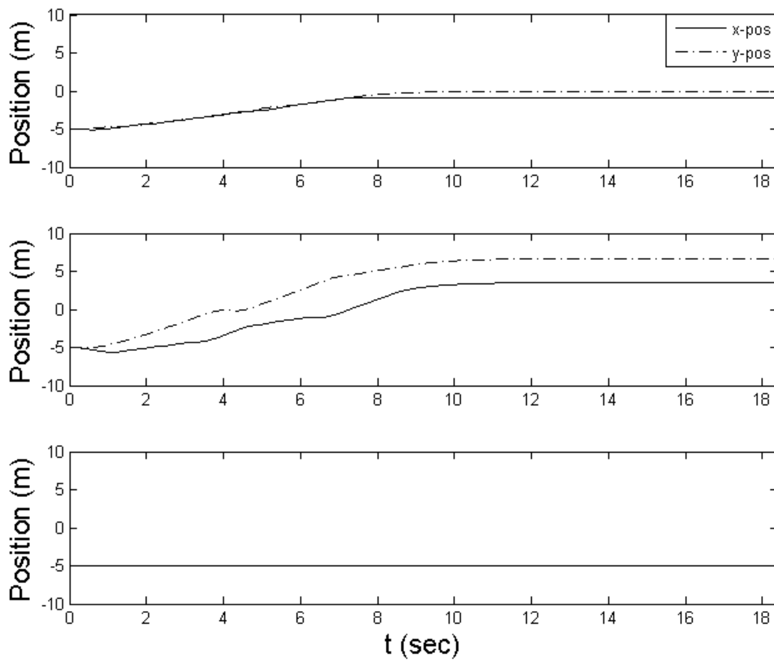


Figure 5.36: Heterogeneous Multiagent Robot States - Sim 4

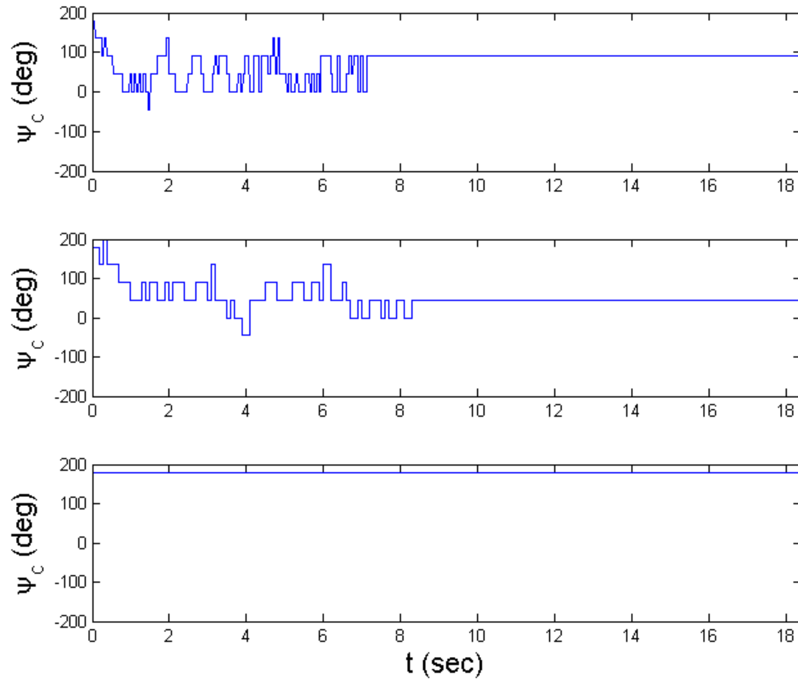


Figure 5.37: Heterogeneous Multiagent Robot Command - Sim 4

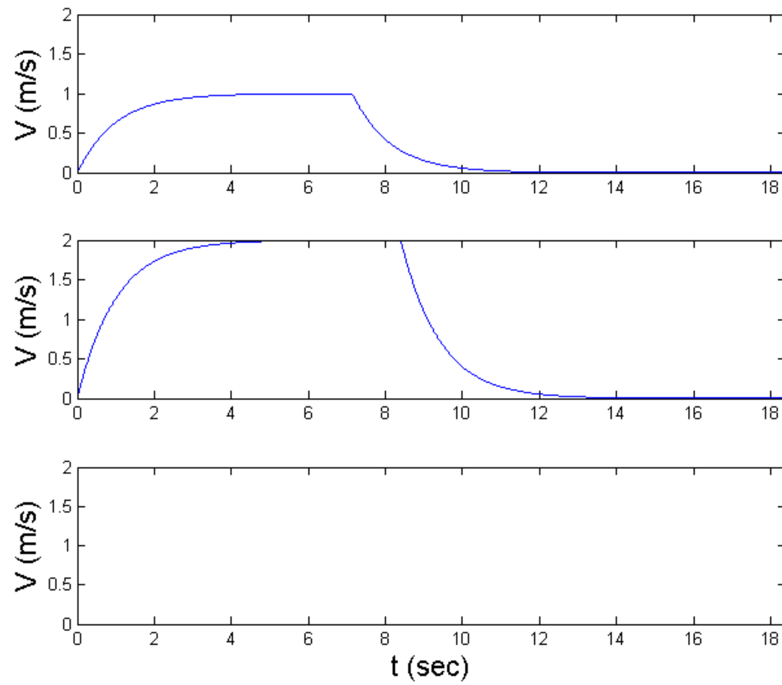


Figure 5.38: Heterogeneous Multiagent Robot Speed - Sim 4

The results presented in this chapter show that learning an approximation of the dynamics for an agent's states is useful for determining minimum time-to-goal commands in a hierarchical multiagent system. The learning of appropriate sample times and control policies by SDQL allows low-level agents to traverse the state-space and achieve goals assigned to them. The dynamics approximations learned from FODL and SODL can be used by the supervisory agent to run internal simulations and determine which agents should be assigned to each goal. This brings together all of the algorithms presented in this dissertation for use in a generalized simulation of dynamical multiagent command and control systems.

# 6. CONCLUSIONS AND RECOMMENDATIONS

This dissertation investigated several points involving the use of Reinforcement Learning in the control of dynamical systems. The following conclusions can be drawn from this dissertation.

## 6.1 Sampled-Data $Q$-learning

1. Sampled-Data $Q$-learning is capable of determining the sample time that maximizes the sample-time reward function. For a reward that is a function of the sample time itself, this gives a non-minimal sampling rate. If the reward is not a function of the sample time, Sampled-Data $Q$-learning chooses the fastest sampling rate allowed by the user.

2. Sampled-Data $Q$-learning is capable of learning the best sample time while simultaneously learning the control policy. This is accomplished with model-free learning. Separate control policies can be determined for different sample times by augmenting the state vector with the sample time itself. The next-state prediction probability is unaffected by this augmentation, so the Markov Property is preserved for systems that are already Markov.

3. A reward function quadratic in sample time is capable of determining a sample time that provides good success in control while being non-minimal. If the sample time reward function, $r(T)$, is linearly dependent on the sample time function, $f(T)$, then the sample time function can be factored out of the episodic sample time value function, $V_T(T)$. The result is the ability to use constant rewards for control within the episode while shaping the sample time value function external to the episodes. This is only the case if all rewards are

linearly dependent on the same sample time function.

## 6.2 Dynamics Learning

1. The First-Order Dynamics Learning algorithm is capable of accurately determining the time constant that models the first-order behavior of the state being investigated. It can also be concluded that the Second-Order Dynamics Learning algorithm is capable of accurately determining the natural frequency and damping ratio that models the second-order behavior of the state being investigated. Additionally, both the First- and Second-Order Dynamics Learning algorithms are capable of accurately learning dynamics parameters for multiple states within the same system.

2. The negative $L_2$-norm of the state prediction error provides a reward function capable of minimizing state prediction error, and therefore maximizing reward on the best approximation of dynamics.

3. The First- and Second-Order Dynamics Learning algorithms require simulation-specific ranges of sampling rates to accurately model the dynamics. For the robot example used in this work, the First-Order Dynamics Learning algorithm was accurate over the sample time range $0.006 < T < 1$. The Second-Order Dynamics Learning Algorithm was accurate over the sample time range $0.01 < T < 0.9$.

## 6.3 Multiagent Systems

1. The Sampled-Data $Q$-learning algorithm is capable of determining the best individual control policies and sample times for each low-level agent, which are not necessarily all the same. The control policy and associated sampling rate

learned for each agent can be used to control all agents simultaneously, given goal commands from the supervisory agent.

2. The First- and Second-Order Dynamics Learning algorithms are able to determine accurate dynamics approximations for all low-level agents involved in the simulations. These learned dynamics approximations are then used by the high-level supervisory agent to determine task allocation for each agent.

3. The high-level supervisor agent is capable of determining minimum total time to achieve all goals once the dynamics approximations, control policies, and sample times of all agents in the system have been learned.

## 6.4 Recommendations

The work presented in this dissertation involves using a Reinforcement Learning framework to determine algorithms capable of improving the performance of Reinforcement Learning control in systems with time dynamics. The algorithms investigated are Sampled-Data $Q$-learning, First-Order Dynamics Learning, and Second-Order Dynamics Learning. There are a number of extensions that merit investigation:

1. The scope of the systems investigated in this dissertation was limited to deterministic systems that can be modeled linearly. Further investigation of these algorithms to stochastic systems and nonlinear dynamics is warranted. Measurement noise, environmental disturbances, and nonlinear dynamics are a regular part of engineering systems that should be considered.

2. The systems investigated in this dissertation were numerical simulations of real-world systems. It is recommended that use of these algorithms on experimental hardware systems be investigated.

3. The First- and Second-Order Dynamics Learning algorithms were developed for learning dynamical models of states exhibiting first- and second-order dynamics, respectively. For systems of order greater than two, it would be of interest to investigate the use of Second-Order Dynamics Learning to approximate them. These systems can usually be approximated well by second-order systems because second-order systems exhibit the necessary traits (i.e. overshoot, oscillation, damping, etc.).

4. Extending these algorithms to the modeling of coupled systems would be a research area with great potential. The states modeled in this dissertation were exclusively uncoupled states with their own independent dynamics. Learning the models of the modes of systems with coupled states would be beneficial for systems with complex state dependence, such as in aircraft linear modeling. Learning time constants for first-order modes and natural frequency and damping ratios for second-order modes for aircraft linear models would be beneficial for autonomous aircraft system identification.

5. Investigate using the learned dynamics approximations to determine improved paths given what is already known in $Q$. Given that an action-value function has been determined for a particular query, it would be beneficial to determine how the learned dynamics approximations can be used to modify the policy for new queries that have not been learned. This can be explored in the context of new goals, new initial conditions, and changes to the environment (such as obstacles).

6. Investigating multiple layers of supervisors interacting with each other would be of interest to this research. In the multiagent systems described in this dissertation, there is one layer of low-level agents and a single supervisor for all of

them. Breaking regions into separate groups requires intercommunication between supervisors to achieve optimal path planning. The use of the algorithms presented in this dissertation could be of benefit to further research in these more complex multiagent environments.

# REFERENCES

[1] P. Abbeel, M. Quigley, and A.Y. Ng. Using inaccurate models in reinforcement learning. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, pages 1–8, New York, NY, 2006. ACM.

[2] J.A. Bagnell and J.G. Schneider. Autonomous helicopter control using reinforcement learning policy search methods. In *Proceedings of the 2001 IEEE International Conference on Robotics and Automation*, volume 2, pages 1615–1620, 2001.

[3] R. Bhattacharya and G.J. Balas. Anytime control algorithm: Model reduction approach. *Journal of Guidance, Control, and Dynamics*, 27(5):767–776, 2004.

[4] L. Busoniu, R. Babuska, and B. De Schutter. A comprehensive survey of multiagent reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 38:156–172, 2008.

[5] S. Chakravorty and J.L. Junkins. A methodology for intelligent path planning. In *Proceedings of the 2005 IEEE International Symposium on Intelligent Control*, pages 592–597, 2005.

[6] S. Chakravorty and S. Kumar. Generalized sampling-based motion planners. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 41(3):855–866, 2011.

[7] C. Claus and C. Boutilier. The dynamics of reinforcement learning in cooperative multiagent systems. In *Proceedings of National Conference on Artificial Intelligence*, AAAI-98, pages 746–752, 1998.

[8] J. Fisher and R. Bhattacharya. Linear quadratic regulation of systems with stochastic parameter uncertainties. *Automatica*, 45(12):2831–2841, 2009.

[9] G.F. Franklin, M.L. Workman, and D. Powell. *Digital Control of Dynamic Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 3rd edition, 1997.

[10] N. Friedman. Bayesian network classifiers. *Machine Learning*, 29(2-3):131–163, 1997.

[11] C. Gaskett, D. Wettergreen, and A. Zelinsky. Learning in continuous state and action spaces. In N. Foo, editor, *Advanced Topics in Artificial Intelligence*, volume 1747 of *Lecture Notes in Computer Science*, pages 417–428. Springer Berlin / Heidelberg, 1999.

[12] P. Gerard and O. Sigaud. Designing efficient exploration with macs: Modules and function approximation. In *Proceedings of the Fifth Genetic and Evolutionary Computation Conference*, volume 2724, pages 1882–1893, 2003.

[13] M. Ghavamzadeh and S. Mahadevan. Learning to communicate and act using hierarchical reinforcement learning. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 3*, AAMAS '04, pages 1114–1121, Washington, DC, 2004. IEEE Computer Society.

[14] M. Ghavamzadeh, S. Mahadevan, and R. Makar. Hierarchical multi-agent reinforcement learning. *Autonomous Agents and Multi-Agent Systems*, 13:197–229, 2006.

[15] E. R. Gomes and R. Kowalczyk. Dynamic analysis of multiagent q-learning with

$\varepsilon$-greedy exploration. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 369–376, Montreal, Quebec, Canada, 2009.

[16] K. Gopal, T. Romo, J.C. Sacchettini, and T.R. Ioerger. Efficient retrieval of electron density patterns for modeling proteins by x-ray crystallography. In *International Conference on Machine Learning and Applications (ICMLA'04)*, pages 380–387, Louisville, KY, 2004.

[17] J. Hu and M.P. Wellman. Multiagent reinforcement learning: Theoretical framework and an algorithm. In *Proceedings of the 15th International Conference on Machine Learning*, pages 242–250, Madison, WI, 1998.

[18] L. C. Baird III. *Reinforcement Learning Through Gradient Descent*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1999.

[19] S. Kapetanakis and D. Kudenko. Reinforcement learning of coordination in cooperative multi-agent systems. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence*, AAAI-02, pages 326–331, 2002.

[20] K. Kirkpatrick. *Reinforcement Learning for Active Length Control and Hysteresis Characterization of Shape Memory Alloys*. Master's thesis, Texas A&M University, College Station, TX, 2009.

[21] K. Kirkpatrick, J. May Jr., and J. Valasek. Aircraft system identification using artificial neural networks. In *AIAA Aerospace Sciences Meeting*, pages 157–163, Grapevine, TX, 2013.

[22] K. Kirkpatrick and J. Valasek. Reinforcement learning for characterizing hysteresis behavior of shape memory alloys. *Journal of Aerospace Computing, Information, and Communication*, 6(3):227–238, 2009.

[23] K. Kirkpatrick and J. Valasek. Active length control of shape memory alloy wires using reinforcement learning. *Journal of Intelligent Material Systems and Structures*, 22(14):1595–1604, 2011.

[24] K. Kirkpatrick and J. Valasek. Morphing smart material actuator control using reinforcement learning. In J. Valasek, editor, *Morphing Aerospace Vehicles and Structures*. John Wiley & Sons, Chichester, UK, 2012.

[25] K. Kirkpatrick and J. Valasek. Reinforcement learning control with time dependent agent dynamics. In F. L. Lewis and D. Liu, editors, *Reinforcement Learning and Approximate Dynamic Programming for Feedback Control*. John Wiley & Sons, Hoboken, NJ, 2012.

[26] M. G. Lagoudakis and R. Parr. Model-free least squares policy iteration. Technical Report CS-2001-05, Duke University, Durham NC, 2001.

[27] A. Lampton. *Discretization and Approximation Methods for Reinforcement Learning of Highly Reconfigurable Systems*. PhD thesis, Texas A&M University, 2009.

[28] P. L. Lanzi. Learning classifier systems from a reinforcement learning perspective. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 6:162–170, 2002.

[29] M. Lauer and M. Riedmiller. An algorithm for distributed reinforcement learning in cooperative multi-agent systems. In *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, pages 535–542, San Francisco, CA, 2000. Morgan Kaufmann Publishers Inc.

[30] M. L. Littman. Markov games as a framework for multi-agent reinforcement

learning. In *The Eleventh International Conference on Machine Learning*, pages 157–163, San Francisco, CA, 1994.

[31] R. A. McCallum. Instance-based state identification for reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 7, Cambridge, MA, 1995. MIT Press.

[32] T. M. Mitchell. *Machine Learning.* The McGraw-Hill Companies, Inc., Singapore, 1997.

[33] R. Munos. A study of reinforcement learning in the continuous case by the means of viscosity solutions. *Machine Learning*, 40:265–299, 2000.

[34] A. Y. Ng, D. Harada, and S. Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Sixteenth International Conference on Machine Learning*, pages 278–287, 1999.

[35] L. Panait and S. Luke. Cooperative multi-agent learning: The state of the art. *Autonomous Agents And Multi-Agent Systems*, 11(3):387–434, 2005.

[36] T. Poggio and F. Girosi. Networks for approximation and learning. In *Proceedings of the IEEE*, Cambridge, MA, 1990.

[37] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[38] M.A. Rahman and M.A. Hoque. On-line adaptive artificial neural network based vector control of permanent magnet synchronous motors. *IEEE Transactions on Energy Conservation*, 13(4):311–318, 1998.

[39] M. Riedmiller. Neural fitted q iteration - first experiences with a data efficient neural reinforcement learning method. In *Machine Learning: ECML 2005*, vol-

ume 3720 of *Lecture Notes in Computer Science*, pages 317–328, Porto, Portugal, 2005. Springer.

[40] P. Sabes. Approximating q-values with basis function representations. In *The Fourth Connectionist Models Summer School*, pages 264–271, Hillsdale, NJ, 1993.

[41] W.D. Smart and L.P. Kaelbling. Practical reinforcement learning in continuous spaces. In *Proceedings of the Sixteenth International Conference on Machine Learning*, pages 903–910, 2000.

[42] R. Sutton and A. Barto. *Reinforcement Learning: An Introductions*. The MIT Press, Cambridge, MA, 1998.

[43] M. Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *The Tenth International Conference on Machine Learning*, pages 330–337, Amherst, MA, 1993.

[44] G. Taylor and R. Parr. Kernelized value function approximation for reinforcement learning. In *Proceedings of the 26th International Conference on Machine Learning*, Montreal, Canada, 2009.

[45] S. H. G. ten Hagen. *Continuous State Space Q-Learning for Control of Nonlinear Systems*. PhD thesis, University of Amsterdam, 2001.

[46] G. Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2):215–219, 1994.

[47] G. Tesauro. Extending q-learning to general adaptive multiagent systems. In *Advances in Neural Information Processing Systems*, volume 16, Vancouver and Whistler, Canada, 2003.

[48] J. Valasek and W. Chen. Observer/kalman filter identification for online system identification of aircraft. *Journal of Guidance, Control, and Dynamics*, 26(2):347–353, 2003.

[49] H. Vollbrecht. *Hierarchical Reinforcement Learning in Continuous State Spaces*. PhD thesis, Abteilung Neuroinformatik, Universität Ulm, Ulm, Germany, 2003.

[50] C. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992.

[51] S. Whiteson, M. E. Taylor, and P. Stone. Empirical studies in action selection with reinforcement learning. *Adaptive Behavior*, 15:33–50, 2007.

[52] S. Wollkind, J. Valasek, and T.R. Ioerger. Automated conflict resolution for air traffic management using cooperative multiagent negotiation. In *AIAA Guidance, Navigation, and Control Conference*, Providence, RI, 2004.

[53] C. Zhang, S. Abdallah, and V. Lesser. Efficient multi-agent reinforcement learning through automated supervision. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 3*, AAMAS '08, pages 1365–1370, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[54] C. Zhang, S. Abdallah, and V. Lesser. Integrating organizational control into multi-agent learning. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '09, pages 757–764, Richland, SC, 2009. International Foundation for Autonomous Agents and Multiagent Systems.

# APPENDIX A

## DERIVATION OF INVERTED PENDULUM EQUATIONS OF MOTION

In this appendix, the equations of motion for an inverted pendulum on a cart are derived. The system consists of a cart of mass $M$ with a massless pendulum arm of length $L$ oriented vertically upwards and a mass $m$ at the tip. The pendulum arm is perturbed from vertical by the angle $\theta$, and is controlled by a force $F$ that acts horizontally on the cart. This system is shown in Figure A.1.

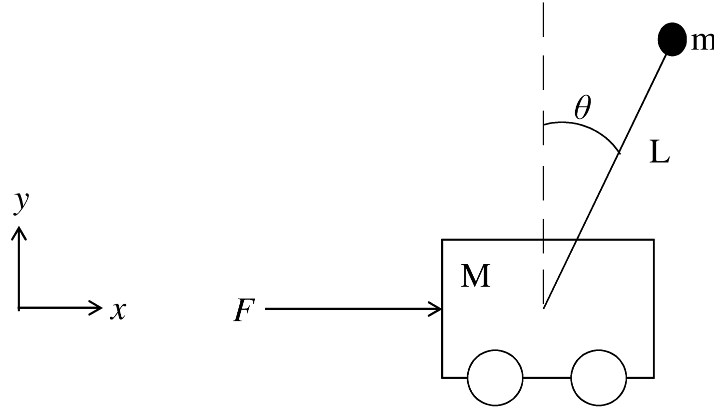


Figure A.1: Inverted Pendulum on a Cart

To determine the equations of motion for the system described by Figure A.1, Lagrangian mechanics can be used. The Lagrangian function is described by Equation A.1, where $L_f$ is the Lagrangian function, $K$ is the kinetic energy, and $U$ is the potential energy.

$$L_f = K - U \tag{A.1}$$

Using the Lagrangian function, the equations of motion for a system can be determined by the Euler-Lagrange equations, where $q_i$ are the generalized coordinates. The derivatives of $q_i$ are $\dot{q}_i$, and are independent of $q$'s. The generalized forces are represented by the symbol $Q_i$. These equations are as shown in Equation A.2.

$$\frac{d}{dt}\left(\frac{\partial L_f}{\partial \dot{q}_i}\right) - \frac{\partial L_f}{\partial q_i} = Q_i \tag{A.2}$$

Determining the generalized forces can be done using a calculation of the work rate. The work rate can be described by the inner product between the forces of the system and the velocity. This is shown in Equation A.3, where $\dot{W}$ is the work rate, $\underline{F}$ is the force vector, and $\underline{v}$ is the velocity vector.

$$\dot{W} = \underline{F} \cdot \underline{v} \tag{A.3}$$

If the velocity is written in terms of generalized coordinates, $\underline{v} = \underline{v}(q)$, then the velocity can be written according to Equation A.4, where there are $n$ degrees of freedom.

$$\underline{v} = \sum_{i=1}^{n} \frac{\partial \underline{r}}{\partial q_i} \dot{q}_i \tag{A.4}$$

This yields the form of the work rate shown in Equation A.5.

$$\dot{W} = \underline{F} \cdot \sum_{i=1}^{n} \frac{\partial \underline{r}}{\partial q_i} \dot{q}_i = Q_i \dot{q}_i \tag{A.5}$$

By examining Equation A.5, it can be seen that the generalized forces are as described by Equation A.6.

$$Q_i = \underline{F} \cdot \sum_{i=1}^{n} \frac{\partial \underline{r}}{\partial q_i} \tag{A.6}$$

To make the determination of generalized forces simpler, the relationship described by Equation A.7 can be used.

$$\frac{\partial \underline{r}}{\partial q_i} = \frac{\partial \underline{v}}{\partial \dot{q}_i} = \frac{\partial \underline{a}}{\partial \ddot{q}_i} \tag{A.7}$$

It is often simpler to determine the velocity partial than the displacement partial, so substituting this equality into Equation A.6 yields the following form for determining generalized forces:

$$Q_i = \underline{F} \cdot \sum_{i=1}^{n} \frac{\partial \underline{v}}{\partial \dot{q}_i} \tag{A.8}$$

So, the equations of motion can be determined according to Equation A.9.

$$\frac{d}{dt}\left(\frac{\partial L_f}{\partial \dot{q}_i}\right) - \frac{\partial L_f}{\partial q_i} = \underline{F} \cdot \sum_{i=1}^{n} \frac{\partial \underline{v}}{\partial \dot{q}_i} \tag{A.9}$$

In the system described by Figure A.1, there are 2 independent degrees of freedom and therefore 2 generalized coordinates. The motion can be broken up into two different systems, with the cart as one system and the mass as the other. The velocities of each of these two systems can be described according to Equations A.10 and A.11.

$$v_c^2 = \dot{x}^2 \tag{A.10}$$

$$v_m^2 = \left(\frac{d}{dt}(x + L\sin\theta)\right)^2 + \left(\frac{d}{dt}(L\cos\theta)\right)^2 \tag{A.11}$$

The mass velocity, $v_m$, can be expanded to the form shown in Equations A.12-A.14.

$$v_m^2 = \left(\dot{x} + L\dot{\theta}\cos\theta\right)^2 + \left(-L\dot{\theta}\sin\theta\right)^2 \tag{A.12}$$

$$v_m^2 = \dot{x}^2 + 2\dot{x}\dot{\theta}L\cos\theta + L^2\dot{\theta}^2\cos^2\theta + L^2\dot{\theta}^2\sin^2\theta \tag{A.13}$$

$$v_m^2 = \dot{x}^2 + 2\dot{x}\dot{\theta}L\cos\theta + L^2\dot{\theta}^2 \tag{A.14}$$

Using Equations A.10 and A.14, the kinetic energy can be determined.

$$K = \frac{1}{2}(Mv_c^2 + mv_m^2) = \frac{1}{2}M\dot{x}^2 + \frac{1}{2}m\left(\dot{x}^2 + 2\dot{x}\dot{\theta}L\cos\theta + L^2\dot{\theta}^2\right) \tag{A.15}$$

$$U = mgL\cos\theta \tag{A.16}$$

This results in the following Lagrange function:

$$\begin{aligned} L_f &= \frac{1}{2}M\dot{x}^2 + \frac{1}{2}m\left(\dot{x}^2 + 2\dot{x}\dot{\theta}L\cos\theta + L^2\dot{\theta}^2\right) - mgL\cos\theta & \text{(A.17)} \\ &= \frac{1}{2}(M+m)\dot{x}^2 + \frac{1}{2}mL^2\dot{\theta}^2 + mL\dot{x}\dot{\theta}\cos\theta - mgL\cos\theta & \text{(A.18)} \end{aligned}$$

The generalized forces for this system are as shown in Equations A.19 and A.20.

$$Q_x = F \tag{A.19}$$

$$Q_\theta = 0 \tag{A.20}$$

The derivatives of Equation A.18 can be evaluated, and are shown in Equations A.21-A.26.

$$\frac{\partial L_f}{\partial \dot{x}} = (M+m)\dot{x} + mL\dot{\theta}\cos\theta \tag{A.21}$$

$$\frac{\partial L_f}{\partial x} = 0 \tag{A.22}$$

$$\frac{\partial L_f}{\partial \dot{\theta}} = mL^2\dot{\theta} + mL\dot{x}\cos\theta \tag{A.23}$$

$$\frac{\partial L_f}{\partial \theta} = -mL\dot{x}\dot{\theta}\sin\theta + mgL\sin\theta \tag{A.24}$$

$$\frac{d}{dt}\left(\frac{\partial L_f}{\partial \dot{x}}\right) = (M+m)\ddot{x} + mL\ddot{\theta}\cos\theta - mL\dot{\theta}^2\sin\theta \tag{A.25}$$

$$\frac{d}{dt}\left(\frac{\partial L_f}{\partial \dot{\theta}}\right) = mL^2\ddot{\theta} + mL\ddot{x}\cos\theta - mL\dot{x}\dot{\theta}\sin\theta \tag{A.26}$$

By substituting these relations into Equation A.2, the equations of motion can be determined by the following:

$$F = (M+m)\ddot{x} + mL\ddot{\theta}\cos\theta - mL\dot{\theta}^2\sin\theta \tag{A.27}$$

$$0 = mL^2\ddot{\theta} + mL\ddot{x}\cos\theta - mL\dot{x}\dot{\theta}\sin\theta + mL\dot{x}\dot{\theta}\sin\theta - mgL\sin\theta \tag{A.28}$$

$$= L\ddot{\theta} + \ddot{x}\cos\theta - g\sin\theta \tag{A.29}$$

By rearranging Equation A.29, the equation for the second time derivative of $\theta$ can be determined, as shown in Equation A.30

$$\boxed{\ddot{\theta} = \frac{g\sin\theta - \ddot{x}\cos\theta}{L}} \tag{A.30}$$

Equation A.30 can then be substituted into Equation A.27 to determine a solution for $\ddot{x}$. This is shown in Equations A.31-A.33, with the solution for $\ddot{x}$ shown in Equation A.34.

$$F = (M+m)\ddot{x} + mL\left(\frac{g\sin\theta - \ddot{x}\cos\theta}{L}\right)\cos\theta - mL\dot{\theta}^2\sin\theta \tag{A.31}$$

$$F = (M+m)\ddot{x} - m\ddot{x}\cos^2\theta + mg\sin\theta\cos\theta - mL\dot{\theta}^2\sin\theta \tag{A.32}$$

$$\left(M + m(1-\cos^2\theta)\right)\ddot{x} = F + mL\dot{\theta}^2\sin\theta - mg\sin\theta\cos\theta \tag{A.33}$$

$$\boxed{\ddot{x} = \frac{F + mL\dot{\theta}^2\sin\theta - mg\sin\theta\cos\theta}{M + m\sin^2\theta}} \tag{A.34}$$

Now if the variables are redefined in state-space form according to Equations A.35-A.38, the equations of motion can be written in those terms. The full dynamics describing the system of Figure A.1 are shown in state-space notation in Equation A.39.

$$x_1 = x \tag{A.35}$$

$$x_2 = \dot{x} = \dot{x_1} \tag{A.36}$$

$$x_3 = \theta \tag{A.37}$$

$$x_4 = \dot{\theta} = \dot{x_3} \tag{A.38}$$

$$\boxed{\begin{aligned} &\dot{x_1} = x_2 \\ &\dot{x_2} = \frac{F + mLx_4^2 \sin(x_3) - mg\sin(x_3)\cos(x_3)}{M + m\sin^2(x_3)} \\ &\dot{x_3} = x_4 \\ &\dot{x_4} = \frac{g\sin(x_3) - \dot{x_2}\cos(x_3)}{L} \end{aligned}} \tag{A.39}$$

# APPENDIX B

## PARTIAL FRACTION EXPANSION SOLUTIONS

In this appendix, partial fraction expansion is performed on a second-order system in response to a step input of commanded next state. The second-order differential equation being solved is as shown in Equation B.1, where the state is $s$ and the system is responding to a nonzero setpoint state command of $s_c$.

$$\ddot{s} + 2\zeta\omega\dot{s} + \omega^2 s = \omega^2 s_c \tag{B.1}$$

The frequency domain solution for this equation is as shown in Equation B.2.

$$S(\lambda) = \frac{\dot{s}(0) + (\lambda + 2\zeta\omega)s(0)}{\lambda^2 + 2\zeta\omega\lambda + \omega^2} + \frac{\omega^2 s_c}{\lambda(\lambda^2 + 2\zeta\omega\lambda + \omega^2)} \tag{B.2}$$

To determine the time domain solution to Equation B.1, the inverse Laplace transform needs to be determined for Equation B.2. Determining the appropriate solution requires partial fraction expansion, and to simplify matters separate solutions will be found for the 4 uniques cases of damping ratio.

### B.1 No Damping

In this first scenario, the case of no damping is considered so $\zeta = 0$. This results in the following simplification of Equation B.2.

$$S(\lambda) = \frac{\dot{s}(0) + \lambda s(0)}{\lambda^2 + \omega^2} + \frac{\omega^2 s_c}{\lambda(\lambda^2 + \omega^2)} \tag{B.3}$$

The second term of Equation B.3 can be expanded as shown in Equations B.4-B.9.

$$\frac{\omega^2 s_c}{\lambda(\lambda^2+\omega^2)} = \frac{C_1}{\lambda} + \frac{C_2\lambda + C_3}{\lambda^2+\omega^2} \tag{B.4}$$

$$\omega^2 s_c = C_1(\lambda^2+\omega^2) + C_2\lambda^2 + C_3\lambda \tag{B.5}$$

$$\omega^2 s_c = (C_1+C_2)\lambda^2 + C_3\lambda + C_1\omega^2 \tag{B.6}$$

$$\Rightarrow \begin{cases} C_1 + C_2 &= 0 \\ C_3 &= 0 \\ \omega^2 C_1 &= \omega^2 s_c \end{cases} \tag{B.7}$$

$$\Rightarrow \begin{cases} C_1 &= s_c \\ C_2 &= -s_c \\ C_3 &= 0 \end{cases} \tag{B.8}$$

$$\frac{\omega^2 s_c}{\lambda(\lambda^2+\omega^2)} = \frac{s_c}{\lambda} - \frac{s_c\lambda}{\lambda^2+\omega^2} \tag{B.9}$$

So the expanded form of the frequency domain solution can be written as shown in Equation B.10.

$$S(\lambda) = \frac{\dot{s}(0) + \lambda s(0)}{\lambda^2+\omega^2} + \frac{s_c}{\lambda} - \frac{s_c\lambda}{\lambda^2+\omega^2} \tag{B.10}$$

This can be further expanded to the form shown in Equation B.11.

$$S(\lambda) = \frac{s_c}{\lambda} + \frac{\dot{s}(0)}{\omega}\left(\frac{\omega}{\lambda^2+\omega^2}\right) + (s(0) - s_c)\frac{\lambda}{\lambda^2+\omega^2} \tag{B.11}$$

The time domain solution can be found by using the inverse Laplace transforms

in Equations B.12-B.14.

$$\mathcal{L}^{-1}\left\{\frac{1}{\lambda}\right\} = 1 \tag{B.12}$$

$$\mathcal{L}^{-1}\left\{\frac{\omega}{\lambda^2+\omega^2}\right\} = \sin(\omega t) \tag{B.13}$$

$$\mathcal{L}^{-1}\left\{\frac{\lambda}{\lambda^2+\omega^2}\right\} = \cos(\omega t) \tag{B.14}$$

With these inverse Laplace transforms, the time domain solution for a the second-order system described in Equation B.1 can be found when $\zeta = 0$. This solution is shown in Equation B.15.

$$\boxed{s(t) = s_c + \frac{\dot{s}(0)}{\omega}\sin(\omega t) + (s(0) - s_c)\cos(\omega t)} \tag{B.15}$$

## B.2 Underdamped

The next case will consider underdamping, or a range of $0 < \zeta < 1$. In this scenario, the full form of Equation B.2 must be expanded. The first term of Equation B.2 can be expanded according to Equations B.16-B.18.

$$\frac{\dot{s}(0) + (\lambda + 2\zeta\omega)s(0)}{\lambda^2 + 2\zeta\omega\lambda + \omega^2} = \frac{\lambda s(0) + \dot{s}(0) + 2\zeta\omega s(0)}{(\lambda + \zeta\omega)^2 + \omega^2(1 - \zeta^2)} \tag{B.16}$$

$$= s(0)\left(\frac{\lambda + \frac{\dot{s}(0)}{s(0)} + 2\zeta\omega}{(\lambda + \zeta\omega)^2 + \omega^2(1 - \zeta^2)}\right) \tag{B.17}$$

$$\begin{aligned} &= s(0)\frac{\zeta\omega}{(\lambda + \zeta\omega)^2 + \omega^2(1 - \zeta^2)} \\ &\quad + s(0)\frac{\lambda + \zeta\omega}{(\lambda + \zeta\omega)^2 + \omega^2(1 - \zeta^2)} \\ &\quad + \frac{\dot{s}(0)}{\zeta\omega}\frac{\zeta\omega}{(\lambda + \zeta\omega)^2 + \omega^2(1 - \zeta^2)} \end{aligned} \tag{B.18}$$

By using the inverse Laplace transforms in Equations B.13 and B.14, Equation B.18 can be solved for the time domain solution of the first term in Equation B.2. This is shown in Equations B.19-B.20.

$$\begin{aligned} \mathcal{L}^{-1}\left\{\frac{\dot{s}(0) + (\lambda + 2\zeta\omega)s(0)}{\lambda^2 + 2\zeta\omega\lambda + \omega^2}\right\} &= s(0)e^{-\zeta\omega t}\frac{\zeta}{\sqrt{1 - \zeta^2}}\sin(\omega t\sqrt{1 - \zeta^2}) \\ &\quad + s(0)e^{-\zeta\omega t}\cos(\omega t\sqrt{1 - \zeta^2}) \\ &\quad + \frac{\dot{s}(0)}{\zeta\omega}e^{-\zeta\omega t}\frac{\zeta}{\sqrt{1 - \zeta^2}}\sin(\omega t\sqrt{1 - \zeta^2}) \end{aligned} \tag{B.19}$$

$$\begin{aligned} &= \frac{e^{-\zeta\omega t}\sin(\omega t\sqrt{1 - \zeta^2})}{\omega\sqrt{1 - \zeta^2}}(\zeta\omega s(0) + \dot{s}(0)) \\ &\quad + s(0)e^{-\zeta\omega t}\cos(\omega t\sqrt{1 - \zeta^2}) \end{aligned} \tag{B.20}$$

Now to determine a time domain solution to the second term in Equation B.2 requires partial fraction expansion. The expansion is demonstrated in Equations B.21-B.27.

$$\frac{\omega^2 s_c}{\lambda(\lambda^2 + 2\zeta\omega\lambda + \omega^2)} = \frac{C_1}{\lambda} + \frac{C_2\lambda + C_3}{\lambda^2 + 2\zeta\omega\lambda + \omega^2} \tag{B.21}$$

$$\omega^2 s_c = C_1(\lambda^2 + 2\zeta\omega\lambda + \omega^2) + (C_2\lambda + C_3)\lambda \tag{B.22}$$

$$\omega^2 s_c = \lambda^2(C_1 + C_2) + \lambda(2\zeta\omega C_1 + C_3) + C_1\omega^2 \tag{B.23}$$

$$\Rightarrow \begin{cases} C_1 + C_2 &= 0 \\ 2\zeta\omega C_1 + C_3 &= 0 \\ \omega^2 C_1 &= \omega^2 s_c \end{cases} \tag{B.24}$$

$$\Rightarrow \begin{cases} C_1 &= s_c \\ C_2 &= -s_c \\ C_3 &= -2\zeta\omega s_c \end{cases} \tag{B.25}$$

$$\frac{\omega^2 s_c}{\lambda(\lambda^2 + 2\zeta\omega\lambda + \omega^2)} = \frac{s_c}{\lambda} - \frac{s_c\lambda + 2\zeta\omega s_c}{\lambda^2 + 2\zeta\omega\lambda + \omega^2} \tag{B.26}$$

$$\frac{\omega^2 s_c}{\lambda(\lambda^2 + 2\zeta\omega\lambda + \omega^2)} = \frac{s_c}{\lambda} - \frac{s_c\zeta\omega}{(\lambda + \zeta\omega)^2 + \omega^2(1-\zeta^2)} - \frac{s_c(\lambda + \zeta\omega)}{(\lambda + \zeta\omega)^2 + \omega^2(1-\zeta^2)} \tag{B.27}$$

The inverse Laplace transforms in Equations B.12-B.14 can then be used to transform Equation B.27 into the time domain. This produces the time domain solution to the second term of Equation B.2, shown in Equation B.28.

$$\mathcal{L}^{-1}\left\{\frac{\omega^2 s_c}{\lambda(\lambda^2 + 2\zeta\omega\lambda + \omega^2)}\right\} = s_c - s_c\frac{\zeta}{\sqrt{1-\zeta^2}}e^{-\zeta\omega t}\sin(\omega t\sqrt{1-\zeta^2}) - s_c e^{-\zeta\omega t}\cos(\omega t\sqrt{1-\zeta^2}) \tag{B.28}$$

By combining Equations B.20 and B.28, we arrive at the full time domain solution to Equation B.1 for the case of $0 < \zeta < 1$ shown in Equation B.29.

$$\boxed{\begin{aligned} s(t) = s_c + \frac{\zeta}{\sqrt{1-\zeta^2}} e^{-\zeta\omega t}\left(s(0) - s_c + \frac{\dot{s}(0)}{\zeta\omega}\right)\sin(\omega t\sqrt{1-\zeta^2}) \\ + e^{-\zeta\omega t}(s(0) - s_c)\cos(\omega t\sqrt{1-\zeta^2}) \end{aligned}} \tag{B.29}$$

## B.3 Critically Damped

The critically damped case is considered here, so the damping ratio is designated as $\zeta = 1$. This yields the following reduced form of Equation B.2.

$$S(\lambda) = \frac{\dot{s}(0) + (\lambda + 2\omega)s(0)}{\lambda^2 + 2\omega\lambda + \omega^2} + \frac{\omega^2 s_c}{\lambda(\lambda^2 + 2\omega\lambda + \omega^2)} \tag{B.30}$$

$$S(\lambda) = \frac{\dot{s}(0) + (\lambda + 2\omega)s(0)}{(\lambda + \omega)^2} + \frac{\omega^2 s_c}{\lambda(\lambda + \omega)^2} \tag{B.31}$$

The first term of Equation B.31 can be expanded as shown below.

$$\frac{\dot{s}(0) + (\lambda + 2\omega)s(0)}{(\lambda + \omega)^2} = \frac{C_1}{\lambda + \omega} + \frac{C_2}{(\lambda + \omega)^2} \tag{B.32}$$

$$\dot{s}(0) + (\lambda + 2\omega)s(0) = C_1(\lambda + \omega) + C_2 \tag{B.33}$$

$$\Rightarrow \begin{cases} s(0)\lambda = C_1 \\ \dot{s}(0) + 2\omega s(0) = \omega C_1 + C_2 \end{cases} \tag{B.34}$$

$$\Rightarrow \begin{cases} C_1 &= s(0) \\ C_2 &= \dot{s}(0) + \omega s(0) \end{cases} \tag{B.35}$$

This results in the following expansion for the first term.

$$\frac{\dot{s}(0) + (\lambda + 2\omega)s(0)}{(\lambda + \omega)^2} = \frac{s(0)}{\lambda + \omega} + \frac{\dot{s}(0) + \omega s(0)}{(\lambda + \omega)^2} \tag{B.36}$$

The second term of Equation B.31 can be expanded as shown below.

$$\frac{\omega^2 s_c}{\lambda(\lambda + \omega)^2} = \frac{C_1}{\lambda} + \frac{C_2}{\lambda + \omega} + \frac{C_3}{(\lambda + \omega)^2} \tag{B.37}$$

$$\omega^2 s_c = C_1(\lambda + \omega)^2 + C_2\lambda(\lambda + \omega) + C_3\lambda \tag{B.38}$$

$$\omega^2 s_c = \lambda^2(C_1 + C_2) + \lambda(2C_1\omega + C_2\omega + C_3) + C_1\omega^2 \tag{B.39}$$

$$\Rightarrow \begin{cases} C_1 + C_2 &= 0 \\ 2C_1\omega + C_2\omega + C_3 &= 0 \\ C_1\omega^2 &= s_c\omega^2 \end{cases} \tag{B.40}$$

$$\Rightarrow \begin{cases} C_1 &= s_c \\ C_2 &= -s_c \\ C_3 &= -s_c\omega \end{cases} \tag{B.41}$$

This results in the following expanded form of the second term.

$$\frac{\omega^2 s_c}{\lambda(\lambda + \omega)^2} = \frac{s_c}{\lambda} - \frac{s_c}{\lambda + \omega} - \frac{\omega s_c}{(\lambda + \omega)^2} \tag{B.42}$$

The fully expanded form of Equation B.31 is shown in Equations B.43 and B.44.

$$S(\lambda) = \frac{s(0)}{\lambda + \omega} + \frac{\dot{s}(0) + \omega s(0)}{(\lambda + \omega)^2} + \frac{s_c}{\lambda} - \frac{s_c}{\lambda + \omega} - \frac{\omega s_c}{(\lambda + \omega)^2} \tag{B.43}$$

$$S(\lambda) = \frac{s_c}{\lambda} + \frac{s(0) - s_c}{\lambda + \omega} + \frac{\dot{s}(0) + \omega s(0) - \omega s_c}{(\lambda + \omega)^2} \tag{B.44}$$

The inverse Laplace transform can be used to obtain the time-domain solution to Equation B.44. Equations B.45 and B.46 are more general Laplace transforms that can be used in this case.

$$\mathcal{L}^{-1}\left\{\frac{1}{\lambda - a}\right\} = e^{at} \tag{B.45}$$

$$\mathcal{L}^{-1}\left\{\frac{n!}{(\lambda - a)^{n+1}}\right\} = t^n e^{at} \tag{B.46}$$

Using these inverse transforms, the time domain solution for the state can be determined. The solution for a critically damped second-order system is shown in Equation B.47.

$$\boxed{s(t) = s_c + (s(0) - s_c)e^{-\omega t} + (\dot{s}(0) + \omega s(0) - \omega s_c)te^{-\omega t}} \tag{B.47}$$

### B.4 Overdamped

Here the case of overdamping is discussed, so in this scenario $\zeta > 1$. For this case the full form of Equation B.2 must be expanded, but it will be done using different substitutions than the underdamped case in Section B.2. First we will define two new constants to make the derivation easier.

$$z_1 = \omega(\zeta - \sqrt{\zeta^2 - 1}) \tag{B.48}$$

$$z_2 = \omega(\zeta + \sqrt{\zeta^2 - 1}) \tag{B.49}$$

With these constants, Equation B.2 can be rearranged to the form shown in Equation B.50.

$$S(\lambda) = \frac{\dot{s}(0) + (\lambda + 2\zeta\omega)s(0)}{(\lambda + z_1)(\lambda + z_2)} + \frac{\omega^2 s_c}{\lambda(\lambda + z_1)(\lambda + z_2)} \tag{B.50}$$

The first term of Equation B.50 will be expanded using partial fractions as shown in Equations B.51-B.56.

$$\frac{\dot{s}(0) + (\lambda + 2\zeta\omega)s(0)}{(\lambda + z_1)(\lambda + z_2)} = \frac{C_1}{\lambda + z_1} + \frac{C_2}{\lambda + z_2} \tag{B.51}$$

$$\frac{\dot{s}(0) + (\lambda + 2\zeta\omega)s(0)}{(\lambda + z_1)(\lambda + z_2)} = C_1(\lambda + z_2) + C_2(\lambda + z_1) \tag{B.52}$$

$$\frac{\dot{s}(0) + (\lambda + 2\zeta\omega)s(0)}{(\lambda + z_1)(\lambda + z_2)} = \lambda(C_1 + C_2) + C_1 z_2 + C_2 z_1 \tag{B.53}$$

$$\Rightarrow \begin{cases} C_1 + C_2 &= s(0) \\ C_1 z_2 + C_2 z_1 &= \dot{s}(0) + 2\zeta\omega s(0) \end{cases} \tag{B.54}$$

$$\Rightarrow \begin{cases} C_1 &= \frac{\dot{s}(0) + (2\zeta\omega - z_1)s(0)}{z_2 - z_1} \\ C_2 &= -\frac{\dot{s}(0) + (2\zeta\omega + z_2)s(0)}{z_2 - z_1} \end{cases} \tag{B.55}$$

$$\frac{\dot{s}(0) + (\lambda + 2\zeta\omega)s(0)}{(\lambda + z_1)(\lambda + z_2)} = \left(\frac{\dot{s}(0) + (2\zeta\omega - z_1)s(0)}{z_2 - z_1}\right)\frac{1}{\lambda + z_1} - \left(\frac{\dot{s}(0) + (2\zeta\omega + z_2)s(0)}{z_2 - z_1}\right)\frac{1}{\lambda + z_2} \tag{B.56}$$

The frequency domain solution for the first term of Equation B.50 can then be transformed into the time domain using Equation B.45. This time domain solution to the first term is shown in Equation B.57.

$$\mathcal{L}^{-1}\left\{\frac{\dot{s}(0) + (\lambda + 2\zeta\omega)s(0)}{(\lambda + z_1)(\lambda + z_2)}\right\} = \left(\frac{\dot{s}(0) + (2\zeta\omega - z_1)s(0)}{z_2 - z_1}\right)e^{-z_1 t} \tag{B.57}$$

$$- \left(\frac{\dot{s}(0) + (2\zeta\omega + z_2)s(0)}{z_2 - z_1}\right)e^{-z_2 t} \tag{B.58}$$

Now the second term of Equation B.50 can be expanded in a similar way. This expansion is shown in Equations B.59-B.64.

$$\frac{\omega^2 s_c}{\lambda(\lambda + z_1)(\lambda + z_2)} = \frac{C_1}{\lambda} + \frac{C_2}{\lambda + z_1} + \frac{C_3}{\lambda + z_2} \tag{B.59}$$

$$\omega^2 s_c = C_1(\lambda + z_1)(\lambda + z_2) + C_2\lambda(\lambda + z_2) + C_3\lambda(\lambda + z_1) \tag{B.60}$$

$$\omega^2 s_c = \lambda^2(C_1 + C_2 + C_3) + \lambda(C_1(z_1 + z_2) + C_2 z_2 + C_3 z_1) + C_1\omega^2 \tag{B.61}$$

$$\Rightarrow \begin{cases} C_1 + C_2 + C_3 &= 0 \\ C_1(z_1 + z_2) + C_2 z_2 + C_3 z_1 &= 0 \\ C_1 \omega^2 &= \omega^2 s_c \end{cases} \tag{B.62}$$

$$\Rightarrow \begin{cases} C_1 &= s_c \\ C_2 &= -\frac{s_c z_2}{z_2 - z_1} \\ C_3 &= \frac{s_c z_1}{z_2 - z_1} \end{cases} \tag{B.63}$$

$$\frac{\omega^2 s_c}{\lambda(\lambda + z_1)(\lambda + z_2)} = \frac{s_c}{\lambda} - \left(\frac{s_c z_2}{z_2 - z_1}\right)\frac{1}{\lambda + z_1} + \left(\frac{s_c z_1}{z_2 - z_1}\right)\frac{1}{\lambda + z_2} \tag{B.64}$$

Using the inverse Laplace transforms of Equations B.12 and B.45, Equation B.64 can be transformed to the time domain to obtain Equation B.65.

$$\mathcal{L}^{-1}\left\{\frac{\omega^2 s_c}{\lambda(\lambda + z_1)(\lambda + z_2)}\right\} = s_c - \left(\frac{s_c z_2}{z_2 - z_1}\right) e^{-z_1 t} + \left(\frac{s_c z_1}{z_2 - z_1}\right) e^{-z_2 t} \tag{B.65}$$

Combining Equations B.57 and B.65, we arrive at the time domain solution to Equation B.1 for the case of $\zeta > 1$ shown in Equation B.66.

$$\boxed{\begin{aligned} s(t) = s_c &+ \left(\frac{\dot{s}(0) + (2\zeta\omega - z_1)s(0) - s_c z_2}{z_2 - z_1}\right) e^{-z_1 t} \\ &- \left(\frac{\dot{s}(0) + (2\zeta\omega + z_2)s(0) - s_c z_1}{z_2 - z_1}\right) e^{-z_2 t} \\ \text{where} & \\ z_1 &= \omega(\zeta - \sqrt{\zeta^2 - 1}) \\ z_2 &= \omega(\zeta + \sqrt{\zeta^2 - 1}) \end{aligned}} \tag{B.66}$$