Adaptive Motion Planning for Autonomous Mass Excavation

Patrick Sean Rowe

CMU-RI-TR-99-09

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Robotics

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

January 28, 1999

Copyright © 1999 by Patrick Sean Rowe. All rights reserved.

Abstract

Autonomous excavation has attracted interest because of the potential for increased productivity and lower labor costs. This research concerns the problem of automating a *hydraulic excavator* for *mass excavation*, where tons of earth are excavated and loaded into trucks. This application is commonly found in many construction and mining scenarios. In such applications, fast operational speed of these machines is desired, because it directly translates to increased productivity.

A hydraulic excavator can be considered a large, four degree-of-freedom manipulator mounted on a tracked base. The bucket at the end of the manipulator is used for digging and depositing the excavated material into the trucks. A core technology required for automation is the motion planning of the excavator's manipulator. This research focuses on planning the *free motion* of the excavator, which begins after digging a bucket of material, and ends after the material has been deposited and the bucket has been returned to the digging area. The goal is to plan the excavator's motions such that autonomous task performance approaches that of a highly skilled human expert operator working in similar conditions.

Much of the prior research in autonomous excavation has focused on digging and related topics such as soil modeling and bucket-soil force interactions. Only a few researchers have looked into the free motion planning problem within the context of the mass excavation task. Also, much of the autonomous excavation research has concentrated on functionality, where simply digging a full bucket of material is good enough. In contrast, this research has explicit performance goals, placing importance on high productivity. Finally, while other work has been done in the difficult area of controlling large hydraulic machines, not much emphasis has been placed on the motion planning phase.

There are several characteristics about this problem that led to the motion planning approach that is presented in this thesis. The excavator's motions for each bucket loading cycle[1] are highly repetitive and deliberate, almost to the point of being scripted. However, the precise dig, dump, and truck locations do change from cycle to cycle. The hydraulic actuation system of the excavator is power-limited and highly non-linear, making it difficult to model. The operation proceeds quickly, with many buckets being dug and loaded in a short amount of time.

With these characteristics in mind, we have developed a motion planning approach known as *parameterized scripting*. A script describes the task as a series of simple steps. The parameters of the script define both the specific goals for each script step and the transitions between steps. Different script parameter values are computed for each bucket loading cycle based on the current task conditions. The parameter values affect both the operational speed and the accuracy in achieving desired task goals.

Because of the modeling difficulties, the script parameter values are computed using information about the excavator's own performance, which is gathered on-line during task execution. The

1. A loading cycle consists of the excavator's manipulator moving the bucket to the truck, dumping the material, and moving back to the dig area.

excavator's performance, resulting from a particular parameter set, is evaluated and stored in a data base. Memory-based learning techniques are used to generalize across parameter sets and find the best set of parameter values for the given task conditions. The vehicle motion itself is also analyzed to help in the search for the best parameter values and rapidly improve task performance.

The adaptive motion planning system has resulted in autonomous performance that approaches a skilled operator in the short term and outperforms him in the long term under our testing conditions. The autonomous excavator's motions are also more accurate and consistent than a human's. The adaptive motion planning approach provides a highly flexible system. Because the adaptive motion planning system uses data gathered on-line, it can be used on any excavator with any control system in any worksite conditions. The excavator can modify its behavior to achieve maximum productivity in its current working environment.

Acknowledgments

First and foremost I would like to thank my research advisor Tony Stentz. Tony gave me the direction and encouragement that was needed to keep this work on track, and was always very enthusiastic every step of the way. I would also like to thank John Bares, a member of my thesis committee and integral part of this research effort, for his advice and support, and Jeff Schneider for taking the time to answer my many questions concerning machine learning.

Together, John and Tony led the Autonomous Loading System project which inspired and funded this work. This large, four year project was by all accounts a great success and pushed the bounds of technologies that are needed to make autonomous machines a reality. I would like to thank all of the team members over the years, Stephannie Behrens, Scott Boehmke, Howard Cannon, Lonnie Devier, Jim Frazier, Tim Hegadorn, Herman Herman, Al Kelly, Murali Krishna, Keith Lay, Chris Leger, Bob McCall, Rich Moore, Jorgen Pedersen, Chris Ravotta, Les Rosenberg, Wenfan Shi, Sanjiv Singh, and Hitesh Soneji for their support and dedication.

This work also could not have been completed without the help of my friends and colleagues who served in the combined roles of office mates, sounding boards for new ideas, softball teammates, dinner companions, study groups, game (both computerized and non-computerized) opponents, interesting conversationalists, and other necessary distractions. To the RoboGrads of my incoming class, Zack, Terry, Murali, Andy, Daniel, Lisa, Henry, Sundar, and Dongmei, I wish the best of luck in their own research endeavors and future careers. To the folks at the Robotics Engineering Consortium, an off-campus facility where this research project was located, Mike, Jorgen, Tom, and Dave, I just hope they seek the professional help they require.

Finally, I would like to thank my family for their love and encouragement. Without them, I would not be where I am today or where I will be tomorrow.

Table of Contents

Abstract i
Acknowledgments iii
Table of Contents iv
List of Figures vi
List of Tables x

Chapter 1 Introduction 1
1.1 Problem Description 1
1.2 Problem Statement 3
1.3 Problem Characteristics 3
1.4 Problem Approach 5
1.5 Research Issues 6
1.6 Roadmap To This Thesis 7

Chapter 2 Related Work 10
2.1 Autonomous Excavation 10
2.2 Machine Learning 13
2.3 Autonomous Excavation and Machine Learning 18

Chapter 3 Motion Planning: Parameterized Scripts 20
3.1 Preliminaries: Definition of Terms 20
3.2 Initial Motion Planning 21
3.3 Parameterized Scripts 25
3.4 Truck Loading Parameterized Script 34
3.5 Discussion 49

Chapter 4 Task States and Actions 51
4.1 Task States 51
4.2 Actions 53
4.3 Action Decoupling 53
4.4 Truck Loading Task States and Actions 59
4.5 Implementational Details 64
4.6 Discussion 65

Chapter 5 Motion Evaluation: The Reward Function 66
5.1 Execution Time 68
5.2 Task Constraint Error 71
5.3 Combining Time and Error 75
5.4 Example Task Reward Space 76

5.5 Truck Loading Task Rewards 77
5.6 Discussion 86

Chapter 6 Experience Data Base 89
6.1 Experience Data Base 90
6.2 Predictions 93
6.3 Finding the Best Action 94
6.4 Truck Loading Experiences 97
6.5 Discussion 98

Chapter 7 Command Shifting 100
7.1 Command Shifting 101
7.2 Truck Loading Command Shifting 108
7.3 Discussion 110

Chapter 8 Action Selection and the Policy 113
8.1 Action Selector 115
8.2 The Policy 115
8.3 Policy Updates 117

Chapter 9 Experimental Results 119
9.1 Adaptive Motion Planning Example 119
9.2 Example Task: New Task State 128
9.3 Example Task: Interpolation vs. Extrapolation 130
9.4 Example Task: Changing Error Threshold 132
9.5 Autonomous Loading System 135
9.6 Experimental Results 140
9.7 Simulation 149

Chapter 10 Conclusions, Future Work, and Contributions 156
10.1 Conclusions 156
10.2 Future Work 160
10.3 Contributions 162

Appendix A Command Script Parameters 166
Appendix B Locally Weighted Learning Techniques 174
References 177

List of Figures

Figure 1.1: Hydraulic excavator loading trucks in a mass excavation scenario. 2
Figure 1.2: A loading pass consists of digging (left) and dumping or “free” motions (right). .3
Figure 1.3: High level description of the adaptive motion planning approach. 6
Figure 1.4: Block diagram of entire adaptive motion planning system. 7

Figure 2.1: Five examples of memory-based function approximators. 16

Figure 3.1: (Left) Top view showing the implements reference frame and swing angle. (Middle) Side view showing the boom, stick, and bucket angles. (Right) Perspective view showing various other mass excavation terms. 21
Figure 3.2: Velocity profiles of cubic spline trajectories showing their use in finding the minimum execution time. 22
Figure 3.3: Via points and sub-trajectories (shown in Cartesian space) that define the desired path of the bucket tip. 22
Figure 3.4: Only two hydraulic pumps actuate the four joints of an excavator. 23
Figure 3.5: Effects of actuator coupling on the vehicle motion. (Left) Independent motion. (Right) Simultaneous motion. 24
Figure 3.6: (Solid line) The bucket is forced to pass through the via point resulting in slow, awkward motion. (Dashed line) The natural arc of motion if the bucket were not moved at all. 25
Figure 3.7: Block diagram of the adaptive learning system. In bold are the system components that are described in this chapter. 26
Figure 3.8: The Example Task motion. 27
Figure 3.9: Script Dependency Graph for the Example Task. 31
Figure 3.10: Block diagram showing the flow of information when the parameterized script is used to command the vehicle. 32
Figure 3.11: Vehicle motions produced by three different sets of action parameter values. . .33
Figure 3.12: The sequence of vehicle motions for the truck loading task free motion. 35
Figure 3.13: The four truck bed corner points and the desired dump coordinate. 39
Figure 3.14: Schematic of simplified excavator kinematic model. 41
Figure 3.15: Schematic illustrating the boom clearance angle. 41
Figure 3.16: Schematic showing the swing dump angle command parameter. 42
Figure 3.17: Schematic showing the two stick dump command parameters. 42
Figure 3.18: Diagram showing the bucket capture angle command parameter. 43
Figure 3.19: Truck loading task Script Dependency Graph 46
Figure 3.20: Joint traces showing the free motion of the excavator for one loading pass. . . .47
Figure 3.21: Joint traces for a second loading pass with different script parameter values. Notice the change in times between the joint traces of Figure 3.20 and these joint traces. 48

Figure 4.1: Block diagram of the adaptive learning system. In bold are the system components that are described in this chapter. 52
Figure 4.2: Simplest case of a script dependency graph. 54
Figure 4.3: Script dependency graph with two independent action parameters. 55

Figure 4.4: Script dependency graph with two coupled actions. 56
Figure 4.5: Another case of action coupling. 57
Figure 4.6: Script dependency graph with many nodes in between the two motions of Joint 1 58
Figure 4.7: Script dependency graph for the Example Task. 58
Figure 4.8: Truck Loading Script Dependency Graph 60
Figure 4.9: Schematic showing the swing angle that corresponds to reaching the truck. 61
Figure 4.10: Bucket angle parameters for the Dumping Motion task state. 62

Figure 5.1: Block diagram of the adaptive learning system. In bold are the system components that are described in this chapter. 67
Figure 5.2: Joint trace displaying a start event. 68
Figure 5.3: Joint trace displaying a goal event. 69
Figure 5.4: Joint trace showing a failure mode for finding target and goal events. 70
Figure 5.5: Vehicle motions produced by three different sets of action parameter values. . . .71
Figure 5.6: Target-target task constraint example. 72
Figure 5.7: Plots showing the vehicle motions in joint space from the joint traces of Figure 5.5 and how close they come to the task constraint point. 74
Figure 5.8: Contour plots showing the two dimensional action space for the Example Task for one task state. (Top) Contours of the time score. (Bottom) Contours of the error score. 76
Figure 5.9: Joint traces from two sample bucket loading passes. (Left) Slower bucket loading pass. (Right) Faster bucket loading pass. 78
Figure 5.10: Swing and boom joint traces for two different loading passes. 79
Figure 5.11: Joint space path in swing/boom space for two different loading passes. 80
Figure 5.12: Swing and bucket joint traces for two different loading passes. 81
Figure 5.13: Joint space path in stick/bucket space for two different loading passes. 82
Figure 5.14: Joint space path in swing/bucket space for two different loading passes. 83
Figure 5.15: Swing and boom joint traces for two different loading passes. 84
Figure 5.16: Joint space path in swing/boom space for two different loading passes. 85
Figure 5.17: Swing and stick joint traces for two different loading passes. 86
Figure 5.18: Joint space path in swing/boom joint space for the Boom Up subtask. (Solid line) Path produced by the parameterized script. (Dashed line) Direct path to the correct side of the task constraint point shown by the circle. 87

Figure 6.1: Block diagram of the adaptive learning system. In bold are the system components that are described in this chapter. 90
Figure 6.2: Conceptual diagram of an experience data base. 91
Figure 6.3: Simplified model of a task state-action slice from the experience data base. 91
Figure 6.4: Time and error reward plots for the Boom Up subtask. 94
Figure 6.5: Multi-resolution search technique. 95
Figure 6.6: Valid experience data base search ranges for the actions are defined by existing experiences. 96

Figure 7.1: Block diagram of the adaptive learning system. In bold are the system components that are described in this chapter. 101
Figure 7.2: The three events that are needed for command shifting. 103

Figure 7.3: Shifted swing and bucket joint traces. The three detected events have been aligned. 104
Figure 7.4: Swing/Bucket joint space showing the joint space path of the Example Task motion. 107
Figure 7.5: Swing and bucket joint traces for the Example Task. (Left) Predicted joint traces produced by command shifting. (Right) Actual joint traces produced by executing the task motion using the values of the action parameters found by command shifting. 107
Figure 7.6: Joint traces from an actual truck loading task motion. The 12 events that have been detected are shown as the vertical lines. 109
Figure 7.7: Joint traces from an actual truck loading task motion using the action parameter values found from shifting the traces from Figure 7.6. 109
Figure 7.8: Swing and stick joint traces showing problems with command shifting for coupled motion. 111

Figure 8.1: Block diagram of the adaptive learning system. In bold are the system components that are described in this chapter. 114

Figure 9.1: Time and error scores for the two separate task states. 128
Figure 9.2: Action parameter values for each task execution cycle. 128
Figure 9.3: Values of the action parameters for different task state values. (Left) The values for the extrapolation test are all default actions. (Right) The values for the interpolation test change because previous actions influence future actions. 132
Figure 9.4: Time and error scores from the Example Task algorithm showing the effects of adjusting the error threshold value. 133
Figure 9.5: Action parameter values for the three different error thresholds. 134
Figure 9.6: Hydraulic excavator testbed. 135
Figure 9.7: Autonomous Loading System software architecture. 136
Figure 9.8: Time line showing the evolution of the adaptive motion planning system. 139
Figure 9.9: (Left) Excavator digging a bucket of soil. (Right) Excavator dumping the bucket of soil in the pit. 140
Figure 9.10: Plot comparing automated system performance with human performance for the dump pit task. 141
Figure 9.11: (left) Excavator after digging a bucket of soil and moving towards the truck. (right) The truck is loaded with six buckets of soil. 142
Figure 9.12: Diagram showing the different dig locations and truck parking locations for the truck loading experiments. 143
Figure 9.13: (left) Free motion time for each truck. (right) Total truck loading time including dig times for each truck. 144
Figure 9.14: Plots of error scores for each bucket loading cycle. 144
Figure 9.15: Joint traces of the excavator joints for the first truck, using default actions, and the last truck, using the best actions found by the adaptive motion planning system. 146
Figure 9.16: Graphs of action parameter values for one loading cycle over the course of ten trucks. 147
Figure 9.17: Chart showing what percentage of total actions for the policy were chosen by the different action selection methods. 147

Figure 9.18: Excavator joint traces loading one truck with six buckets. (left) Human expert. (right) Adaptive motion planning system. 149
Figure 9.19: Simulated work site set up for side loading configuration. 150
Figure 9.20: Graph of free motion loading times for each truck for simulated side loading. 151
Figure 9.21: Task constraint errors for simulated side loading. 151
Figure 9.22: Same-level loading using simulator 152
Figure 9.23: Graph of free motion loading times for each truck for simulated side loading. 153
Figure 9.24: Task constraint errors for simulated side loading. 153
Figure 9.25: End loading using simulator 154
Figure 9.26: Graph of free motion loading times for each truck for simulated end loading. 154
Figure 9.27: Task constraint errors for simulated end loading. 155

Figure 10.1: Other types of construction, mining, forestry or industrial machines where the use of the adaptive motion planning system would be beneficial. (pictures taken from (Bruun & Keith, 97)) 163
Figure 10.2: More construction, mining, forestry or industrial machines where the use of the adaptive motion planning system would be beneficial. (pictures taken from (Bruun & Keith, 97)) 164

Figure A.1: Schematic of simplified excavator kinematic model that is used in the following command parameter computation sections. 167
Figure A.2: Schematic showing one method of computing the angle of the boom that guarantees clearance of the truck. 167
Figure A.3: The closest distance from the implement reference frame to the truck is a line that is perpendicular to the truck bed wall. 168
Figure A.4: Schematic illustrating a second way of computing the boom clearance angle that takes the lateral position of the truck into account. 170
Figure A.5: Schematic showing the swing angle computed from the desired deposit location. 171
Figure A.6: Two imaginary lines are drawn down the length of the truck bed to position the wrist joint for the two-step dumping maneuver. 172

List of Tables

Table 3.1: Example Task Swing Script States 28
Table 3.2: Example Task Bucket Script States 28
Table 3.3: Example Task Swing Script Rules 29
Table 3.4: Example Task Bucket Script Rules 29
Table 3.5: Example Task Script Parameters 30
Table 3.6: Truck Loading Swing Script States 36
Table 3.7: Truck Loading Boom Script States 37
Table 3.8: Truck Loading Stick Script States 37
Table 3.9: Truck Loading Bucket Script States 37
Table 3.10: Truck Loading Swing Script Rules 38
Table 3.11: Truck Loading Boom Script Rules 38
Table 3.12: Truck Loading Stick Script Rules 38
Table 3.13: Truck Loading Bucket Script Rules 38
Table 3.14: Truck Loading Script Parameters 44
Table 3.15: Truck Loading Script Parameter Values 47
Table 3.16: Truck Loading Script Parameter Values 48

Table 4.1: Example Task State Variables 53
Table 4.2: Example Task Actions 53
Table 4.3: Boom Up Task State and Action variables 61
Table 4.4: Dumping Motion Task State and Action variables 62
Table 4.5: Stick Dig Task State and Action variables 63
Table 4.6: Boom Down Task State and Action variables 64

Table 6.1: Data base of three Example Task experiences. 92
Table 6.2: Truck Loading Task Error Thresholds 97
Table 6.3: Extrapolation step values for truck loading actions 98

Table 7.1: Example Task events for each task constraint. 103
Table 7.2: Times of the events for the Example Task joint traces. 104
Table 7.3: Truck loading events for the command shifting function. 108

Table 9.1: Values of Variables for Task State 1 122
Table 9.2: Task execution times and error scores for the extrapolation and interpolation tests. 131
Table 9.3: Human operator’s truck loading times. 148

Chapter 1 Introduction

The automation of large construction and earth-moving machines holds the promise of higher productivity, increased safety, and lower operational costs. Higher productivity results from autonomous earth-moving machines working at peak performance levels all of the time. Automation of these machines eliminates the issues of human fatigue, required rest breaks, and decreasing job performance over the course of a work shift. The ability of autonomous machines to work in areas that are hazardous or inaccessible to humans, such as toxic or radioactive environments, decrease potential safety risks. The majority of machine-related accidents occur when the operator is mounting and dismounting the machine, so simply removing the human from the machine can improve safety statistics considerably. Enhanced sensing and abilities to reason about the machine's own actions can also reduce unsafe conditions. Lower operational costs are realized by having one human operator supervise several autonomous earth-moving machines simultaneously, as opposed to requiring a separate operator for each machine. The concept of autonomous earth-moving machines has sparked intense interest from manufacturers and customers because of the massive potential financial wins in the multi-billion dollar construction, mining, and quarrying industries.

1.1 Problem Description

Figure 1.1 shows an example of the type of earth-moving machine and task that this research addresses. The machine is a *hydraulic excavator*, which can be considered a four degree-of-freedom manipulator mounted on a mobile tracked base. The bucket at the end of the manipulator is used for digging and transporting material to where it is unloaded, for instance in a dump truck as shown in the figure.



Figure 1.1: Hydraulic excavator loading trucks in a mass excavation scenario.

The task for the hydraulic excavator is *mass excavation*. In a typical mass excavation application, large amounts of earth are excavated and loaded into trucks that haul it away. This repetitive process is performed quickly, day and night, in most weather conditions and in a variety of materials from loose sand to hard rock, with a skilled operator achieving a throughput of hundreds of trucks per day. In highly efficient operations the excavator is never waiting on a truck to load, as shown in Figure 1.1. This specific type of mass excavation application is also referred to as *truck loading*.

There is a clear need for these automated machines to operate at peak performance levels in order to achieve maximum productivity. Even a few seconds of wasted time on each bucket's motion can add up considerably over the course of a work shift. Therefore, optimal motion planning for the excavator's manipulator is essential for an automated mass excavation system.

It is also desirable to provide the ability for these automated machines to operate in a variety of work environments and worksite geometries. The trucks can be positioned and oriented anywhere within the excavator's workspace. The worksite in Figure 1.1, for instance, shows trucks which are significantly lower in elevation than the excavator, and whose truck beds are perpendicular to the excavator's manipulator when loading. Another common loading strategy is to position the trucks such that the truck beds are parallel to the manipulator. The trucks could also be parked on the same grade as the excavator's tracks depending on the topology of the worksite.

There can be differences in machine characteristics as well. Hydraulic excavators come in a variety of sizes and types of hydraulic systems. There can even be differences among excavators of the same model, or changes that occur as the machines age. This can result in different dynamic properties of the excavator, which may affect the motion planning strategy. Therefore, another requirement for an automated mass excavation system is operational flexibility so that maximum productivity can be achieved in many different vehicle and worksite configurations.

1.2 Problem Statement

A key technology that is needed for an automated mass excavation system is motion planning for the excavator's manipulator. There are two distinct motions involved in a complete loading pass of the bucket: *digging*, which fills the bucket with material, and *dumping*, which deposits the material and returns the bucket to the dig region for another load. Examples of these motions are shown in Figure 1.2. The dumping portion of the motion is also referred to as the *free motion,* as it is the portion of the loading cycle when the bucket is not in contact with the ground.



Figure 1.2: A loading pass consists of digging (left) and dumping or "free" motions (right).

This research focuses on planning the free motion portion of the excavator's loading cycle. More specifically, for each loading pass, once the digging portion of the loading cycle is complete, the problem is to plan a sequence of commands for the excavator's manipulator that deposits the excavated material at a specified location and returns the bucket to another specified location at the dig region for the next load. This command sequence is sent to the joint controllers of the autonomous excavator. Information about the work environment, such as the location and size of the truck, the desired dump location and the next desired dig location is assumed to be available and used in the motion planning process.

The free motion planning algorithm has the goals of: 1) planning the excavator's motions such that it safely and successfully completes its task, 2) providing operational flexibility so it can be used on a wide variety of machine types and worksite topologies, and 3) achieving maximum productivity in the given working conditions.

1.3 Problem Characteristics

There are a number of problem characteristics that motivate the approach to plan the excavator's

free motion. These characteristics arise from the type of machine that is used and the environment in which the task is performed. These characteristics define a class of problems, of which mass excavation with a hydraulic excavator is one example, which can be solved with the motion planning approach that is presented here.

The following characteristics concern the nature of the task and the task environment.

- The task motion of the excavator is very deliberate. The motion is highly repetitive and rarely deviates from a set sequence of motions for each loading pass.

- For a given worksite topology, the excavator only works in a small fraction of its total workspace. These areas do not change over the course of the task. For example, it is assumed that the excavator always digs in the same area relative to itself, and the trucks always park in the same local area. The excavator does reposition its base once the surrounding soil is excavated, but the digging and dumping regions move along with it.

- Within these local dig and dump regions, however, there are slight changes. The trucks do not always park in exactly the same position. Obviously, the precise digging and dumping locations must differ between loading passes as the material is excavated away and the truck is progressively loaded.

- The workspace is generally obstacle free. The locations of the obstacles that are present, namely the truck and unexcavated terrain, are assumed to be known and stationary.

- Ideally, the task is performed very quickly with hundreds of buckets being excavated over the course of a day's work. This implies that motion plans must be computed quickly, and that a large amount of task performance data can be gathered in a short period of time.

- While it is true that during actual loading of a truck the motion planning system is fully engaged, there are also times, such as waiting for the next truck to park, when computational time is available and could be taken advantage of.

The following characteristics concern the type of machine that is used to perform the task.

- Excavators are hydraulically actuated. This fact raises the issues of power limitations, slow or delayed dynamic response times, and actuator coupling of different degrees of freedom since the hydraulic system is self-contained and interconnected.

- The coupled hydraulic actuation, as well as very large dynamic payloads, makes it difficult to derive analytical models of the excavator and its interaction with the excavated material. This may be problematic for optimal motion planning algorithms, which require accurate dynamic models of the robot.

- Control of hydraulic machines, particularly very accurate trajectory tracking, is also a difficult problem, partially because modeling is a difficult problem.

- As mentioned before, hydraulic excavators come in a variety of sizes and hydraulic systems resulting in different dynamic characteristics between machines.

1.4 Problem Approach

Because of the deliberate, repetitive nature of the mass excavation task, this research has developed a script-based approach to planning the free motion of the excavator. The complete truck loading task is decomposed into a series of simple steps. These steps form a *script*, which define the sequence of motions of the excavator for one bucket loading pass to the truck and back.

The script is parameterized in order to deal with changing conditions for each individual loading cycle, such as different dig, dump, and truck locations. A set of variables, known as *script parameters*, define the precise goals for each step of the script. New values for the script parameters are computed for each bucket loading pass based on the current task conditions.

The parameterization of the script helps to satisfy the goal of operational flexibility. While the general machine motions remain the same for a given task, different script parameters are computed for different worksite topologies. For example, consider the scenario of loading trucks that are lower in elevation than the excavator, as shown in Figure 1.1, versus loading trucks that are at the same elevation as the excavator. In both cases, the general motion of raising the bucket above the truck is required, however the precise height to raise the bucket is determined by the value of the script parameters, which in turn are determined by the location of the truck.

The scripting approach also eliminates the need for a precise trajectory tracking controller, since explicit trajectories are not generated. Rather, individual degrees of freedom are controlled by the script, requiring only independent joint controllers. The trajectory of the bucket emerges from the script sequence. This trajectory-free planning approach is viable since the work environment is generally free of obstacles.

The values of the script parameters directly affect the task performance. In order to achieve the goal of maximum productivity, it is important that good sets of script parameters are found. Because of the difficulties with modeling the excavator's dynamics, and taking advantage of the large number of loading passes performed over the course of a work shift, this research has developed an adaptive approach to computing the values of the script parameters. Information about the excavator's current task performance is recorded and used to improve future task performance. This information is gathered on-line as the excavator actually performs its task.

Figure 1.3 shows a high level description of the adaptive motion planning approach. For each loading pass, a motion plan is constructed by the Motion Planner module based on the current task conditions. This motion plan consists of a set of script parameter values that are filled in to the parameterized script. The plan is executed by the excavator, and the motion is evaluated by the Motion Evaluator to determine how well the script parameters achieved the desired task goals. This information is used by the Motion Plan Updater to update the Motion Planner so that future motion plans better achieve the desired task goals. In the case of the truck loading task, these task

goals include maximum speed of operation and successful completion of each loading task assignment. This cycle continues over the course of the task.

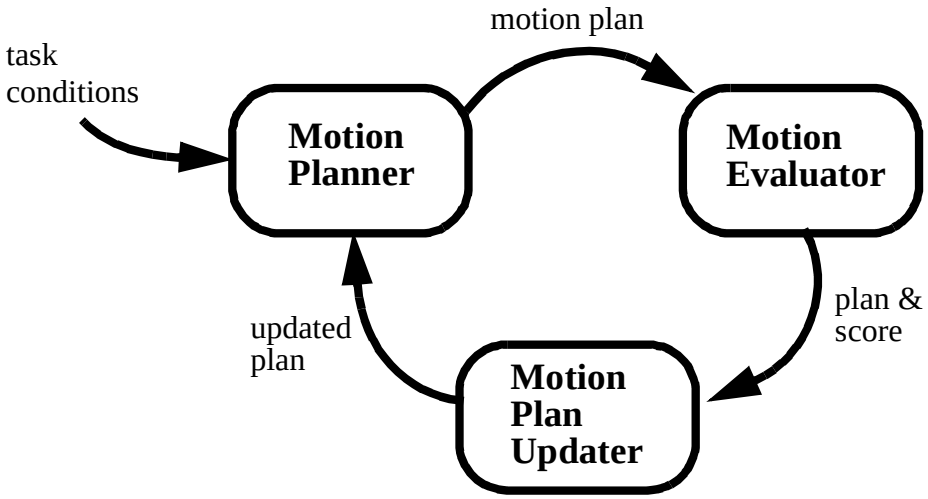


Figure 1.3: High level description of the adaptive motion planning approach.

The adaptive motion planning approach achieves the goals of maximum productivity and operational flexibility. Because information about the task performance is gathered on-line, no a priori information about the worksite topology or machine characteristics is required. The excavator can start performing its task with no information and “learn” as it goes. The Motion Plan Updater shown in Figure 1.3 is performing an optimization in the space of the script parameters using the on-line evaluation information. This means that the system can automatically adapt the excavator’s motion to achieve the best task performance on the current working conditions.

1.5 Research Issues

This thesis seeks to solve a real-world problem for a real application and strives to meet several performance goals. Much of the research that is presented involves dealing with the constraints that are imposed by the real-world nature of the task. These issues are listed below.

- *Large state and action spaces*: The spaces of possible task conditions and scrip parameters are large. How can efficient exploration of these spaces be performed so that task improvement can be realized in a reasonable amount of time and number of trials?

- *Continuous state and action spaces*: These spaces are also continuous. What sort of learning or optimization approaches are best suited?

- *No vehicle models*: In order to achieve the goal of operational flexibility, it was decided that

no a priori information concerning the vehicle dynamics would be used. What are the advantages and disadvantages of such a decision?

- *Efficient use of available data*: Because the system is only using information that is gathered on-line, how can the available information be used to its fullest potential? How can all of the information be used to increase the rate of performance improvement?

- *Safety*: Robotic machines that exist in the real world have the potential to do damage to themselves or other objects[1], so care must be taken in selecting actions for the robot. How can the motion planning system learn to optimize its motions safely, and what limitations does this place on the approach?

- *Conflicting goals*: Generally speaking, the goals of performing a task quickly and performing a task safely can conflict. How are these conflicting goals handled for this problem?

1.6 Roadmap To This Thesis

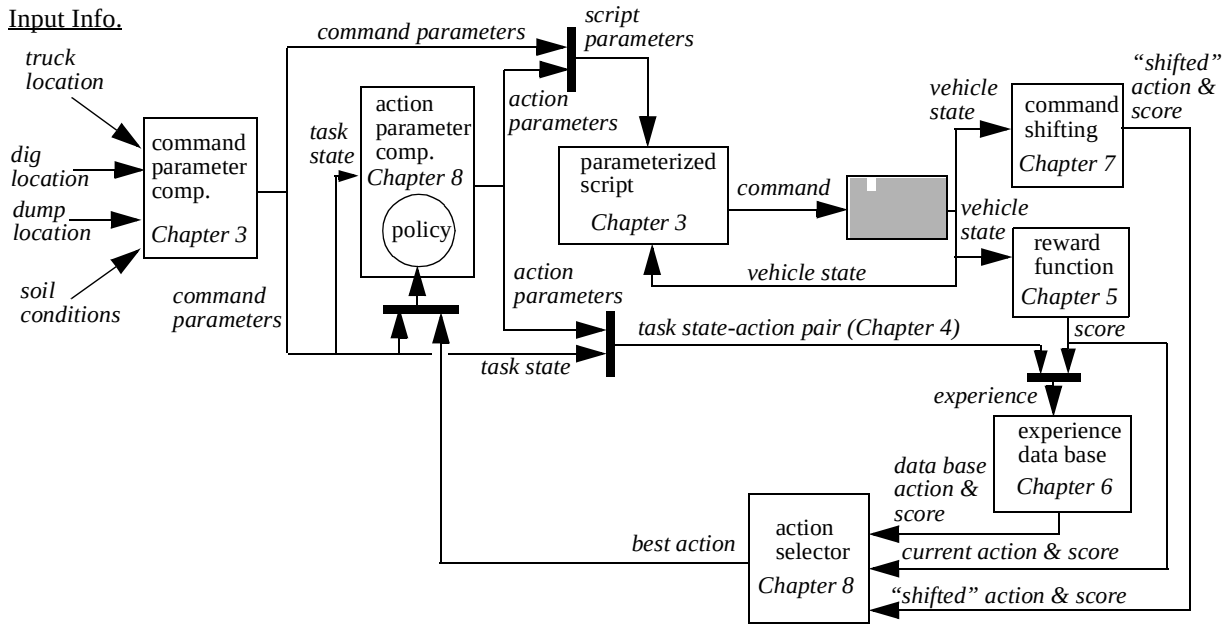


Figure 1.4: Block diagram of entire adaptive motion planning system.

Chapter 2 describes other research that relates to the adaptive motion planner presented here. Chapters 3 through 8 present the adaptive motion planning approach.

1. This is especially true when dealing with a 25 ton hydraulic excavator.

Figure 1.4 shows a diagram of the entire adaptive motion planning system. It is not intended that this figure be understood immediately, as the diagram will be built piece by piece over the course of the document.

The system begins in the upper lefthand corner of the diagram, which shows the input information that is provided to the adaptive motion planning system. Chapter 3 describes the *parameterized script* motion planning approach shown in the middle of the diagram.

Chapter 4 introduces the concepts of *task states* and *actions*, two definitions required for the adaptive motion planning approach. Once the motion plan has been executed on the excavator, the system proceeds to the right in the diagram as the excavator's motions are analyzed and scored. Chapter 5 describes how the motion evaluation is performed by the *reward function* and introduces the idea of *task constraints*, which specify the desired task goals.

Moving down from the reward function, once the motion plan has been evaluated, it is then stored in the *experience data base*. Chapter 6 describes this system component and how it is used to predict the score of an untried motion plan. The experience data base is also used to find the best motion for the given task conditions based on what it knows so far.

A key to rapid improvement of the adaptive motion planning system is the use of a powerful heuristic called *command shifting*. The command shifting function, which is presented in Chapter 7, analyzes the vehicle state information from each loading pass. From this information it can also find the best motion to take for the given task conditions.

Finally, Chapter 8 closes the loop on the adaptive motion planning system. The best motion plans that are suggested by the experience data base and the command shifting function, along with the current motion plan, are sent to the *action selector.* The action selector selects the best motion plan among the three and stores it in the *policy.* The policy provides rapid look-up of the best motion to take for the given task conditions. The action that the policy returns becomes part of the script parameters which are sent to the parameterized script. The policy is continually updated as new experiences are gathered and the motion plans approach the optimal ones.

Along with an introduction and a discussion at the end of each chapter, Chapters 3 through 8 are divided into two main sections. The first of the two sections presents each of the major concepts of the adaptive motion planning system in the context of a simple Example Task. The Example Task is a small piece of the overall truck loading motion and will be continued throughout the chapters to illustrate various aspects of the system. The second section of each chapter presents the system components as they were implemented for the truck loading task. The chapters have been divided in this way so that readers who only wish to understand the basics of the adaptive motion planning algorithm itself can do so by reading the first section and avoid being overwhelmed by the implementational details of the real system.

Chapter 9 presents the experimental results for the different testbeds that were used to develop and test this research. The primary testbed is a 25 ton hydraulic excavator that loads trucks in a mass excavation scenario. Other worksite scenarios were tested using a simulated hydraulic excavator. Comparisons to human performance are also provided.

Finally, Chapter 10 presents the conclusions, future directions of work, and contributions to the field.

Chapter 2 Related Work

The adaptive motion planning approach that is presented in this thesis is an application of machine learning to the field of autonomous excavation. This chapter describes related work in both of these areas focusing on research that is particularly appropriate to this work. Section 2.1 discusses some of the research that has been performed in the area of autonomous excavation. Section 2.2 describes relevant work in the vast field of machine learning. Finally, Section 2.3 discusses work done at the intersection of these two areas and describes other ways machine learning has been applied to autonomous excavation.

2.1 Autonomous Excavation

A great deal of research has been performed on autonomous excavation, so much so that it can be considered its own category in the area of field and outdoor mobile robots. There exists a survey paper describing the state of the art in autonomous earth-moving that details many of the relevant topics including soil modeling, excavator kinematics and dynamics, tele-operation, motion planning, tactical dig planning, and other autonomous earth-moving systems (Singh, 97).

The vast majority of research in autonomous excavation has focused on digging and related topics, which include soil modeling, soil-tool interaction, and planning both where to dig and how to dig at various levels of autonomy (Bisse, 94; Bullock et. al., 90; Huang and Bernold, 93; Rocke, 94; Rocke, 95; Sakai and Cho, 88; Sameshima and Tozawa, 92; Seward et. al., 92; Singh, 95; Singh and Cannon, 98; Shi et. al., 96).

Another research topic of interest has been in machine control (Lawrence et. al., 95; Song and Koivo, 95; Krishna and Bares, 99). Accurate control of hydraulic machines pose different prob-

lems than control of standard electrically actuated manipulators, in particular because of the difficult modeling problem. The hydraulic systems are self-contained on the vehicle, which means that there are power limitations. Furthermore, the hydraulic systems are interconnected resulting in highly non-linear dynamics of the actuation system.

Because the work that is presented in this thesis was guided by the larger goal of automating the entire mass excavation task, this research focused on digging's counterpart, which is the free motion of the excavator's manipulator. Furthermore, this research concentrated on *planning* the free motions as opposed to controlling the machine's end-effector along a pre-planned trajectory. Many of the autonomous excavation research systems value functionality over performance, especially in rocky and other difficult digging conditions. The work that is presented here has the goal of planning the excavator's motions such that the autonomous excavator's performance approaches or equals that of a highly skilled human expert working in similar conditions. As yet, not much research in autonomous excavation currently exists in these areas and for these goals.

2.1.1 Motion Planning for Autonomous Excavation

Although much of the motion planning research in autonomous excavation has concentrated on the digging task, it does offer some guidelines for solving the free motion planning problem. Many researchers have found that because of the difficulty in modeling the highly non-linear vehicle dynamics and the soil-tool interactions, planning an explicit trajectory for the bucket is not practical. Instead, several autonomous systems have developed rule-based methods to control the excavator.

At the University of Arizona, researchers have used fuzzy logic as a control strategy. (Lever et. al., 94; Lever and Wang, 95; Shi et. al., 95; Shi et. al., 96). They argue that the sensory information received, such as force/torque data, is unstructured and difficult to analyze due to large variations in the interactions between the digging tool and the environment. This makes other control strategies inadequate.

The fuzzy logic rules were developed from human experience and heuristic means. The inputs to the fuzzy rules are force and torque information experienced at the bucket. The outputs are the control signals sent to the excavator's manipulator and are computed by defuzzifying the variables *bucket-horizontal-step-size*, *bucket-vertical-step-size*, and *bucket-speed*. An example rule is

If M_x is Negative-Large, Then *bucket-horizontal-step-size* is Positive-Large, *bucket-vertical-step-size* is Positive-Large, and *bucket-speed* is Positive-Small.

This rule states that if the torque on the bucket is large and in the negative direction, then the bucket is probably forced against the floor or an obstacle is below it. In this case, the bucket should be moved forward and up at a slow speed.

During excavation, the bucket is commanded to follow a specified path, typically a horizontal line for trenching operations. The fuzzy logic control system provides the ability to deal with immobile obstacles in the path of the bucket, a condition that is often encountered in very hard, rocky

digging environments. We will discuss extensions of this work in Section 2.3, where the fuzzy logic control system has been organized into behaviors, which themselves have been organized into finite state machines.

LUCIE, an autonomous excavation system from the University of Lancaster (Seward et. al., 92; Seward et. al., 96) uses a production system to plan the motions of their excavator. Their primary task is trenching, as opposed to mass excavation where the path of the bucket does not have to be as precise.

Their system contains approximately 70 rules of the form:

```
IF (bucket penetration > 300 mm) then DO (rotate bucket)
```

As in the system from the University of Arizona, these researchers also derived their rules with input from experts in the excavation field. The rules have been designed to handle a wide variety of conditions, so only a few are active at any one time. The bucket attempts to follow a pre-specified path, and the rules allow it to react to various conditions encountered during the excavation process.

The numerical values that are present in the production rules are a similar idea to the parameters in the parameterized script that is described in Chapter 3. However, it is unclear if the numerical values in the production rules, such as 300 mm, change based on the conditions of the worksite. These researchers claim that with properly tuned rules, the system can produce effective digging. It appears as if the numerical values that are in the rules must be tuned by hand and can not adapt automatically to changes.

Other rule-based systems exist for planning the motions of the excavator's manipulator, but with a slight difference from the previous two. In the previous two systems, the bucket is given a specified path to follow, and the fuzzy rules or production rules allow it to modify its behavior based on conditions encountered during excavation. Other researchers use rule-based systems to control each degree of freedom individually, having the bucket's path emerge based on which rules where used.

One system developed by Sameshima (Sameshima and Tozawa, 92) is also based on fuzzy logic, however the inputs and outputs are the measured and desired velocities of the manipulator's joints. Other inputs include the bucket angle and the depth of the bucket in the ground. The intuition here is that soil conditions, such as the relative stiffness, can be deduced from the measured joint velocities. Thus, if they are moving slowly, the soil is assumed to be hard to dig, and the appropriate control actions are taken. No trajectory for the bucket is planned. Instead, the bucket is placed at a starting point above the soil and the rule-based control system takes over from there.

A similar system involves computing the desired joint velocities based on hydraulic cylinder pressures rather than measured joint velocities (Rocke, 94; Rocke, 95). Again, the quantity that is being measured indirectly through the cylinder pressures is the stiffness of the soil. Look-up tables are used that map measured cylinder pressure to commanded joint velocity. These tables were constructed by analyzing the digging motions of a human expert operator in different dig-

ging conditions.

The lesson that can be taken away from the existing research is that rule-based systems for planning and controlling the excavator's actions appear to be a successful approach for autonomous excavation. Also, human expert input is essential in constructing effective rules for the excavation control strategy.

The parameterized scripting approach that is presented in this thesis is another in this family of techniques. Although the rules of the parameterized script are slightly simpler than a production system or fuzzy rule-base, the ability to modify the parameters of the script offers more flexibility than other currently existing rule-based systems. One clear advantage of this flexible approach is minimal reconfiguration if the autonomous excavator should change worksites, or if the adaptive motion planning system is to be used on a different machine.

2.2 Machine Learning

The adaptive approach to motion planning can, in many ways, be considered machine learning. The excavator begins its task with a functional set of motions, though slow and minimally productive. By using information that it gathers on-line as it performs its task, it can "learn" how to improve its motions and achieve maximum productivity in its current working conditions.

There are many flavors of machine learning; a description of all of them is beyond the scope of this thesis. The following sections present a few relevant areas of machine learning, emphasizing those systems that have been implemented on real-world robots or machines.

2.2.1 Robot Skill Learning

The work presented in this thesis falls most closely into the area known as *robot skill learning* or *task level learning*. The task of loading trucks with soil can be considered a skill for the excavator to do. It is often the case that the skill should be performed optimally with respect to some metric such as time or expended energy. The distinction of skill or task-level learning is that the robot learns how to perform optimally over a wide range of task conditions, as opposed to optimizing a single trajectory between two fixed points.

Several researchers have developed systems for robotic machines to learn a task or skill. Schneider (Schneider, 93; Schneider, 95) applies robot skill learning techniques to a manipulator that can throw a tennis ball. There are several criteria for success including accuracy in throwing at a target, maximum distance thrown, and minimal control effort.

In his work, many real-world issues come to bear including dealing with high-dimensional spaces of possible motions, and efficient exploration in this space. Schneider developed the Guided Table Fill-In algorithm that ranked robot actions based on how well they performed. Different throwing actions were taken by the robot, evaluated based on the success criteria, and stored in the table. Linear combinations of these actions were used to quickly generate new actions that the algorithm

believed would achieve the desired behavior.

In another example of robot skill learning, Aboaf (Aboaf et. al., 88; Aboaf et. al., 89) has developed a robot that can bat a tennis ball up and down with a paddle for long periods of time. As in Schneider's ball throwing manipulator, experiences are gathered and used to help the robot adjust its future actions, which is the motion of the paddle. In this case, a model of the errors between the predicted flight of the ball, based on a ballistics model, and the actual flight of the ball was constructed from the collected data. This error model was used to correct the paddle's location based on the observed position of the tennis ball. The results were many more successful hits than could be achieved using the ballistic model alone.

Atkeson's devil sticking robot (Atkeson, 91; Schaal and Atkeson, 94) is another example of robot skill learning. Devil-sticking is a form of juggling that involves controlling two manipulators to keep a stick bouncing back and forth between them. Each data point that is gathered during the motion is the state of the stick, the control action that is performed, and the next state of the stick. Atkeson uses locally weighted regression, that is described in the next section, as a technique for model fitting. Locally weighted regression makes predictions by only considering the data within a local neighborhood of the query to the model. The model that is fit to the dynamic information is used to construct an LQ controller for the manipulators. A technique known as the shifting setpoint algorithm is also employed which provide setpoints to the LQ controller. As more and more data is collected, the setpoints change to achieve better performance.

Moore (Moore et al., 95) describes a billiards shooting robot. Here the idea is to sink a billiard ball into a desired pocket. The location of the ball, the action of the pool cue, and the location on the cushion where the ball collided are recorded for each trial. Like the devil sticking robot, a locally weighted model is constructed from this data and searched to find the action for the pool cue that will sink the billiard ball from any starting location.

2.2.2 Memory-based Learning

The examples of machine learning systems described above are all a form of *memory-based learning*. In memory-based learning, all of the information that the robot gathers over the lifetime of the task is remembered. This information is used to construct a model that is used to predict the results of untried actions. The stored data can also be weighted so that data points that are closer to the request have more influence on the predicted result than data points that are not as relevant. An advantage of this approach is that there does not need to be any underlying assumptions about the form of the data. Thus, the model that is fitted to the data can change both as more experiences are gathered and as different regions of the experience data base are used.

This strategy of remembering and keeping all of the data is different from other parameterized generalization techniques such as global regression or neural networks. In those techniques, a set of parameters, such as the coefficients of a linear equation or the weights of a neural network, are computed from the data, and the data is discarded. The parameters are used to make the predictions. There are several advantages and disadvantages of each technique that are listed below.

- Adding a new data point in memory-based learning is trivial; simply add it to the data base. It

is unclear what to do with new data for parameterized techniques. When should new parameters be recomputed?

- However, memory-based learning approaches are slower in making predictions than parameterized approaches. With memory-based techniques, the computation is deferred until the time of a prediction request. Increasing the number of data points and the number of data base dimensions adds to the cost of computing a prediction.

- Memory-based learning approaches do not require an a priori assumption of an underlying model which is required for global regression techniques. Instead, a local model, such as a linear model, is formed at the query point, much like a Taylor series expansion is used to linearize a function around a set point. In this way, complex, non-linear functions can be approximated with memory-based learning techniques.

- Memory-based learning techniques provide automatic mechanisms such as cross validation for finding the best weighting parameters, distance functions, and other components that are needed for the locally weighted function approximator.

- Since all of the data is explicitly stored in memory-based learning, it does not suffer from negative interference that occurs when data from a new part of the input space is collected. In neural networks, for example, this condition could cause a change in the weights, resulting in degraded performance of the original task.

- Other advantages come from the mathematical background of memory-based function approximators. Information about noise, gradients, outliers, and confidence intervals on the predictions are readily available.

There are many different ways to use the experience data to generate a prediction (Moore, 91). Figure 2.1 shows a few of the techniques on a sample one-dimensional data set. In all plots, the x-axis plots the input query, such as an action for the robot to take, and the y-axis shows the predicted result, such as a reward for the action.

On the far left, nearest neighbor is the simplest case. To compute a prediction, the output value of the nearest data point is returned. While nearest neighbor is rather simple, it suffers from the problems of no gradient information and tends to fit to noise rather than smoothing out the data.

The next two plots are averaging techniques. In the globally average case, all data points have equal influence on the output value, so there is no variation in the output. For locally weighted averaging, data points that are closer to the input query are given more weight than data points that are farther away. The result is a curve that looks more reasonable and varies with the input query. This technique is also known as *kernel regression*, where the kernel is a parameter that affects how much relative influence the data points have. The global average is one extreme of kernel regression where each data point has equal weight. The nearest neighbor plot is the other extreme of kernel regression where only the closest data point has an influence on the output.

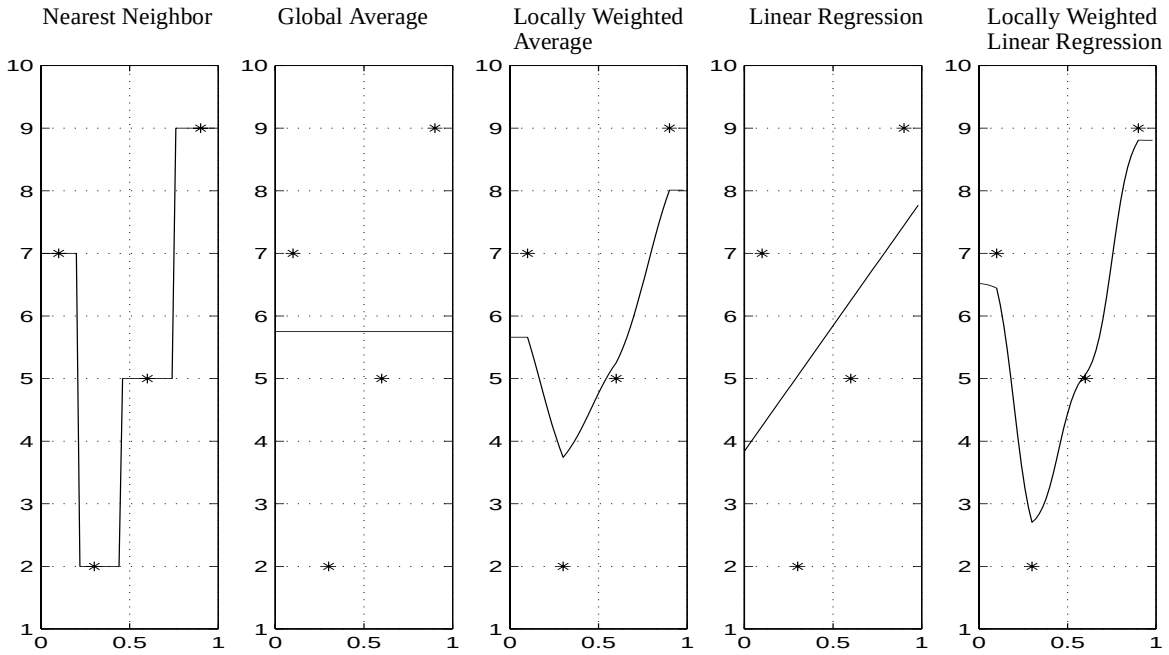


Figure 2.1: Five examples of memory-based function approximators.

At the next level of complexity are linear regression techniques. The fourth plot in Figure 2.1 shows a standard global linear regression, while the fifth plot shows a locally weighted linear regression. Like the locally weighted average, the locally weighted linear regression weights the data points that are closer to the input query. This weighted data is then used in the regression calculation. The result is a curve that does not look very linear at all, but appears to do a good job in approximating the existing data.

2.2.3 Reinforcement Learning

The idea of modifying the robot's behavior based on information that is received during task execution falls into a much larger class of learning problems known as *reinforcement learning*. In reinforcement learning, the robot seeks plans of action that maximize a reward value. A survey paper has also been written on the research in this field (Kaelbling et. al., 96).

Although reinforcement learning has many things in common with robot skill learning, there is one main difference which differentiates the two. Reinforcement learning systems attempt to learn a sequence of actions, perhaps from a selection of action primitives, that will maximize the reward value and achieve the desired goal. Robot skill learning systems already know which actions will do the task. They seek to learn a set of parametric values which will allow the robot to do the task *better.*

However, there are still a number of issues that are addressed by reinforcement learning systems that are relevant to this research. For one, the basic algorithm and system components for any

adaptive system are:

Loop forever:

1) Receive the current state *S* of the world.

2) Determine an appropriate action *A* to take in the given state. One way that this is done is with a policy *P* which maps states to actions $A = P(S)$.

3) Execute the action on the robot and receive a reinforcement value *R*.

4) Use the reinforcement value to update the existing policy.
$P_{new} = U(P, S, A, R)$
where *U* is the policy update function.

It is the burden of the system designer to determine the definitions of states and actions, implement how reinforcement values are computed, and decide on the mechanisms for both selecting an action with the policy and updating the policy. These system components are, of course, dependent on the task for the robot. The core of the research that is presented in this thesis is to define and implement these system components for the task of autonomous mass excavation.

The nature of reinforcement learning problems offer several unique challenges that are not found in other types of learning systems. For one, the learner is never told which is the best action to take for any given situation. Instead, it must determine this by trial and error as it explores its world. This can lead to problems with getting stuck in sub-optimal local minima. Much research has been done on the question of how much to explore one's surroundings versus exploiting the knowledge that has already been gained (Thrun, 98).

Very high dimensional state and action spaces also present problems to reinforcement learning systems. If the spaces are too large, the robot has little hope of acquiring enough samples to do anything useful with. Mataric (Mataric, 94) recommends overcoming problems with large dimensional spaces by taking advantage of readily available domain knowledge. Her task involves mobile robots seeking out and transporting pucks to a designated region. Mataric implemented progress estimators to provide goal-directed "advice" to the robots. For example, one progress estimator is active only when a robot has a puck, and strongly encourages it to take an action to head back to the home base. The idea of using domain knowledge is a powerful one and leads to very rapid learning performance improvement. We use the same idea in this research to provide more information to the robot skill learner than just a single reinforcement signal.

2.3 Autonomous Excavation and Machine Learning

Several researchers have applied a variety of learning techniques to several of the sub-problems of autonomous excavation. These researchers have found learning techniques to work well because of the inherent modeling difficulties with excavation machines. The repetitive nature of the tasks that they perform also provide ample opportunity to collect data from which they can learn.

For machine control, neural networks have been used to model the inverse dynamics of an excavator (Song and Koivo, 95). In Song's work, the inputs to the neural network are three consecutive desired positions from the specified trajectory. The outputs are the joint torques. The neural network, which acts as a feed-forward controller, is combined with a secondary PD feedback controller, which provides corrections during actual motion. The neural network can be updated online using the corrections from the PD controller. These researchers report that the control is much better than a pure feedback controller could provide by itself.

In the topic of tactical dig planning, Singh (Singh, 95) applied learning techniques to predict the resistive forces that are encountered at the bucket during digging. Specifically, Singh implemented and compared three techniques, global regression, memory-based learning, and neural networks, to learn the parameters of the force equation using actual measured bucket force and position data. The force equations are functions of the bucket trajectory, terrain shape, and soil volume. Different basis functions were also tried. Singh reports that using learning greatly improved the predicted forces over a purely analytical model. He then uses the force model to search for optimal digging actions.

The research that is perhaps the closest to this work is from the University of Arizona and was mentioned earlier in Section 2.1. Primitive excavation actions, which are implemented with fuzzy logic controllers, are grouped into higher level behaviors such as *dig-horizontally* or *go-over-immobile-object*. These behaviors are further organized by finite state machines into tasks such as *remove-material-at-defined-elevation.*

The learning comes by placing neural networks within the finite state machines to determine which state, consisting of one or more excavation behaviors, to go to next based. The neural networks are trained using data that was gathered from previous excavation runs. The inputs to the networks are force/torque information and the output is a vote for the next state in the finite state machine. For example, if during digging the force/torque information indicates that the bucket has hit an immobile obstacle with a face that is sloping up, it may be more appropriate for the bucket to move over the object than to try to dig under it. The neural networks have ben trained with this information and act appropriately should the same situation be encountered again.

The work that is presented in this thesis is another application of task-level or robot skill learning to the field of autonomous excavation, and the first known application to the sub-problem of planning the free motion of a hydraulic excavator. This implies that the autonomous system is able to adapt and improve its performance in a wide range of working conditions. It is also the first known work which explores the free motion planning problem with the explicit task goal of maximizing productivity.

The next six chapters describe the technical approach of this work detailing how motion plans are expressed (Chapter 3), the definitions of states and actions (Chapter 4), how reward values are computed (Chapter 5), how the state-action-reward information is used to predict the results of untried actions (Chapter 6), a powerful heuristic that analyzes the motion of the vehicle itself and acts as a guide through the large action space (Chapter 7), and the policy and policy update functions (Chapter 8).

Chapter 3 Motion Planning: Parameterized Scripts

This chapter describes the *parameterized scripting* motion planning algorithm that has been developed to plan the free motion of the excavator for the truck loading task. Starting with this chapter, we begin to build the complete adaptive motion planning system diagram that was shown in Figure 1.4. First, Section 3.1 introduces some of the vocabulary concerning the excavator and its work environment that is used throughout this document. Next, Section 3.2 discusses the problems that were encountered with other manipulator motion planning techniques and that led to a script-based motion planning approach. Section 3.3 presents the parameterized scripting approach and introduces the Example Task. The Example Task is a simpler motion than the truck loading task and is used to illustrate the different components of the adaptive motion planning system. Section 3.4 describes the parameterized script that was created for the truck loading task. Finally, Section 3.5 presents a discussion of the parameterized scripting approach.

3.1 Preliminaries: Definition of Terms

Throughout the following technical chapters, certain terms are used to describe different parts of the excavator and its work environment. Many of these terms are standard in the industry. Particular attention should be paid to the names of the four joints of the excavator, *swing*, *boom*, *stick*, and *bucket*, which is how they are referred to throughout the document. The angular positions of the four joints are symbolized by θ_{sw}, θ_{bm}, θ_{st}, and θ_{bk} respectively and are shown in Figure 3.1. Joint velocities use the symbols $\dot{\theta}_{sw}$, $\dot{\theta}_{bm}$, $\dot{\theta}_{st}$, and $\dot{\theta}_{bk}$. The *vehicle state* is the joint positions and velocities. The boom, stick, and bucket links are collectively referred to as the *imple-*

ments. The implement angles, their base reference frame, and their lines of reference are also shown in Figure 3.1.

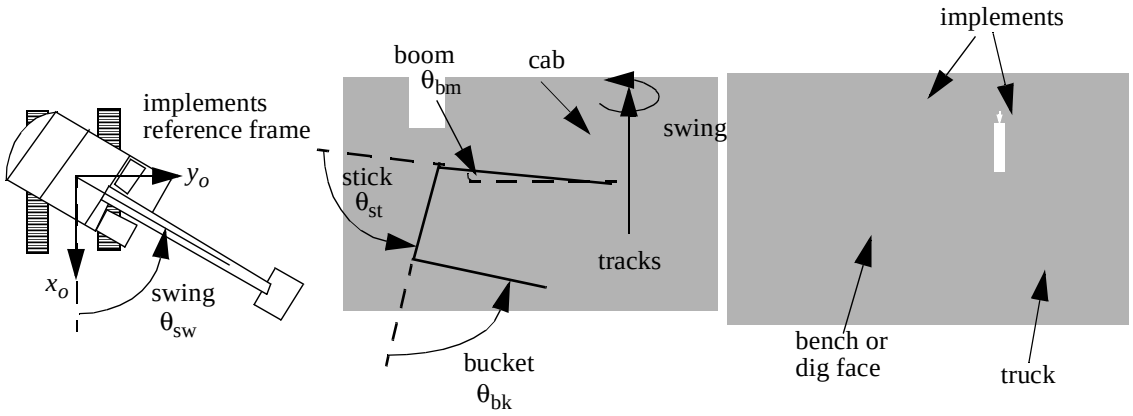


Figure 3.1: (Left) Top view showing the implements reference frame and swing angle. (Middle) Side view showing the boom, stick, and bucket angles. (Right) Perspective view showing various other mass excavation terms.

3.2 Initial Motion Planning

The first attempt at planning the free motion of the excavator was based on a standard method of generating trajectories for robot manipulators using cubic splines. This method can be found in any introductory robotics textbook (Craig, 86). Given a joint's start position and velocity, goal position and velocity, and a time to execute the trajectory, the coefficients of a cubic polynomial are computed that give the desired angular position of the joint as a function of time. The cubic polynomial can be differentiated to get joint velocity and acceleration profiles.

Since performing each loading pass in the minimal time is important, the trajectory execution time, which is required to compute the cubic polynomial's coefficients, is found by using maximum limits on the velocity and acceleration of each of the excavator's joints. Figure 3.2 shows how these execution times are computed. The velocity profile of a cubic spline trajectory is a parabola. As the time to execute the trajectory is shortened, the right edge of the parabola is pushed to the left, and the peak of the parabola moves up (or down for negative velocities). By setting a velocity limit, an upper (or lower) bound can be set on peak of the parabola, and a minimum execution time can be computed. Similarly, minimum times are also found using joint acceleration and deceleration limits, where the acceleration profiles are lines. The maximum of the three minimum execution times, computed for all four excavator joints (12 numbers in all), is taken as the final trajectory execution time.

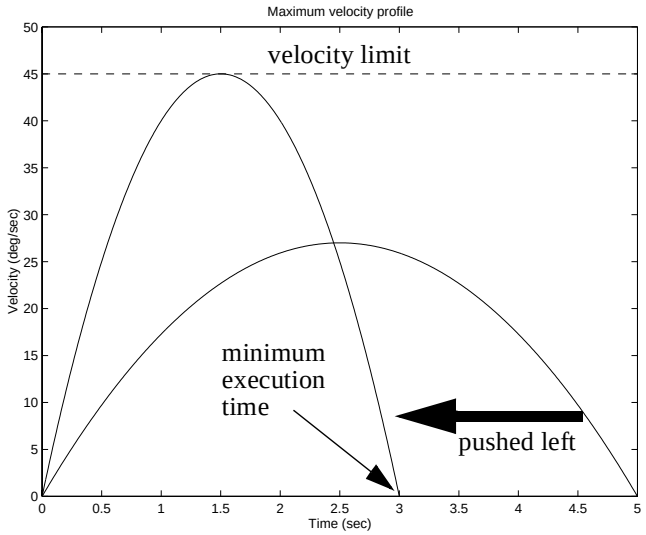


Figure 3.2: Velocity profiles of cubic spline trajectories showing their use in finding the minimum execution time.

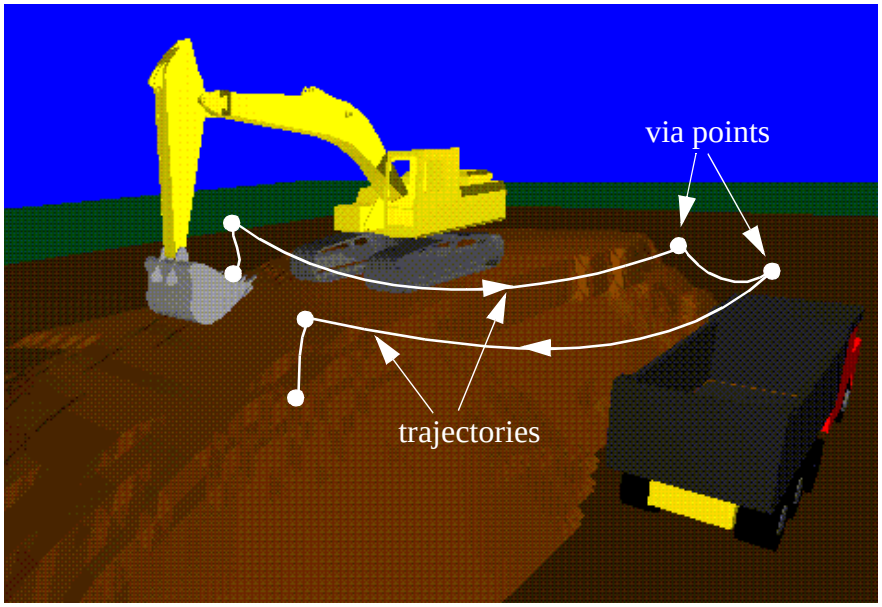


Figure 3.3: Via points and sub-trajectories (shown in Cartesian space) that define the desired path of the bucket tip.

3.2.1 Via Points

The excavator's free motion for the truck loading task is too complex to be expressed as a single cubic spline trajectory. To remedy this, several via points that the excavator's bucket was required to pass through were placed in the workspace as shown in Figure 3.3. Simple heuristics were used

in the placement of these via points. For example, one via point was placed directly above the bucket's initial position, which forced the bucket to raise in order to avoid hitting the truck. Other via points were placed over the truck, defining a desired dumping motion, and back at the dig face specifying the next desired dig point. Sub-trajectories for each joint were then computed between each pair of consecutive via points.

3.2.2 Results

This motion planning method of using cubic splines and via points did not work very well for the truck loading task. The planned trajectories were not tracked with sufficient accuracy, via points were not being passed through, and the loading passes were too slow. The planned trajectories appeared to be forcing the excavator to make motions that were slower than the speeds at which the excavator was capable of moving. There are a number of reasons why this motion planning method failed.

For one, the four joints of a hydraulic excavator are not powered independently. Rather, as shown in Figure 3.4, one hydraulic pump powers two of the joints, the swing and the stick, and a second hydraulic pump powers the other two, the boom and the bucket. When moving one joint at a time, each joint can achieve a certain individual peak velocity. When both coupled joints are moving simultaneously, however, there is not enough hydraulic power to maintain the desired velocities of both joints if the velocities are large.

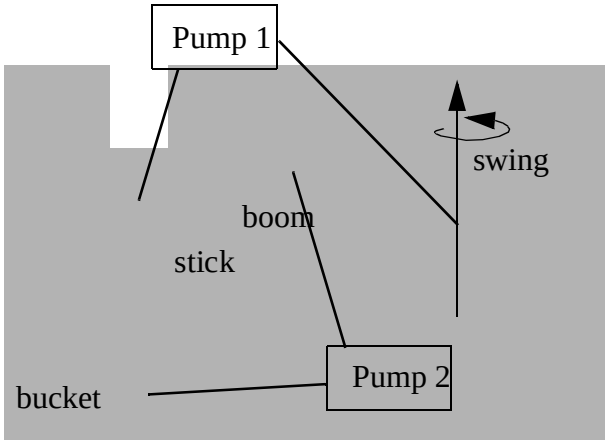


Figure 3.4: Only two hydraulic pumps actuate the four joints of an excavator.

Figure 3.5 shows the effects of this actuator coupling problem. The four plots show an example motion for each joint. The plots on the left show the motions of the joints when all are moving independently, and the plots on the right show the same commanded motions, but now all four joints are moving simultaneously. The two larger joints, the swing and the boom, suffer the most. This is because the hydraulic flow moves along the paths of least resistance, which are the smaller joints with the lowest loads. It takes the swing nearly twice as long to move to the same desired position because the stick is stealing away all of the hydraulic flow from their shared pump. Similarly, the boom does not even begin to move until the bucket has reached its goal position

because there simply is not enough power to move them both simultaneously.

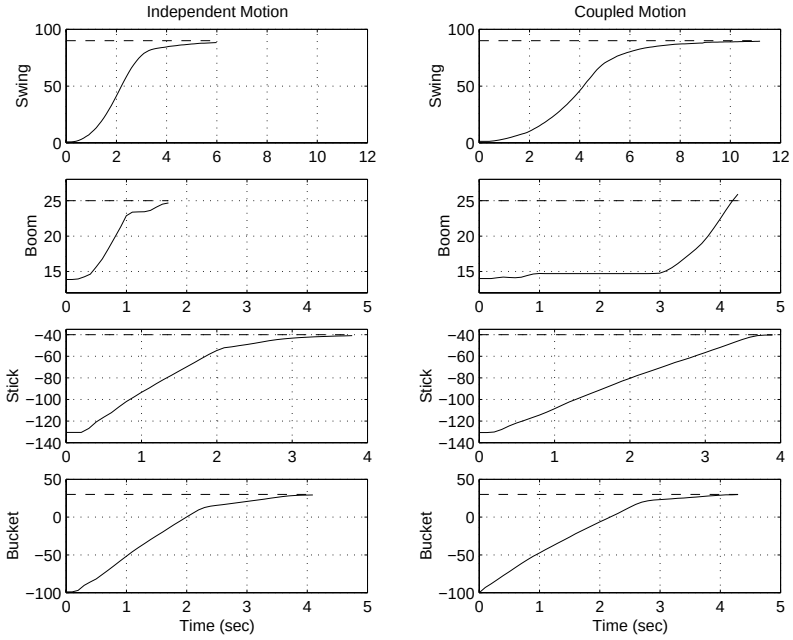


Figure 3.5: Effects of actuator coupling on the vehicle motion. (Left) Independent motion. (Right) Simultaneous motion.

Since minimum trajectory execution times were computed using maximum *individual* joint velocity and acceleration limits, the cubic spline trajectories could not be tracked accurately because all joints were commanded to move simultaneously at or close to their top individual speeds. If vehicle models that captured this coupled motion were available, then it may be possible to use them to plan more realistic trajectories. Initial investigations, however, revealed that these models would be difficult to construct while still performing the computation in an acceptable amount of time, and it was not obvious how they would be used in a real-time planning operation apart from a high dimensional feed-forward search. Only recently has work been done on a sophisticated way of capturing the effects of the coupled motion using memory-based learning techniques (Krishna and Bares, 99).

A second difficulty with the trajectory generation scheme was with the via points. Unlike traditional manipulator path planning problems, for welding or painting robots for example, there is no notion of a desired path through space for the task of loading a truck with soil. Rather, the goal for this task is to deposit the soil at a particular location. Artificial via points were placed in the excavator's workspace in an attempt to create a path to achieve this goal. It was unclear exactly how many via points were needed or where these via points should be, and it was found that poorly placed via points forced the excavator to take a path that was not along its natural motion. This resulted in poor loading times.

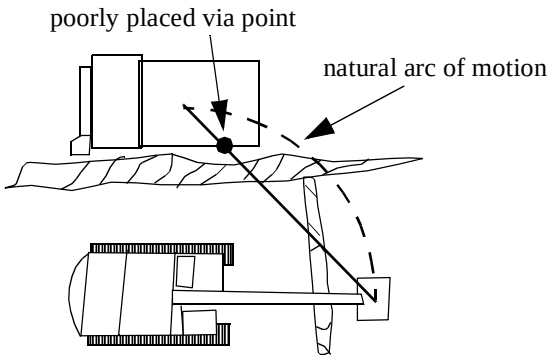


Figure 3.6: (Solid line) The bucket is forced to pass through the via point resulting in slow, awkward motion. (Dashed line) The natural arc of motion if the bucket were not moved at all.

This point is illustrated in Figure 3.6. If a via point is placed over the truck bed wall along a straight line between the dig and dump points, then the excavator is forced to bring in the stick joint wasting valuable time and energy. The circular arc shows the path of the bucket if the stick was not moved at all.

A more practical reason for the failure of the cubic spline approach was the joint controller, supplied with the excavator testbed, was not a true trajectory tracker. It could not receive both a desired position and velocity profile for each joint that would be used in its control law. Instead, it was found that the joint controller operated best when it was given a step command of a desired joint position rather than a trajectory. The initial error between current position and goal is large, which results in a large initial hydraulic flow and a quick and accurate joint response.

The combination of these problems made this kind of trajectory generation scheme insufficient for the truck loading task. In order to achieve the goal of maximum productivity, it was clear that another type of motion planning approach was needed.

3.3 Parameterized Scripts

When examining the motion of an excavator that is loading trucks, it was observed that the general trends of the machine's motion for each loading pass were very similar. The motions always seemed to follow a set pattern, and only a few types of machine motions were needed. For example, after digging a bucket of soil, the excavator raises its boom and, at some point, swings to truck, etc. However, specific kinematic details of the motion, such as exactly how high to raise the boom, or what angle to swing to at the truck, changed from loading cycle to loading cycle. These changes were caused by differences in the truck parking positions, different locations to dump the earth in the truck bed, and different digging locations as the excavator eroded the bench.

The new algorithm that was devised and implemented for planning the excavator's motions for

the truck loading task is referred to as *parameterized scripts* (Rowe & Stentz, 97; Stentz et al., 98). The system components shown in bold in Figure 3.7 are the different pieces of the adaptive motion planning system that involve the parameterized script.

A parameterized script, shown as the box in the middle of Figure 3.7, is a finite state machine with simple rules that determine when to move to the next state in the script. Associated with each state is a vehicle command that is sent to the excavator's joint controller. Thus, changing to a new script state results in a change in the desired positions of the excavator's joints.

The rules of the parameterized script, which define the transitions between script states, are functions of both the vehicle state and *script parameters*. Script parameters are a set of numbers that specify the kinematic details of the motion for each loading pass and comprise *action parameters* and *command parameters*. The action parameters are the product of the adaptive portion of the motion planning system described in Chapter 8. The command parameters are the product of a *command parameter computation* function that transforms information about the work environment to the proper form for the parameterized script. This external information includes the location of the truck, the next desired dig and dump locations, and information about the soil conditions.

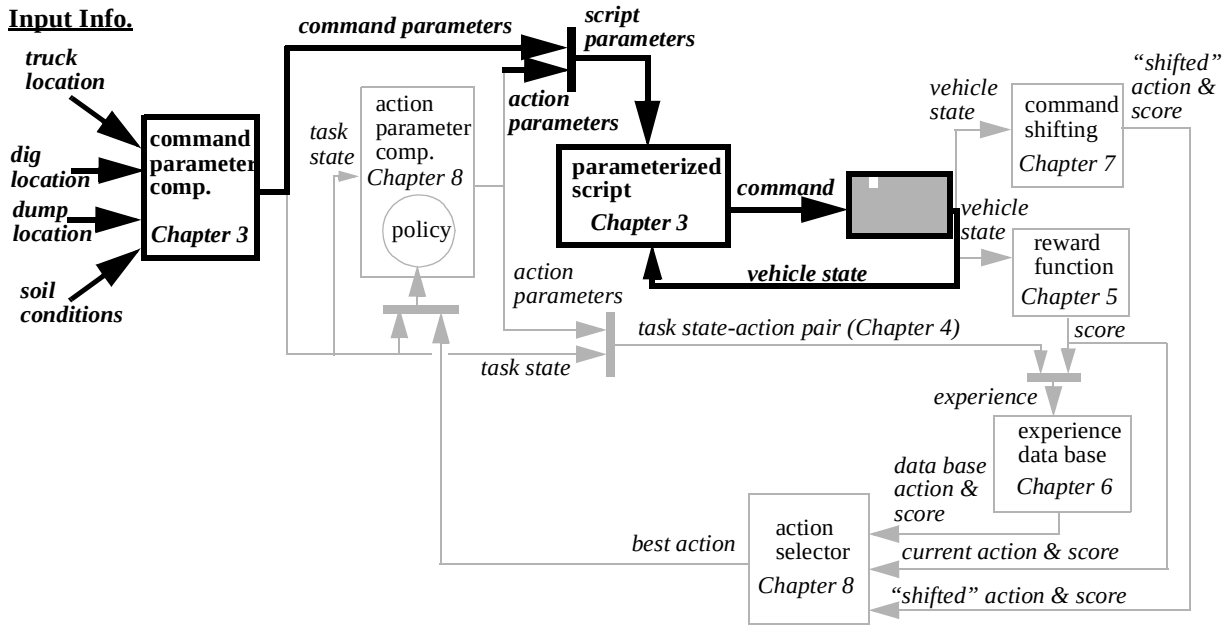


Figure 3.7: Block diagram of the adaptive learning system. In bold are the system components that are described in this chapter.

The next few sections describe the different components of the parameterized script: script states, script rules, and script parameters.

3.3.1 Script States

The states of the script's finite state machine define the sequence of vehicle motions for performing one execution cycle of a task. In the case of the truck loading task, this would be the sequence of the free motion of the excavator for one loading pass of the bucket. The script's finite state machine is reset to the start state for each new bucket loading pass and terminates when the end state is reached. There is a separate script, or finite state machine, for each controllable degree of freedom. For the excavator there would be four separate scripts, one for each joint, that run simultaneously during the task motion.

The states for the truck loading script were created by studying the task and with the input of a skilled excavator operator. This knowledge is reflected in what states are present in the script, which define the types of vehicle motions that are available, and their sequence in the script. For example, if it was advised that moving two particular joints at the same time was not good, then that joint coupling is not present in the script. The script implicitly constrains the possible motions of the machine because the script states specify which joints are commanded to move at any given time.

3.3.2 Example Task

In order to construct a parameterized script, the vehicle must be given a task to do. For this purpose we now introduce the Example Task. The Example Task, which is used to illustrate the different components of the adaptive motion planning system, is a small piece of the overall truck loading motion. The Example Task involves moving the swing and bucket joints only. As shown in Figure 3.8, the implements start from a point over the dig face with the bucket curled as if to capture a load of soil. The swing is commanded to move to a location at the truck. At some point during this motion, the bucket is commanded to open to deposit its material. Once this is done, the swing is commanded to move back to another location at the dig face. The Example Task provides a way of illustrating various aspects of the motion planning system by using simple motions that have less parameters and lower dimensional state spaces than the more complicated truck loading task.

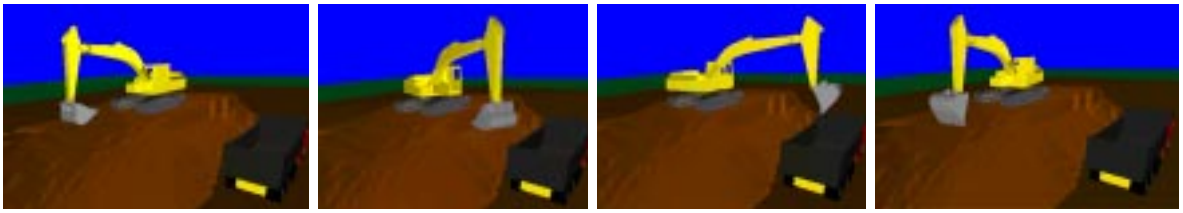


Figure 3.8: The Example Task motion.

To begin building the parameterized script for the Example Task, the states of the script must be created. From the description of the Example Task, the states of the parameterized script are easily obtained for each joint. Table 3.1 shows the script states for the swing joint, and Table 3.2 shows the script states for the bucket joint. Brief descriptions of the commands, that are associated with each script state are shown as well.

State	Motion	Command
0	Swing to truck	Desired swing angle at truck
1	Swing to dig face	Desired swing angle at dig face

Table 3.1: Example Task Swing Script States

State	Motion	Command
0	No motion	Current bucket angle
1	Open bucket	Desired bucket angle to open to

Table 3.2: Example Task Bucket Script States

3.3.3 Script Rules

The rules of the parameterized script specify when to transition from one script state to the next. These state transitions are based on events that occur during the motion of the vehicle. The events are when certain joints reach their desired goal positions, or when a joint passes an intermediate angular position during its motion. Script rules contain comparison functions between the vehicle state and the script parameters. When the comparison function of the script rule that is currently active evaluates to true, the script transitions from the current script state to the next script state.

The script proceeds in the forward direction only. Once a script step transition takes place, the script is not allowed to return to a previous step. This prevents problems with becoming stuck in loops or oscillations in the excavator's commanded position, which could result in vehicle chatter.

The transitions between script states are based on events and not on time. If they were based on time, that would require additional information about the velocities and accelerations of the different joints to compute how long each separate joint motion would take in order to achieve the proper motion coordination. This dynamic information would be hard to obtain without a model of the coupled joint actuation which, as stated earlier, is difficult to construct. With an event-based approach, this dynamic information is not needed, and the proper sequence of motions can be guaranteed regardless of the length of time it takes to execute them.

Because of the event-based nature of the script state transitions, the script rules define inherent dependencies between the different joint motions. For the Example Task, the motion dependencies are quite simple since there are only two joints involved. At the start of the task, the swing is the only joint that is commanded to move to a new location. This means that the second part of the task, opening the bucket, will be dependent on the swing motion to the truck. The comparison function for the script rule that transitions from Bucket State 0 to Bucket State 1 is the swing joint passing a certain angle on its way to the truck.

The final machine motion of the task, swinging back to the dig face, can either depend on the bucket-open motion, or the initial swing-to-truck motion. It is decided that this swing motion is dependent on the bucket-open motion, since the original task description requires the bucket to deposit its material before returning to the dig face. Making the final swing motion be dependent on the initial swing-to-truck motion may not guarantee that the bucket will deposit all of its material before swinging back. The comparison function for the script rule which transitions from Swing State 0 to Swing State 1 is the bucket opening beyond a certain angle.

A final type of script rule is the script termination condition. This is also a function of the vehicle state, but may or may not require any variable script parameters. One possible script termination condition is one or more joints reaching their final commanded goals. For the Example Task, the script termination condition is the swing coming within 1° of its goal at the dig face.

Table 3.3 and Table 3.4 summarize the script rules for the Example Task's parameterized scripts. A way to diagram the motion dependencies in the script rules is presented in Section 3.3.5.

State Transition	Dependent on	Event
0 -> 1	Bucket	Bucket passes certain angle
1 -> END	Swing	Swing comes within 1 degree of dig face goal

Table 3.3: Example Task Swing Script Rules

State Transition	Dependent on	Event
0 -> 1	Swing	Swing passes certain angle

Table 3.4: Example Task Bucket Script Rules

3.3.4 Script Parameters

Script parameters define the specific kinematic details of motion, such as the precise location to move a certain joint, or the exact angular value in a script transition rule. Script parameters are computed for each different task trial and are dependent on the information that the motion planner receives about its environment, such as the location of the truck, the desired dump location, the next desired dig location, and current soil conditions. There are two types are script parameters: *command parameters* and *action parameters*.

Command Parameters: As the name suggests, command parameters specify the desired angular position (step) or velocity (ramp) command values that are associated with each script state. When a script state is active during task execution, the command parameter value is sent to the

joint controller.

Action Parameters: Action parameters specify the values of the events in the comparison functions of the script rules. When a script rule is evaluated, the vehicle state is compared to the appropriate action parameter value, and if the comparison is true, a script state transition takes place.

Since there are four total script states and two simple script rules for the Example Task, there are four command parameters and two action parameters[1]. Table 3.5 summarizes the entire script parameter set for the Example Task's parameterized script. Along with a description of each parameter is also the symbol that is used to refer to that parameter for the remainder of the document. The nomenclature for the command parameter symbols is a brief semantic description of the portion of the task it is associated with followed by an abbreviation of the joint that is commanded to move. For the action parameters, there are two joint abbreviations. The first symbol refers to the joint whose motion the second symbol is dependent on. For example, since the swing-to-truck motion triggers the bucket-open motion, the action parameter is symbolized by $a_{sw\text{-}bk}$.

symbol	parameter description
$dump\theta_{sw}$	desired swing angle at truck
$dig\theta_{sw}$	desired swing angle at dig face
$init\theta_{bk}$	bucket starting angle
$open\theta_{bk}$	angle to open bucket to
$a_{sw\text{-}bk}$	swing angle that causes bucket joint to open
$a_{bk\text{-}sw}$	bucket angle that causes swing joint to return to dig face

Table 3.5: Example Task Script Parameters

3.3.5 Script Dependency Graph

A way to visualize the motion dependencies in the script rules is in the form of a *script dependency graph* (SDG). The script dependency graph combines the individual joint scripts and shows the flow of motion for the entire task. The SDG shows which task motions are affected by which action parameters. This helps in isolating parts of the task and, as is shown in Chapter 4, provides a way to reduce the dimensionality of the overall action parameter space to smaller dimensional spaces.

1. A script rule can contain more than one comparison function, which would require more than one action parameter.

A circle in the script dependency graph is a script state. Inside each circle is the script state number, a brief description of the state, and the associated command parameter symbol. Links between the states in the graph are action parameters. States are linked together in the graph by their motion dependencies, which come directly from the script rules. The SDG for the Example Task is shown in Figure 3.9.

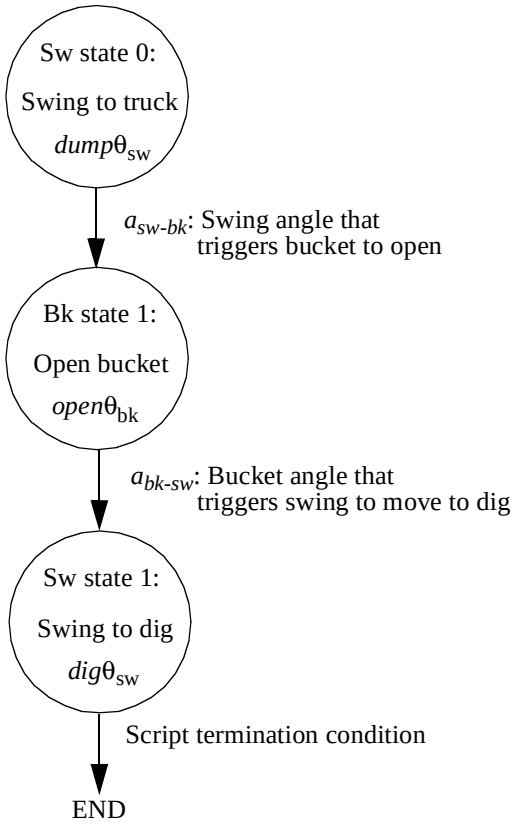


Figure 3.9: Script Dependency Graph for the Example Task.

The script dependency graph shows which motions affect which other motions in the script. Unlike a finite state machine, where only one node can be active at a time, both parent and child states of the script dependency graph can be active at the same time. For example, consider the first two nodes of the Example Task script dependency graph in Figure 3.9. The swing could still be moving to the goal at the truck, specified by $dump\theta_{sw}$, when the bucket begins to open, so both states would be active simultaneously during script execution. The swing would then change to Swing State 1, shown in the third node of the graph, once the bucket has passed the action parameter value specified by $a_{bk\text{-}sw}$.

3.3.6 Script Execution

Figure 3.10 shows the flow of information during execution of the parameterized script. This is a

small piece of the total system diagram shown in Figure 3.7. Before motion begins, the script parameter values are filled in to the parameterized script's states and rules. The output of the script are the commands that are sent to the vehicle's joint controller. The vehicle state, which is needed for the script rule comparisons, is fed back to the script. The script rules are evaluated with the new vehicle state, and state transitions occur when necessary. The rules are updated at the rate that vehicle state information can be obtained. For both the Example Task and the actual truck loading system, this rate is 10 Hz due to the available hardware.

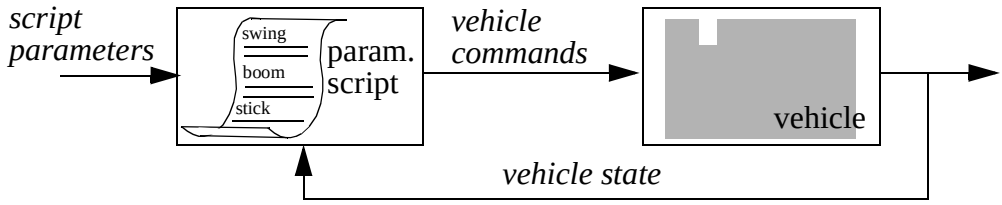


Figure 3.10: Block diagram showing the flow of information when the parameterized script is used to command the vehicle.

Let us now examine the vehicle motion that results from the Example Task parameterized script. For this discussion, the values of the four command parameters are simply given and remain constant. In a more general case, the command parameter values would be computed by the command parameter computation function and would most likely change for every loading pass. The four command parameter values for this example are:

$$
\begin{aligned}
dump\theta_{sw} &= 90^{\circ} \\
dig\theta_{sw} &= -20^{\circ} \\
init\theta_{bk} &= -70^{\circ} \\
open\theta_{bk} &= 40^{\circ}
\end{aligned}
$$

A fifth piece of information that is needed is the start angle of the swing joint. Although it is not a script parameter as defined, it is still a necessary quantity to complete this example. It will be 20°.

Although parameterized scripts do sequence and constrain the vehicle motions for a given task, they can be quite flexible. The flexibility comes in the choice of the values of the script parameters. Different script parameter values can produce very different vehicle motions with the same script. To see how different script parameter values affect the vehicle's motion, the values of the two action parameters are changed for this example.

The values of the command parameters place a range on the legal values of the action parameters. For example, recall that transitioning to the bucket-open script state is dependent on the swing joint passing a certain angle during its motion to the truck. If the swing moves from 20° to 90°,

but the bucket is commanded to open when the swing passes $120°$, then the bucket will never open because the swing joint will never reach $120°$.

For this example, the ranges of the two action parameters for the given command parameter values are:

$$a_{sw\text{-}bk} = [20°, 90°]$$
$$a_{bk\text{-}sw} = [-70°, 40°]$$

The action parameter ranges change as the values of the command parameters change for each different trial of the task.

Low Range	Mid Range	High Range
$a_{sw\text{-}bk} = 20°$	$a_{sw\text{-}bk} = 55°$	$a_{sw\text{-}bk} = 85°$
$a_{bk\text{-}sw} = -70°$	$a_{bk\text{-}sw} = -15°$	$a_{bk\text{-}sw} = 30°$

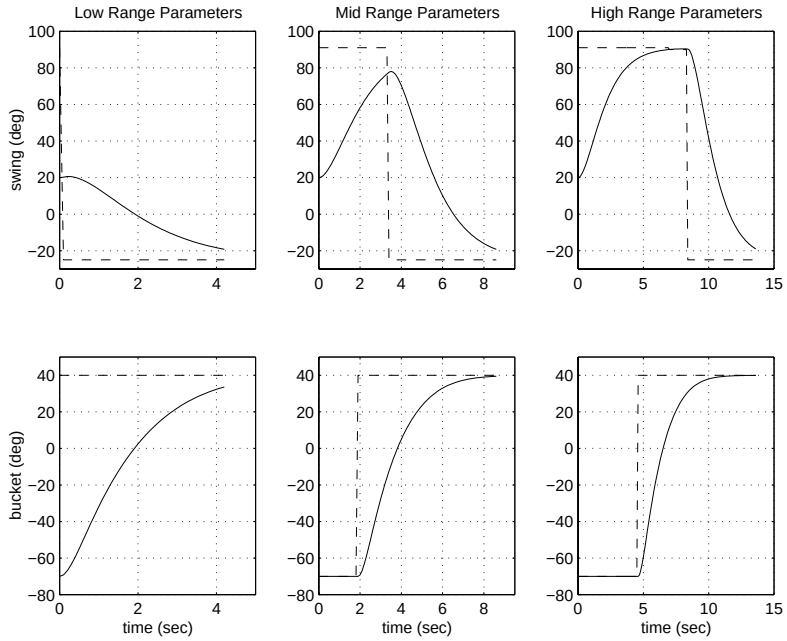


Figure 3.11: Vehicle motions produced by three different sets of action parameter values.

Figure 3.11 shows the motions for the Example Task script that are produced by three different sets of action parameter values. The plots show the angular positions of the swing and bucket

joints versus time. These types of plots are referred to as joint traces. The solid lines are the joint positions and the dashed lines are the joint commands.

The swing angle joint traces are in the top row and the bucket angle joint traces are in the bottom row of the set of plots. Each column represents a different set of action parameter values. The first column uses action parameter values from the low end of the action parameter ranges, the second column uses values in the middle of the ranges, and the third column shows the joint motions resulting from parameter values from the high end of the action parameter value ranges.

For the three motions, the bucket joint traces are always the same shape, except for the amount of time that the bucket waits before opening. The swing joint traces are very different, though. For the low range action parameters, the command to swing to the truck is sent out for only 0.1 seconds before the swing is commanded to move back to the dig face. The mid-range action parameter trace shows the swing almost reaching its goal at the truck but is commanded to swing back before reaching it. The third swing joint trace shows the swing reaching and dwelling at its goal at the truck before swinging back to the dig face.

The idea of parameterizing the excavator's motion by a small set of numbers is a powerful one in terms of applying an adaptive approach to improve task performance. Clearly, the third set of joint traces in Figure 3.11 comes closest to completing the desired task as stated, but at the expense of a long task execution time. It may be possible to perform the same task in a shorter time. If there were a way of capturing the sense of task accomplishment, as a numerical score assigned to each action parameter set for instance, then that information could be used to find the best set of action parameter values in terms of both fast execution times and acceptable task behavior. The remainder of the technical chapters describe the extension of the parameterized script motion planning algorithm to an adaptive one that can find the best set of action parameter values based on its own past experiences.

For this research, we will only consider varying the values of the action parameters, although the command parameters can be adjusted as well. The section on future work in Chapter 10 discusses ways to automatically adjust the command parameter values.

3.4 Truck Loading Parameterized Script

This section presents the parameterized script that was constructed for the task of loading trucks with an excavator. The following sections will describe the different components of the truck loading parameterized script.

3.4.1 Script States

As in the Example Task from Section 3.3, the creation of the script's states come directly from a description of the task. The different motions for the free motion portion of the truck loading task are shown in Figure 3.12.



Digging is complete.



Raise boom and curl bucket to capture the soil.



Swing to the truck.



Move stick to dump position 1.



Open bucket.



Move stick to dump position 2.



Swing to dig face.



Move stick to desired dig angle.



Move bucket to desired dig angle.



Lower boom.



Resume digging.

Figure 3.12: The sequence of vehicle motions for the truck loading task free motion.

This list of vehicle motions is not meant to imply that the previous motion must finish before the next motion can begin. In fact, several vehicle motions could occur simultaneously. For example, the bucket could begin to open as it swings to the truck. Exactly when to begin opening the bucket such that the soil falls in the truck bed is another matter, however, and is specified by the action parameters that determine the joint coordination.

Notice that there is a two-part dumping maneuver involving the stick link. The stick is first extended out over the truck bed and then brought in towards the excavator as the bucket opens. The truck bed is too narrow to catch the soil if the bucket were to simply open in place, which creates a wide swath. Instead, the stick is brought in as the bucket opens in an attempt to keep the mouth of the bucket centered over the truck bed.

Also notice that there is no boom-lower motion once the bucket is over the truck bed. This motion was eliminated for safety reasons. It was found on the excavator testbed that this boom motion is very fast and difficult to control, since gravity and not the hydraulic cylinder is doing the work. Also, the implements are out over the side of the excavator and not the front, which is a less stable machine configuration. Lowering the boom with a full bucket of soil and then stopping quickly has the potential to tip over the excavator, so this motion was not included in the script states. The disadvantage is the stick link also affects the elevation of the bucket over the truck bed. The change in height between the stick when vertical and the stick when fully extended is approximately 4 feet (1.1 meters). If the desired dump location requires extending the stick, then the soil would be dropped from a high elevation into the truck bed, which could damage the truck. One possible remedy to this problem could be to slow down the rate at which the bucket opens to allow the soil to fall out more gradually or to include a very slow boom lower motion over the truck. Both solutions would result in slower loading times.

The truck loading task motion can now be organized by the different joints to produce each joint's individual script states. Tables 3.6 through 3.9 list a description of the truck loading script states and the associated command.

State	Motion	Command
0	No motion	Current swing angle
1	Swing to truck	Swing angle at dump location
2	Swing to dig face	Swing angle at dig location

Table 3.6: Truck Loading Swing Script States

State	Motion	Command
0	Raise boom	Boom angle that allows for truck clearance
1	Lower boom	Safe boom down velocity (ramp command)
2	No motion	Boom angle at dig location

Table 3.7: Truck Loading Boom Script States

State	Motion	Command
0	No motion	Current stick angle
1	Move to first dump location	Stick angle at first dump location
2	Move to second dump location	Stick angle at second dump location
3	Move to dig	Stick angle at dig location

Table 3.8: Truck Loading Stick Script States

State	Motion	Command
0	Capture soil	Bucket angle that keeps soil in bucket
1	Open bucket	Maximum bucket angle
2	Move to dig	Bucket angle at dig location

Table 3.9: Truck Loading Bucket Script States

3.4.2 Script Rules

The rules for the truck loading script come directly from the description of the sequence of motions that are required for the task. Many of the motion dependencies arise because only one joint is moving during part of the loading cycle. For others, there was more than one possibility so the choice that best preserved the task motion sequence was chosen. Many of the motions are dependent on the swing joint, since it is moving during most of the loading cycle. Tables 3.10 to 3.13 list the truck loading script motion dependencies along with a brief description of the transition events.

State Transition	Dependent on	Event
0 -> 1	Boom	Boom raises past certain angle
1 -> 2	Bucket	Bucket opens past certain angle
2 -> END	Swing	Swing enters margin around desired dig angle goal

Table 3.10: Truck Loading Swing Script Rules

State Transition	Dependent on	Event
0 -> 1	Swing	Swing moves past truck when swinging to dig face
1 -> 2	Boom	Boom command passes boom dig angle

Table 3.11: Truck Loading Boom Script Rules

State Transition	Dependent on	Event
0 -> 1	Swing	Swing passes certain angle when swinging to truck
1 -> 2	Bucket	Bucket opens past certain angle
2 -> 3	Swing	Swing passes certain angle when swinging to dig face
3 -> END	Stick	Stick enters margin around desired dig angle

Table 3.12: Truck Loading Stick Script Rules

State Transition	Dependent on	Event
0 -> 1	Swing & Stick	Swing passes certain angle when swinging to truck & stick passes certain angle when moving to first dump angle
1 -> 2	Swing	Swing passes certain angle when swinging to dig face

Table 3.13: Truck Loading Bucket Script Rules

Notice that some script state transitions are not only dependent on when a certain joint has moved past a certain value. They are also dependent on the script state as well. Consider the transition from Boom State 0 to Boom State 1. The rule states that the swing must be in the clear zone between the truck and the dig face for the boom to begin to lower in order to avoid a collision with the truck. However, at the beginning of the loading cycle, the swing is also in the same clear zone

as it swings to the truck, so the boom could begin to lower then as well. In order to resolve this ambiguity, the dependency on the swing script state is needed. Only when the swing is moving to the dig location does that transition rule become valid.

3.4.3 Script Parameters

The script parameters for the truck loading motion planner are computed from information it receives about its work environment. This information is:

- Vehicle state immediately after digging finishes.
- Angular values of the bucket that specify the bucket angle at which the material will remain captured in the bucket, and the bucket angle at which the material will fall out of the bucket. These are based on soil conditions and can be measured ahead of time.
- The location of the four truck bed corners expressed in Cartesian coordinates in a common reference frame which is relative to the excavator. This information could come from an external source, such as a software module that determines the truck location.
- The desired dump location in the truck also specified as a Cartesian point in the same common reference frame. This information can come from an external source such as a software module that determines where the next bucket of soil should be placed in the truck.
- The next desired dig location, which is provided by an external source such as a software module that plans where to dig the next bucket of soil based on the shape of the terrain. A Cartesian point is not enough information, as the angle of the bucket teeth relative to the terrain is also required. In practice, the dig location is expressed as the four desired excavator joint angles.

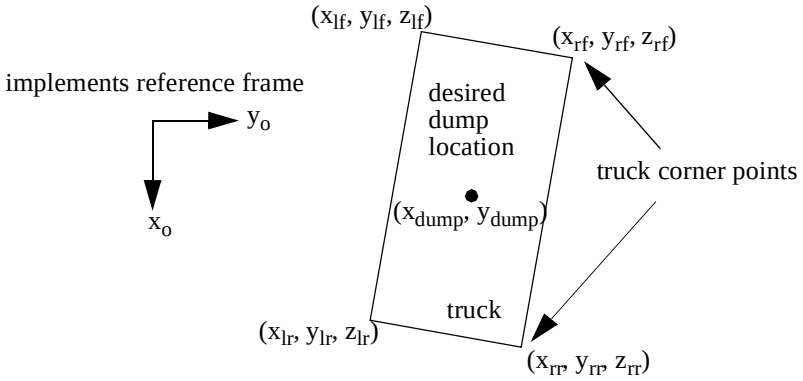


Figure 3.13: The four truck bed corner points and the desired dump coordinate.

The form of the truck location and dump point location information is shown in Figure 3.13.

Chapter 9 contains a more complete description of the autonomous truck loading system, which describes the different external planner components in more detail.

3.4.3.1 Command Parameters

Referring to Table 3.6 through Table 3.9, there are thirteen total script states. Since each script state defines one machine motion for the total task, this means that there are thirteen command parameters.

Initial Configuration: Swing State 0, Stick State 0

Two of the thirteen command parameters are simply the initial state of the excavator. *Swing State 0* and *Stick State 0* require the excavator to remain at the current swing and stick angles when the digging process finishes. These are symbolized as $init\theta_{sw}$ and $init\theta_{st}$ respectively.

Dig Location: Swing State 2, Boom State 2, Stick State 3, Bucket State 2

Four of the thirteen command parameters are the four joint angles at the dig location as provided by the dig location planner. These are entered as is into the script. They are symbolized as $dig\theta_{sw}$, $dig\theta_{bm}$, $dig\theta_{st}$, and $dig\theta_{bk}$ respectively.

Bucket Open Angle: Bucket State 1

The command parameter for *Bucket State 1* is the desired angle to open the bucket to. This is the maximum kinematic limit on the bucket angle and is symbolized as $open\theta_{bk}$.

Boom Down Ramp: Boom State 1

Unlike the other command parameters, the command parameter for *Boom State 1* is not a desired position but a desired velocity. It was found that a step command resulted in very fast boom motion with a sudden stop at the end. Experiments were conducted to determine a slower, less jarring, boom speed which could be controlled with a ramp command. This command parameter is symbolized by $down\dot{\theta}_{bm}$.

The five remaining command parameters use the excavator's inverse kinematics to transform information about the truck and desired dump location to the proper form for the parameterized script. The kinematic equations are presented in Appendix A. To illustrate the different command parameters, a simplified kinematic model of the excavator shown in Figure 3.14 is used.

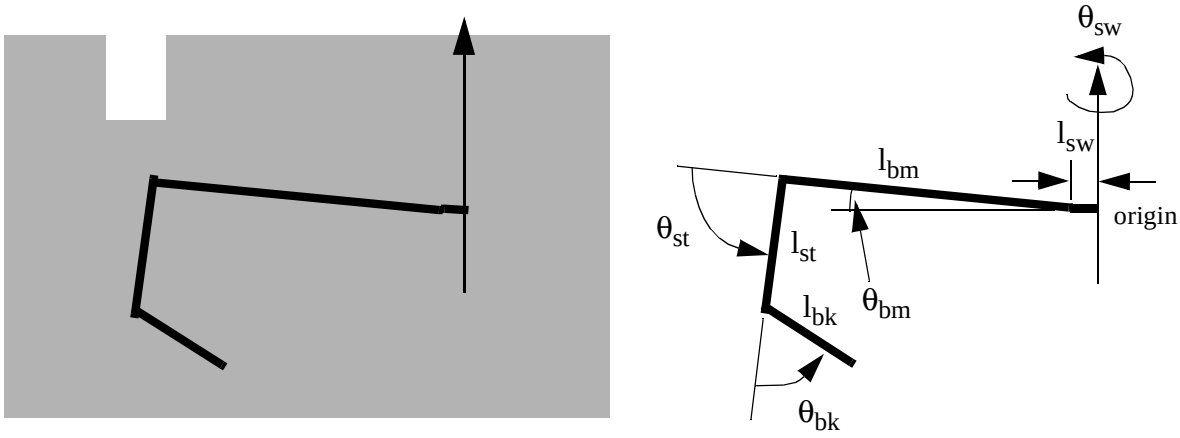


Figure 3.14: Schematic of simplified excavator kinematic model.

Boom Clearance Angle: Boom State 0

This command parameter is the angle of the boom joint that allows the bucket to safely clear the truck. It is a function of the truck height, truck location, excavator link lengths, and safety margin between the bucket and the truck. This command parameter is symbolized by $dump\theta_{bm}$. Figure 3.15 shows a schematic illustrating the boom clearance angle.

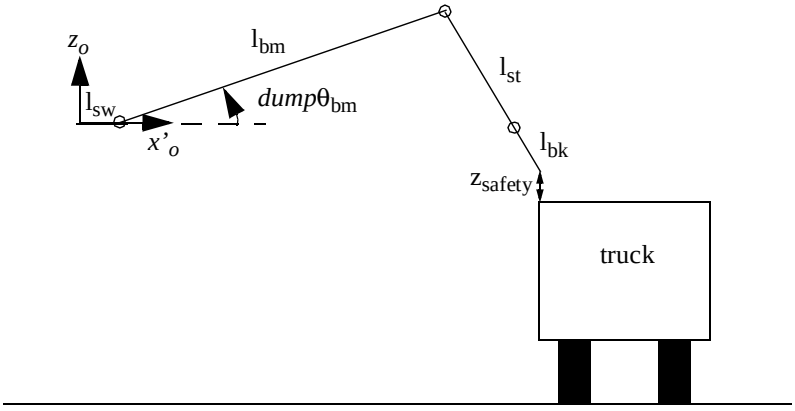


Figure 3.15: Schematic illustrating the boom clearance angle.

Swing Dump Angle: Swing State 1

Figure 3.16 shows the swing dump angle that is found using the desired dump point coordinates. Minimum and maximum limits on the swing dump angle are also found using information about the truck. This command parameter is symbolized as $dump\theta_{sw}$.

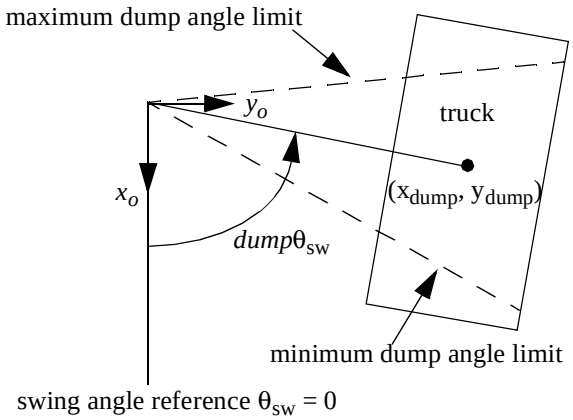


Figure 3.16: Schematic showing the swing dump angle command parameter.

Stick Dump Angles: Stick States 1 and 2

The two angles that define the stick goals for the dumping motion are computed using information about the truck's location and the desired dump point. These command parameters are symbolized as $dump\theta_{st1}$ and $dump\theta_{st2}$ and are shown in Figure 3.17.

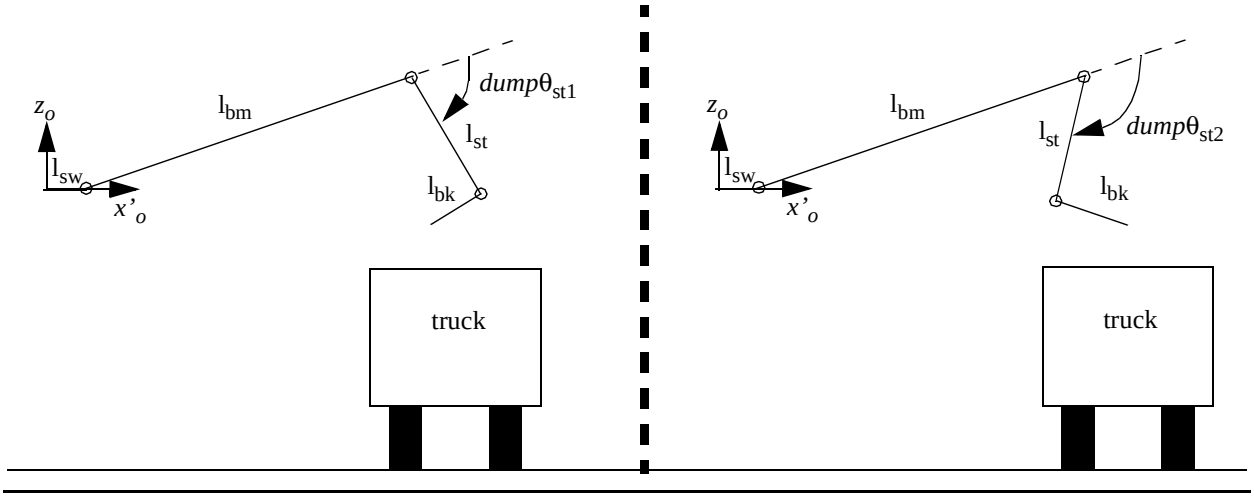


Figure 3.17: Schematic showing the two stick dump command parameters.

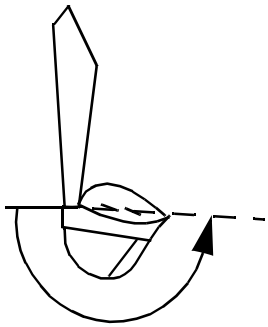


Figure 3.18: Diagram showing the bucket capture angle command parameter.

Bucket Capture: Bucket State 0

This command parameter, symbolized $capture\theta_{bk}$ is the angle of the bucket that ensures the soil remains captured in the bucket after digging and is shown in Figure 3.18. This parameter is a function of the soil conditions that are provided as input to the command parameter computation as well as the $dump\theta_{bm}$ and $dump\theta_{st1}$ command parameters. Extending the stick link is equivalent to opening the bucket, since both move in the same angular direction. Therefore, the bucket may have to be curled slightly beyond horizontal at first so when the stick does extend over the truck, the soil remains captured. This is taken into account when computing the $capture\theta_{bk}$ script parameter.

3.4.3.2 Action Parameters

Referring to Table 3.10 through Table 3.13, there are nine script state transitions, not counting the ones that transition to the script termination state. One script state transition, from *Boom State 1* to *Boom State 2*, is dependent on the $dig\theta_{bm}$ command parameter. Another script state transition, from *Bucket State 0* to *Bucket State 1*, involves the motions of two joints, so there are two action parameters in the same rule, leaving nine total action parameters. The action parameters and their ranges are:

- $a_{bm\text{-}sw}$: boom angle that triggers the swing joint to begin moving to the truck.
  Range: [$init\theta_{bm}$ - $dump\theta_{bm}$]

- $a_{sw\text{-}st_dump}$: swing angle that triggers stick joint to move to first dump angle.
  Range: [$init\theta_{sw}$ - $dump\theta_{sw}$]

- $a_{sw\text{-}bk}$: swing angle that triggers bucket joint to open.
  Range: [$init\theta_{sw}$ - $dump\theta_{sw}$]

- $a_{st\text{-}bk}$: stick angle that triggers bucket joint to open.
  Range: [$init\theta_{st}$ - $dump\theta_{st1}$]

- $a_{bk\text{-}st}$: bucket angle that triggers stick to move from first dump angle to second dump angle.
  Range: [$init\theta_{bk}$ - $open\theta_{bk}$]

- $a_{bk\text{-}sw}$: bucket angle that triggers swing joint to return to dig face.
  Range: [$init\theta_{bk}$ - $open\theta_{bk}$]

- $a_{sw\text{-}bm}$: swing angle that triggers boom joint to lower to dig location.
  Range: [$dump\theta_{sw}$ - $dig\theta_{sw}$]

- $a_{sw\text{-}st_dig}$: swing angle that triggers stick joint to move to dig location.
  Range: [$dump\theta_{sw}$ - $dig\theta_{sw}$]

- $a_{sw\text{-}bk}$: swing angle that triggers bucket joint to move to dig location.
  Range: [$dump\theta_{sw}$ - $dig\theta_{sw}$]

The following table summarizes the entire script parameter set for the truck loading parameterized script.

symbol	parameter description
$init\theta_{sw}$	swing angle after digging
$init\theta_{st}$	stick angle after digging
$capture\theta_{bk}$	bucket angle which captures soil
$dump\theta_{sw}$	desired swing angle at truck
$dump\theta_{bm}$	boom angle which guarantees truck clearance
$dump\theta_{st1}$	stick angle for first half of dump motion
$dump\theta_{st2}$	stick angle for second half of dump motion
$dump\theta_{bk}$	bucket angle to open to
$down\dot{\theta}_{bm}$	boom down desired velocity
$dig\theta_{sw}$	desired swing angle at dig point

Table 3.14: Truck Loading Script Parameters

symbol	parameter description
$dig\theta_{bm}$	desired boom angle at dig point
$dig\theta_{st}$	desired stick angle at dig point
$dig\theta_{bk}$	desired bucket angle at dig point
$a_{bm\text{-}sw}$	boom angle that causes swing joint to move to truck
$a_{sw\text{-}st_dump}$	swing angle that causes stick joint to move to first dump angle
$a_{sw\text{-}bk}$	swing angle that causes bucket joint to open
$a_{st\text{-}bk}$	stick angle that causes bucket joint to open
$a_{bk\text{-}st}$	bucket angle that causes stick joint to move to second dump angle
$a_{bk\text{-}sw}$	bucket angle that causes swing joint to move to dig face
$a_{sw\text{-}bm}$	swing angle that causes boom joint to lower to dig
$a_{sw\text{-}st_dig}$	swing angle that causes stick joint to move to dig angle
$a_{sw\text{-}bk}$	swing angle that causes bucket joint to move to dig angle

Table 3.14: Truck Loading Script Parameters

3.4.4 Script Dependency Graph

The script dependency graph for the truck loading script is shown in Figure 3.19. The script dependency graph shows the motion dependencies between different script states and the action parameters that link them together.

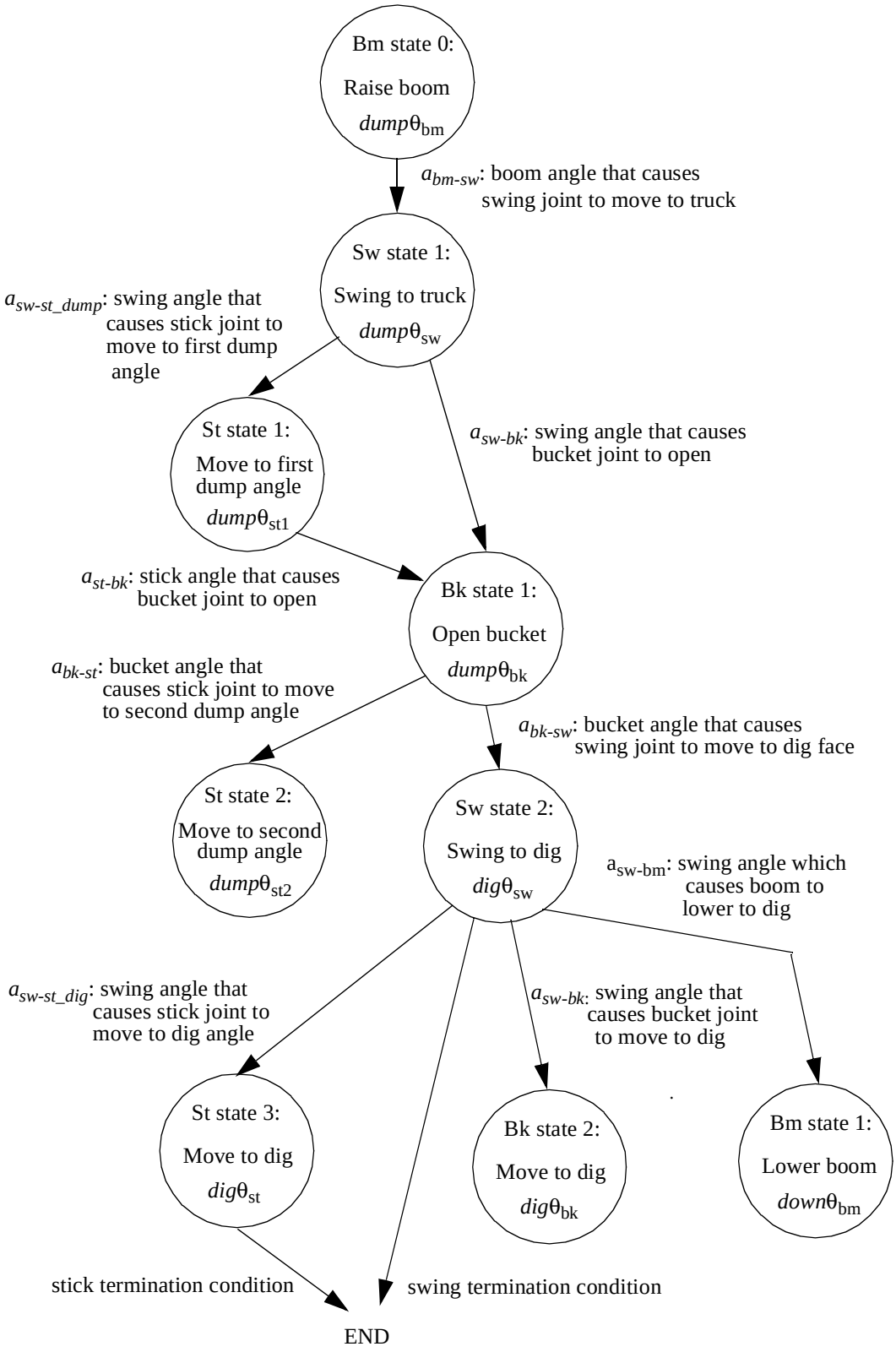


Figure 3.19: Truck loading task Script Dependency Graph

3.4.5 Script Execution

Figures 3.20 and 3.21 show the joint motions which are produced by running the truck loading script on the excavator testbed. The values of all the parameters are shown in the tables below each plot.

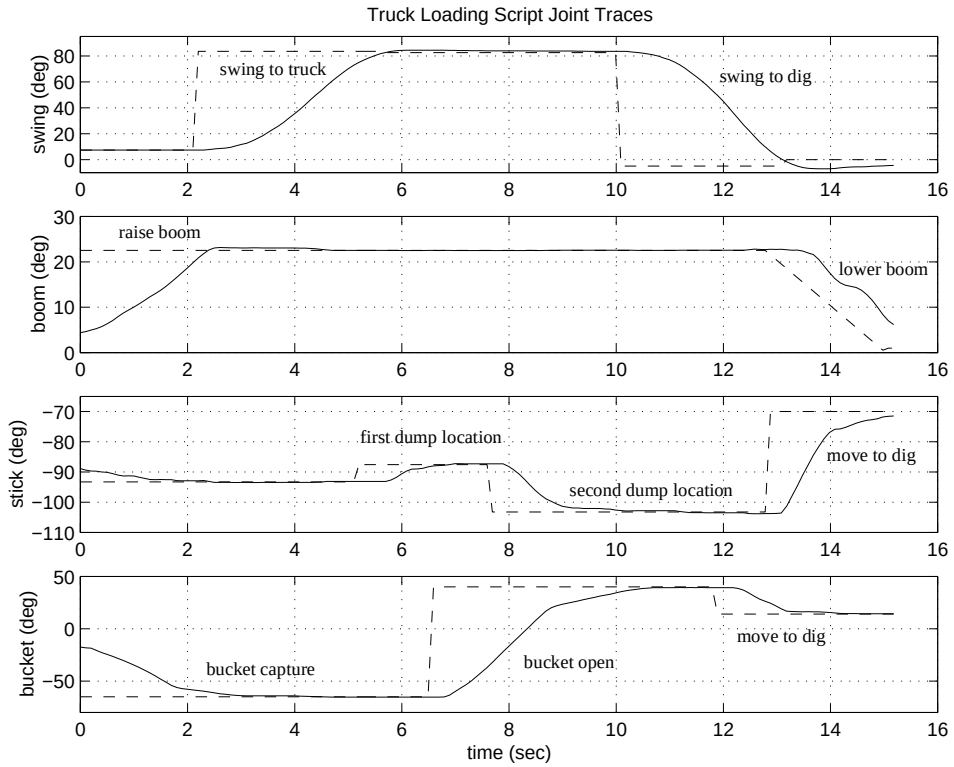


Figure 3.20: Joint traces showing the free motion of the excavator for one loading pass.

symbol	value (deg)	symbol	value (deg)	symbol	value (deg)	symbol	value (deg)
$init\theta_{sw}$	7.37°	$dump\theta_{st2}$	-103.23°	$dig\theta_{bk}$	14.10°	$a_{bk\text{-}sw}$	35.00°
$init\theta_{st}$	-93.29°	$dump\theta_{bk}$	40.00°	$a_{bm\text{-}sw}$	20.71°	$a_{sw\text{-}bm}$	8.26°
$capture\theta_{bk}$	-64.98°	$down\dot{\theta}_{bm}$	-10.00°/s	$a_{sw\text{-}st_dump}$	75.04°	$a_{sw\text{-}st_dig}$	8.26°
$dump\theta_{sw}$	82.56°	$dig\theta_{sw}$	0.00°	$a_{sw\text{-}bk}$	75.04°	$a_{sw\text{-}bk}$	51.72°
$dump\theta_{bm}$	22.53°	$dig\theta_{bm}$	1.00°	$a_{st\text{-}bk}$	-88.13°		
$dump\theta_{s1t}$	-87.55°	$dig\theta_{s}$	-70.00°	$a_{bk\text{-}st}$	-34.98°		

Table 3.15: Truck Loading Script Parameter Values

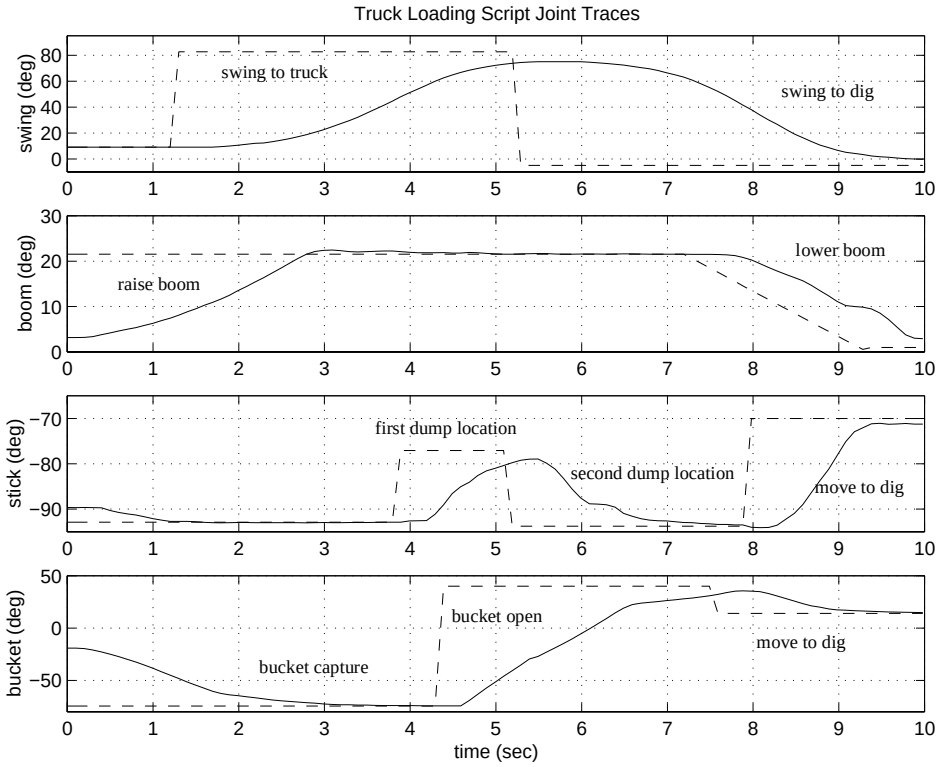


Figure 3.21: Joint traces for a second loading pass with different script parameter values. Notice the change in times between the joint traces of Figure 3.20 and these joint traces.

symbol	value (deg)	symbol	value (deg)	symbol	value (deg)	symbol	value (deg)
$init\theta_{sw}$	9.16°	$dump\theta_{st2}$	-93.75°	$dig\theta_{bk}$	14.10°	$a_{bk\text{-}sw}$	-36.84°
$init\theta_{st}$	-92.87°	$dump\theta_{bk}$	40.00°	$a_{bm\text{-}sw}$	7.60°	$a_{sw\text{-}bm}$	61.61°
$capture\theta_{bk}$	-74.48°	$down\dot{\theta}_{bm}$	-10.00°/s	$a_{sw\text{-}st_dump}$	45.46°	$a_{sw\text{-}st_dig}$	39.20°
$dump\theta_{sw}$	81.61°	$dig\theta_{sw}$	0.00°	$a_{sw\text{-}bk}$	49.24°	$a_{sw\text{-}bk}$	52.55°
$dump\theta_{bm}$	21.55°	$dig\theta_{bm}$	1.00°	$a_{st\text{-}bk}$	-90.28°		
$dump\theta_{st1}$	-77.07°	$dig\theta_{s}$	-70.00°	$a_{bk\text{-}st}$	-44.48°		

Table 3.16: Truck Loading Script Parameter Values

Notice the differences in the two joint traces, particularly in the task execution times. The first loading pass shown in Figure 3.20 takes almost 6 seconds longer than the second one shown in Figure 3.21. The command parameter values for the two truck loading trials are nearly identical.

The difference in the speed of the two trials results from different values of the action parameters, which affect the script state transitions and the coordination of the joint motions.

3.4.6 Implementational Details

This section describes a few of the problems and the solutions when implementing and testing the truck loading parameterized script on the actual hydraulic excavator testbed. For one, the swing joint controller performs well for the first 90%-95% of the swing's motion toward a goal point, but for the last 5%-10% of the motion the swing joint moves very slowly to the goal. To overcome this problem, a slight overshoot was added to each swing goal. When the swing passed the desired goal, the swing command was reset to the desired goal. This greatly improved swing performance while adding minimal overshoot. For the adaptive parameter computation described in the remainder of this document, this commanded overshoot is eliminated.

Another problem occurred with the initial stick command. The stick is commanded to stay at its current position, however, when the digging algorithm finishes, the stick joint is still moving with a substantial velocity. Setting its goal position to its current state when the digging algorithm terminates resulted in a violent jerk of the stick joint, which shook most of the heaped soil out of the bucket. To prevent this from occurring, the stick command for this script step was modified to ramp down the stick velocity in a more gradual way and prevent the violent shaking.

3.5 Discussion

The parameterized scripting motion planning algorithm brought about a remarkable change in the performance of the autonomous excavator for the truck loading task. Initial loading pass times using the cubic spline approach were between 30 and 45 seconds. The parameterized script resulted in loading passes (including digging) around 15 to 20 seconds. Because the sequence of joint motions was made explicit, problems with power limitations and actuator coupling were eliminated. With properly tuned parameter values, especially the action parameters, the motion of the autonomous excavator testbed approached that of a skilled human excavator operator in the same working conditions. Parameterized scripting is a simple idea, yet it is the key to meeting the desired performance specifications of the autonomous truck loading system.

Parameterized scripting reduces the number of vehicle command options to a small set of manageable parameters. Many robot manipulation systems require a vector of command inputs, which can be very high dimensional, and is usually expressed as a closed-form function of time. A high dimensional command vector makes the problem of modifying the commands to achieve optimal behavior very difficult. Parameterized scripts, on the other hand, have a relatively low-dimensional and more tractable search space.

The disadvantage of parameterized scripts, however, is that an artificial structure or limitation is set on the vehicle's motion because of the reduced number of command options. This can be problematic if seeking the truly globally optimal behavior. Techniques that optimize in the space of the script parameters will find the optimal set of values, but only within the context and confines of

the script states and script rules.

Parameterized scripts are intended for robots working in predictable environments with tasks that are highly repetitive, whose general sequence of motions are fixed, but whose kinematic or geometric details can change between individual task trials. This is a higher level of functionality than teach-playback manipulator systems where the robot constantly executes exactly the same path. Parameterized scripts are best used for robot tasks where the robot's motion can be expressed in the form of a set of simple states and rules, or where following a specified trajectory is not required. Parameterized scripting is not for every robot application. Welding robots, for example, whose goal is to follow a pre-specified trajectory would not be well suited for parameterized scripts, nor would a more general robot problem such as navigating out of a cluttered room.

Quite a bit of human engineering and knowledge is required in creating a script and specifying its parameters. This assumes that the human is the best at knowing how to do a particular job and how to express it in the form of a script. It remains to be seen, however, if the script is correct or complete. Perhaps future extensions of the parameterized scripting approach could seek proofs that guarantee that the script will terminate for all parameter values, or that the excavator's bucket will cover the desired amount of the workspace. Other future extensions of the parameterized script approach could explore means to generate scripts more automatically, perhaps by having a human demonstrate the task motion that is then captured in the form of a parameterized script.

The next chapter presents two key definitions that are used throughout the remainder of the document, *task states* and *actions*. These definitions are required for posing the problem of adaptively computing the script parameters as one of machine learning.

Chapter 4 Task States and Actions

This chapter presents the concepts of *task states* and *actions*, which are used in the adaptive motion planning system. Task states are so named to avoid confusion with previously mentioned vehicle states and script states. The task state is the input to the action parameter computation component of the system. An action is computed for the given task state that the system believes will maximize the reward that the system receives.

Figure 4.1 shows how the task states and actions fit into the adaptive motion planning system as it has been constructed so far. The thicker lines and bold text show the new additions to the system diagram.

Section 4.1 and Section 4.2 describe task states and actions in more detail. Section 4.3 discusses how complicated tasks consisting of many actions can be decomposed into separate subtasks, which reduces the action space dimensionality. Section 4.4 describes the task states and actions for the truck loading task. Some implementational details are given in Section 4.5. Finally, Section 4.6 presents a discussion concerning task states and actions.

4.1 Task States

In the general sense, a *state* is the condition of the world that the learning system uses to decide what to do. In a slightly different sense, a *task state* can be thought of as a set of specifications for one instance of a given task. For example, for each loading pass to the truck and back, which we will refer to as a *task execution cycle*, the adaptive motion planning system is presented with information about the current vehicle state, environmental information such as the location of the truck, and the goals for the current loading pass, such as where to dump the soil. The adaptive

motion planning algorithm must then select a set of actions that it believes will meet the given specifications and result in the best task performance.

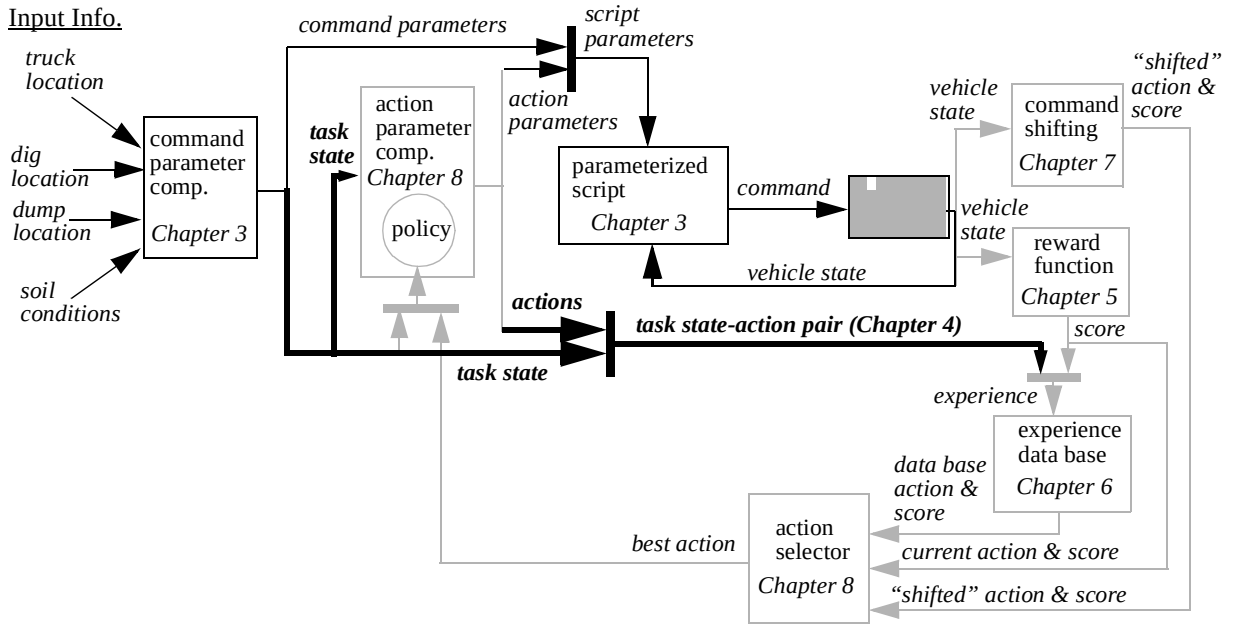


Figure 4.1: Block diagram of the adaptive learning system. In bold are the system components that are described in this chapter.

Specifically for the truck loading task, the task state contains information about the initial configuration of the implements, the current soil conditions, the desired location to dump the soil, and the location of the next dig point. Much of this information is already found in the form of the command parameters that are used in the parameterized script. Other information about the work environment that is not needed for the parameterized script may also be required for the task state. As shown in Figure 4.1, the task state becomes one half of the *task state-action pair*, which is used later in the system, and is also sent to the *action parameter computation function*, which is described in Chapter 8.

4.1.1 Example Task States

Returning to the Example Task that was introduced in Chapter 3, its task state is composed of six variables. Recall that the Example Task is to swing to the truck, open the bucket, and swing back to the dig face. Four task state variables are the four command parameters that define where to swing to at the truck and dig face, and the start and goal angles of the bucket. The fifth task state variable is the initial angle of the swing at the dig face, which has also been seen before.

The sixth task state variable is a new parameter that specifies information about the nature of the material in the bucket. This parameter is the angle at which the material will fall out of the bucket, which can change for different material conditions. For very sticky material, for instance, the bucket may need to be opened more than for very dry material. The six task state variables for the

Example Task are summarized in the Table 4.1.

symbol	description
$init\theta_{sw}$	initial angle of swing at dig face
$init\theta_{bk}$	initial angle of bucket at dig face
$dump\theta_{sw}$	desired swing angle at truck
$dump\theta_{bk}$	angle of bucket at which material falls out
$open\theta_{bk}$	angle of bucket to open to (could be the same value as $dump\theta_{bk}$)
$dig\theta_{sw}$	desired final swing angle at dig face

Table 4.1: Example Task State Variables

4.2 Actions

Actions are simply the action parameters, which affect the coordination of the sequence of task motions. Once the adaptive motion planning algorithm has the task state information, it must then determine the appropriate joint coordination that will achieve the best task performance. Recall that the joint coordination is defined by the script state transitions, which are dictated by the values of the action parameters in the script rules.

4.2.1 Example Task Actions

For the Example Task there are two actions, which are the two action parameters. These are summarized in Table 4.2.

symbol	description
$a_{sw\text{-}bk}$	swing angle that causes bucket joint to open
$a_{bk\text{-}sw}$	bucket angle that causes swing joint to return to dig face

Table 4.2: Example Task Actions

4.3 Action Decoupling

The combined space of task states and actions can be very large. Learning or optimization algo-

rithms that seek to uniformly sample this space may take a long time to do so. It would be helpful if the overall task state-action space could be broken down into smaller subspaces. For instance, if each action parameter could be completely isolated, then the n-dimensional action space could be made into n 1-dimensional action spaces. A simple discretized search across all values of each action parameter could then be an acceptable technique for selecting a value of each action.

As this research progressed, it became clear that certain task state variables and actions have very little, if any, bearing on other task state variables and actions due to the sequential nature of the task motions. For example in the truck loading task, the final dig location at the end of the excavator's free motion has no connection to the initial motion of raising the bucket to clear the truck, and vice versa. Changing the final dig location at the end of the task should not change the starting motions of the task. If there were a way of identifying which task states and actions influence other motions, then it may be possible to isolate different portions of the task. The script dependency graph presented in Chapter 3 can be used to identify these separate task portions because it graphs the motion dependencies of the parameterized script.

Consider the simplest case of a script dependency graph shown in Figure 4.2, which involves one pair of joints, Joint 1 and Joint 2. There are two script states in the graph. They are linked together by one action parameter $a_{1\text{-}2}$, thus Joint 2's motion is dependent on Joint 1.

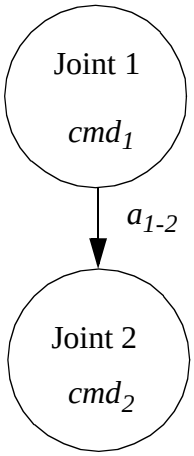


Figure 4.2: Simplest case of a script dependency graph.

For this simple case, there are at least four task state variables that influence the choice of the value of the one action parameter as well as its valid range. The four task state variables are the initial locations of Joints 1 and 2, and the desired goal locations of Joints 1 and 2, the latter two also being the command parameters cmd_1 and cmd_2. There also may be additional task state variables that provide other necessary information about the work environment.

Although this five-dimensional task state-action space may appear unwieldy, recall that the task state is already computed for each task execution cycle based on environmental information. The adaptive motion planning system is only responsible for selecting a value of the action for the

given task state. For the case shown in Figure 4.2, there is only one action value to select.

4.3.1 Independent Actions

Now consider a third state that is added to the simple script dependency graph as shown in Figure 4.3. This new state involves a new joint, Joint 3, and a new action, $a_{2\text{-}3}$, which links the motion of Joint 2 and Joint 3. Although it appears as if there are now at least two more additional task state variables (Joint 3's initial and goal locations) and one more action in the total task state-action space, it is possible to separate this motion into two task state-action spaces.

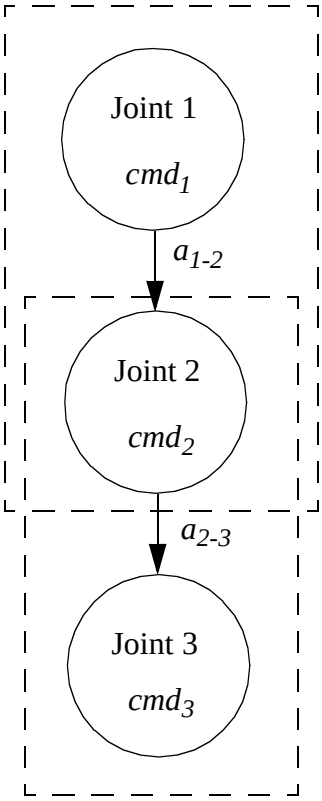


Figure 4.3: Script dependency graph with two independent action parameters.

Consider the vehicle motion for the script that would produce the script dependency graph of Figure 4.3. First, Joint 1 starts to move from its initial location to its goal location. At some point during Joint 1's motion, which is given by the value of $a_{1\text{-}2}$, Joint 2 begins to move from its initial location to its goal location. At some point during Joint 2's motion, which is specified by the value of $a_{2\text{-}3}$, Joint 3 begins to move to cmd_3. From Joint 3's perspective, it does not matter how Joint 2 begins to move, whether it simply began on its own or was triggered by the motion of other joints. Therefore, regardless of the value of $a_{1\text{-}2}$, or Joint 1's initial and final locations, the motion of Joint 3 with respect to Joint 2 is unaffected.

The dashed boxes in Figure 4.3 show which parts of the script dependency graph can be isolated into separate task state-action spaces. Like the simple script dependency graph of Figure 4.2, each sub-space has four task state variables and one action variable. Two task state variables, Joint 2's initial and goal positions, appear in both task state-action spaces. The two dimensional search for the values of the action parameters has now been broken down into two one-dimensional searches, a clear win.

4.3.2 Coupled Actions

Not all task state-action spaces can be decomposed into smaller sub-spaces, however. The script dependency graph of Figure 4.4 shows one obvious example of action coupling. The child state of Joint 3 has two parent states linked by two different action parameters. Thus, the values of both actions affect when Joint 3 begins its motion because the two comparison functions of this particular script rule must both evaluate to true for the script state transition to occur and Joint 3 to begin to move.

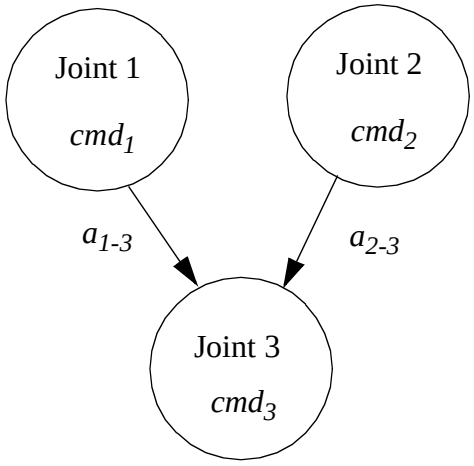


Figure 4.4: Script dependency graph with two coupled actions.

Figure 4.5 shows a more subtle case of action coupling. This script dependency graph appears similar to the one from Figure 4.3, however there are only two joints involved instead of three. The first and last nodes of the graph are the same joint, which has two different motions during the task (such as the swing joint moving to truck and moving back to the dig face). Between the two motions of Joint 1, Joint 2 begins its motion.

Consider a description of the script that would produce this script dependency graph. Joint 1 begins to move towards its first goal cmd_{1a}. Its motion triggers the commencement of Joint 2's motion, which in turn, causes Joint 1 to change its current motion and move towards its second goal $cmd_{1\text{b}}$. Joint 1 may or may not reach its first goal, which could have consequences on how the vehicle's performance is rated. It makes sense, then, to couple the two action parameters $a_{1\text{-}2}$ and $a_{2\text{-}1}$ because of this effect.

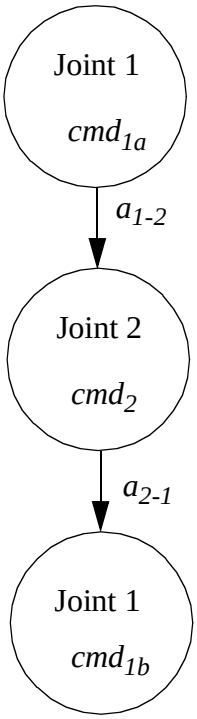


Figure 4.5: Another case of action coupling.

This same action coupling phenomenon could occur with more distant ancestor nodes. For example, the script dependency graph shown in Figure 4.6 begins and ends with the motion of Joint 1, but there could be many intermediate nodes. It may be possible to select values of all the actions that are involved such that Joint 1 is still on its initial motion by the time the command is sent to change to its final motion. If this were the case, then there is no hope of separating the task state-action spaces into smaller spaces.

In order to simplify the task state-action spaces, we will make an assumption that only grandparent/grandchild nodes of a script dependency graph that are the same joint will have coupled actions. Any nodal relationship that is more distant will not be considered a coupled action case of this kind.

The strongest argument for the validity of this assumption is the sense of task functionality. The cases in which more distant ancestors affect joint motion that occurs later in the script usually means poor or unexpected task performance. For example, it is possible to select values of the truck loading script action parameters such that the bucket opens immediately after digging and dumps the material back on the dig face. In this case, the task is clearly not being performed as desired. If the adaptive planning algorithm begins to explore this bad region of the action space, then something has gone wrong. Therefore, it is crucial that the adaptive motion planner finds initial sets of actions that are functional, even if they are not the optimal ones to take based on the desired vehicle behavior.

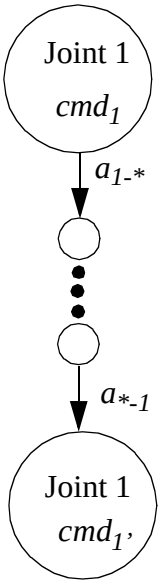


Figure 4.6: Script dependency graph with many nodes in between the two motions of Joint 1

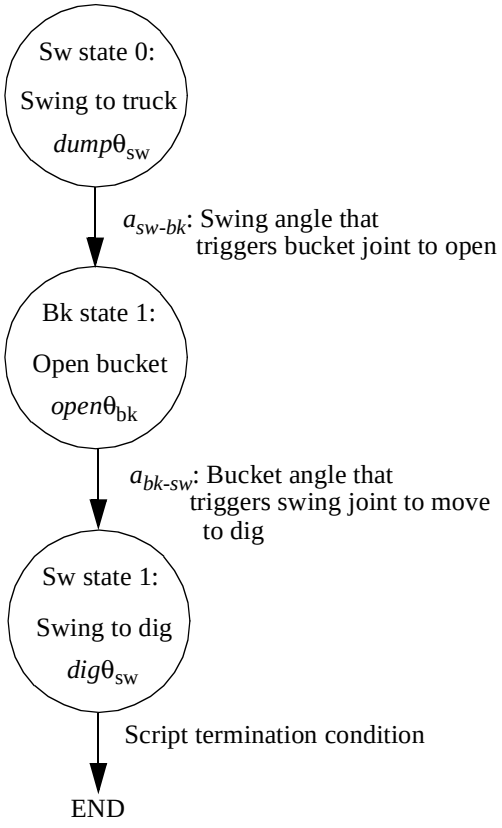


Figure 4.7: Script dependency graph for the Example Task.

Referring to Figure 4.7, which displays the script dependency graph for the Example Task, it is unfortunately a case in which the actions cannot be decoupled. Thus, the task state-action space contains six task state variables and two coupled actions. Task state-action decoupling has more benefits on larger parameterized scripts, such as the one constructed for the truck loading task.

4.4 Truck Loading Task States and Actions

The combined task state-action space for the truck loading parameterized script is enormous with thirteen command parameters, nine action parameters, and additional task state parameters. Fortunately, the total task state-action space can be broken down into four smaller subspaces using the analysis and assumptions that were presented in Section 4.3.

Figure 4.8 redisplays the script dependency graph that was shown in Chapter 3. The four boxes show the four separate task state-action subspaces that were created from the graph. Three of the subspaces have one action parameter, and the fourth has a four-dimensional action space. Four dimensions is unfortunately still high, but better than nine dimensions.

Two of the nine action parameters, $a_{bk\text{-}st}$ and $a_{sw\text{-}bk}$ were not included in the final version of the adaptive motion planning system. As this research progressed, only the most important action parameters were first included, with more being added as the system was developed and tested. In the end, these last two action parameters were never added in, and we believe that they would have minimal effect on the final system results. If they were added, both would constitute their own separate task-state action subspace.

The four separate task state-action spaces can be thought of as subtasks of the overall truck loading task, each with its own task state variables, actions, and desired behaviors. For example, one subtask involves coordinating the boom and swing joints to quickly move the bucket to the truck without a collision, while another subtask involves the dumping motion itself. These subtasks are described in more detail in the next sections.

4.4.1 *Boom Up* Task States and Actions

The *Boom Up* task state-action space concerns the initial motions of the truck loading task. As the boom is raising to its clearance angle immediately after digging, the swing is commanded to begin moving to the truck. Four task state variables come from the initial and goal positions of the swing and boom. The action parameter $a_{bm\text{-}sw}$ defines the point at which the swing begins to move in relation to the boom's motion.

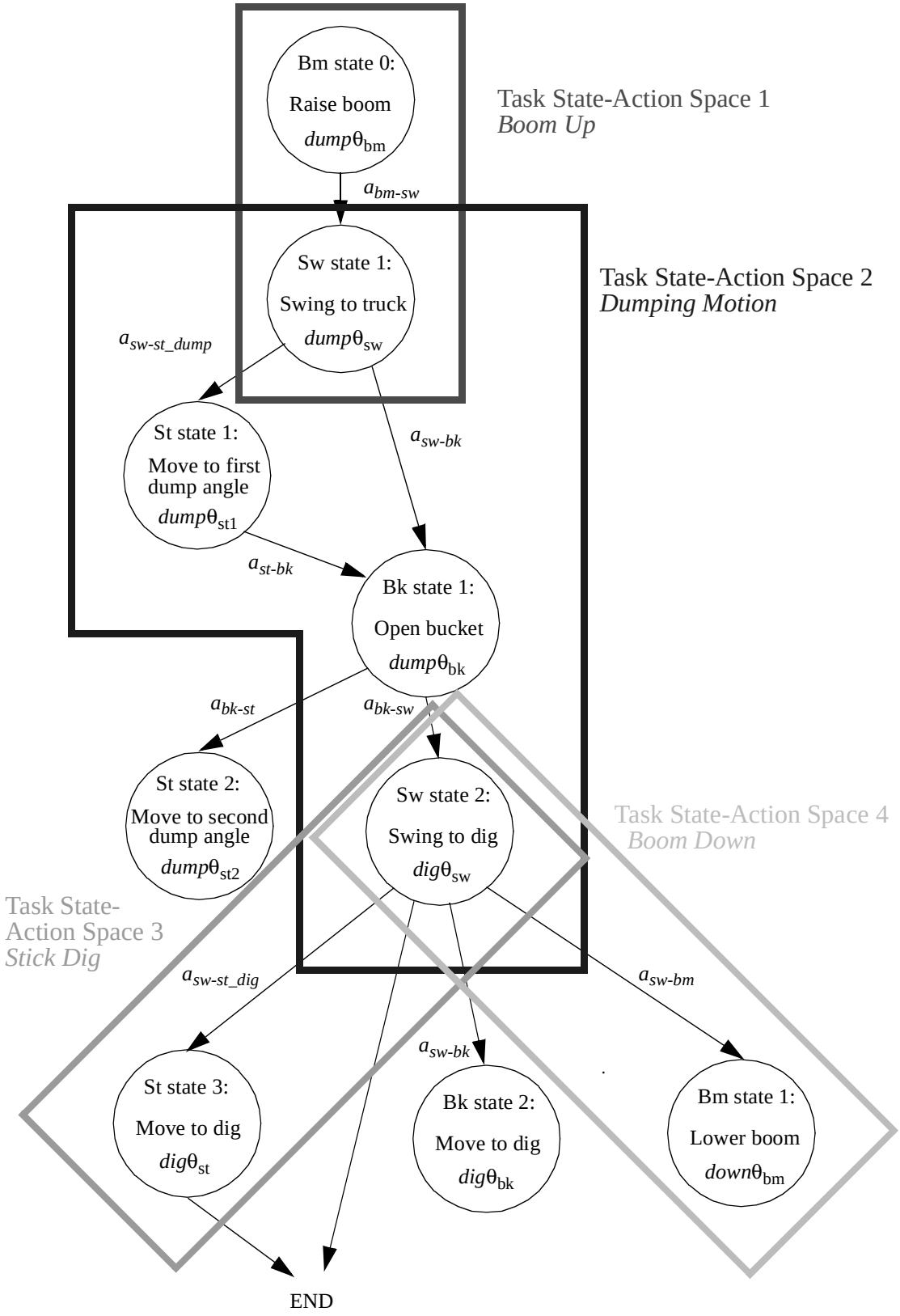


Figure 4.8: Truck Loading Script Dependency Graph

4.4.1.1 Additional Task State Parameters

One new parameter is needed for this task state-action space. Since the desired behavior of this subtask is to avoid hitting the truck, the location of the nearest point on the truck to the excavator is required. This parameter is the swing angle that corresponds to the rear left corner of the truck bed, as shown in Figure 4.9. This task state variable is symbolized as $truck\theta_{sw}$.

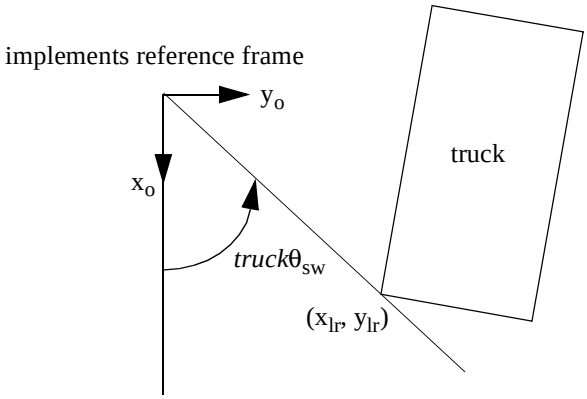


Figure 4.9: Schematic showing the swing angle that corresponds to reaching the truck.

Table 4.3 shows the task state-action variables for the *Boom Up* subtask.

symbol	description
$init\theta_{sw}$	initial swing angle
$init\theta_{bm}$	initial boom angle
$dump\theta_{sw}$	desired swing angle at truck
$dump\theta_{bm}$	boom clearance angle
$truck\theta_{sw}$	swing angle that corresponds to rear of truck
$a_{bm\text{-}sw}$	boom angle that causes swing joint to move to truck

Table 4.3: *Boom Up* Task State and Action variables

4.4.2 *Dumping Motion* Task State-Action Space

The task state-action space for the *Dumping Motion* subtask is the most complicated of the four. The dumping motion involves the swing, stick, and bucket motions as the bucket is brought to the

truck and the material is deposited. Six task state variables are the initial and goal positions of the swing, stick, and bucket joints. There are four action parameters involved: $a_{sw\text{-}st_dump}$, which is the swing angle at which the stick moves to its first dump goal position, $a_{sw\text{-}bk}$, which is the swing angle at which the bucket opens, $a_{st\text{-}bk}$, which is the stick angle at which the bucket opens, and $a_{bk\text{-}sw}$, which is the bucket angle at which the swing returns to the dig face.

4.4.2.1 Additional Parameters

The *Dumping Motion* task state-action space requires two additional parameters. The new parameters represent certain angles of the bucket that affect the behavior of the material in the bucket. The $spill\theta_{bk}$ parameter is the angle at which the material in the bucket just begins to fall out and the $dump\theta_{bk}$ parameter is the angle of the bucket at which all of the material has fallen out. These angles are dependent on the soil conditions. For example, dry, non-cohesive materials like sand may require a steeper angle to contain the material but not as steep a bucket angle to release the material, where stickier materials like wet clay may require less steep capture angles but larger release angles. These angles are shown in Figure 4.10.

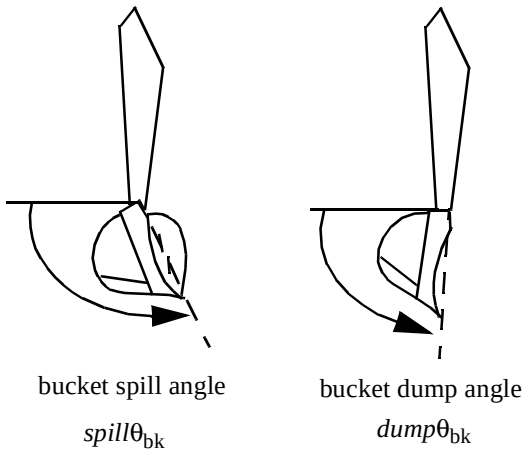


Figure 4.10: Bucket angle parameters for the *Dumping Motion* task state.

Table 4.4 summarizes the task state-action space for the *Dumping Motion* subtask.

symbol	description
$init\theta_{sw}$	initial swing angle
$init\theta_{st}$	initial stick angle

Table 4.4: *Dumping Motion* Task State and Action variables

symbol	description
$init\theta_{bk}$	initial bucket angle
$dump\theta_{sw}$	swing angle at truck
$dump\theta_{st1}$	stick angle for first half of dump motion
$open\theta_{bk}$	bucket angle to open to
$spill\theta_{bk}$	bucket angle, with respect to horizontal, at which material begins to fall out
$dump\theta_{bk}$	bucket angle, with respect to horizontal, at which material has completely fallen out
$a_{sw\text{-}st_dump}$	swing angle that causes stick joint to move to first dump angle
$a_{sw\text{-}bk}$	swing angle that causes stick joint to move to first dump angle
$a_{st\text{-}bk}$	stick angle that causes bucket joint to open
$a_{bk\text{-}sw}$	bucket angle that causes swing joint to move to dig face

Table 4.4: *Dumping Motion* Task State and Action variables

symbol	description
$dump\theta_{sw}$	swing angle at truck - initial swing angle
$dump\theta_{st2}$	stick angle for second half of dump motion - initial stick angle
$dig\theta_{sw}$	swing angle at dig location
$dig\theta_{st}$	stick angle at dig location
$a_{sw\text{-}st_dig}$	swing angle that causes stick joint to move to dig angle

Table 4.5: *Stick Dig* Task State and Action variables

4.4.3 *Stick Dig* Task State-Action Space

The *Stick Dig* task state-action space involves moving the stick to the desired dig location as the swing returns to the dig face. This is an important action because the swing and stick share the same hydraulic pump, so poor joint coordination can result in slower loading cycles. There are the four standard task state variables and no additional parameters. For this portion of the task, the

swing and stick starting angles are actually goal positions from previous script steps. Again, we are making the assumption that the vehicle behaves correctly, and that the previous swing and stick goals are achieved with reasonable proficiency. The task state-action space variables are summarized in Table 4.5.

4.4.4 *Boom Down* Task State-Action Space

The fourth and final subdivision of the total truck loading task state-action space is the *Boom Down* task state-action space. This subtask involves the boom lowering to its goal at the dig location as the swing returns to the dig face. Along with the four initial and goal position task state variables, the $truck\theta_{sw}$ constraint parameter that was used in the *Boom Up* task state-action space is also needed. The variables in this task state-action space are shown in Table 4.6.

symbol	description
$dump\theta_{sw}$	swing angle at truck - initial swing angle
$dump\theta_{bm}$	boom clearance angle - initial boom angle
$dig\theta_{sw}$	swing angle at dig location
$dig\theta_{bm}$	boom angle at dig location
$truck\theta_{sw}$	swing angle that corresponds to rear of truck
$a_{sw\text{-}bm}$	swing angle that causes boom joint to lower to dig angle

Table 4.6: *Boom Down* Task State and Action variables

4.5 Implementational Details

All four task state-action spaces involve the swing joint. Because the swing is minimally affected by gravitational dynamics[1], the start and goal positions become irrelevant for the same swing range. For example, swinging from $0°$ to $90°$ would be no different than swinging from $-100°$ to $-10°$. The direction of the swing's motion does not matter either.

With this in mind, the task state variables that involve the swing's initial and final positions can be collapsed into one variable, which is the swing range. This reduces one task state dimension from each of the task state-action subspaces.

1. Assuming the excavator's tracks are level.

4.6 Discussion

Task states and actions are two definitions that are required for representing the motion planning problem as a learning problem. The parameterized script motion planning algorithm creates a nice framework for defining task states and actions. Essentially, the command parameters, with a few additional parameters, are the task state, and the action parameters are the actions. The task states and actions define the inputs and outputs of the adaptive motion planning system. Chapter 8 discusses how sets of actions are found given a task state.

The method used for separating the task states and actions into separate subspaces may appear ad hoc, but it does help to simplify the problem. At first, an attempt was made to have each action in its own one dimensional subspace, which resulted in seven subtasks rather than the four that are present now. While this did help to find actions very quickly, problems were soon encountered where the same action value would result in drastically different machine motions for the same task state. This was due to the fact that the values of other action parameters were also influencing the vehicle motions. At that point, more analysis was done to determine which actions were coupled and why, leading to the ability to decouple actions from other actions by analyzing the script dependency graph.

One potential problem that this approach may have is that actions that can be decoupled by analyzing the script dependency graph might not be able to be truly decoupled due to coupling of the vehicle kinematics or dynamics. For example in the truck loading task, after dumping the soil and returning to the dig face, the boom joint's action (*Boom Down* subtask) and the stick joint's action (*Stick Dig* subtask) are decoupled. However, both joint motions affect the location of the bucket in space, in particular its elevation. There may be cases where boom and stick actions that are the best to execute in isolation, actually drive the bucket into the ground. In this case, these actions truly are coupled not because of parameterized script motion dependencies, but because of vehicle kinematic coupling.

The next chapter describes how the vehicle motion is evaluated and a reward is assigned to each task state-action set.

Chapter 5 Motion Evaluation: The Reward Function

The previous two chapters have presented a way of planning the motions of a vehicle to perform a given task, a means of changing the motion by modification of a small set of parameters, and definitions that are needed to extend the motion planning algorithm to an adaptive one. The next logical system component is a method of evaluating the motion of the vehicle after each task execution cycle in order to assign a score to each parameter set. This is known as a *reward* and is computed by a *reward function*.

One obvious evaluation criterion that can be used to compute a reward is the time it takes to execute each cycle. Recall for the truck loading task, a task execution cycle is one loading pass. Minimum loading pass times directly translate to higher productivity. However, time is not quite enough to evaluate a task execution cycle. There must be some notion of completion of the task. For example, a valid set of truck loading script parameters can be found which causes the excavator to immediately deposit the material that is in the bucket onto the dig face and not at the desired location in the truck. Although this loading pass[1] would be very fast, it falls short of achieving the desired task results.

This chapter introduces the notion of *task constraints*, which provide a second criterion for evaluating the motion of the vehicle. Task constraints define the desired behavior for a given task. For example, the desired behavior for the free motion of the truck loading task is to avoid hitting the truck with the bucket and to deposit the soil as close to the desired location as possible. Task constraints allow the vehicle's motion to be evaluated on the basis of how well the vehicle completed

1. Technically, this motion does have all the elements that constitute a loading pass.

the task.

Figure 5.1 shows the continuation of the system diagram for the adaptive motion planning system with the new system component that is described in this chapter shown in bold. As shown in the diagram, the *reward function* module receives the vehicle state history, which is acquired during task execution. The reward function analyzes the vehicle state history and produces a set of scores that rate both the task performance time and the error in satisfying the task constraints.

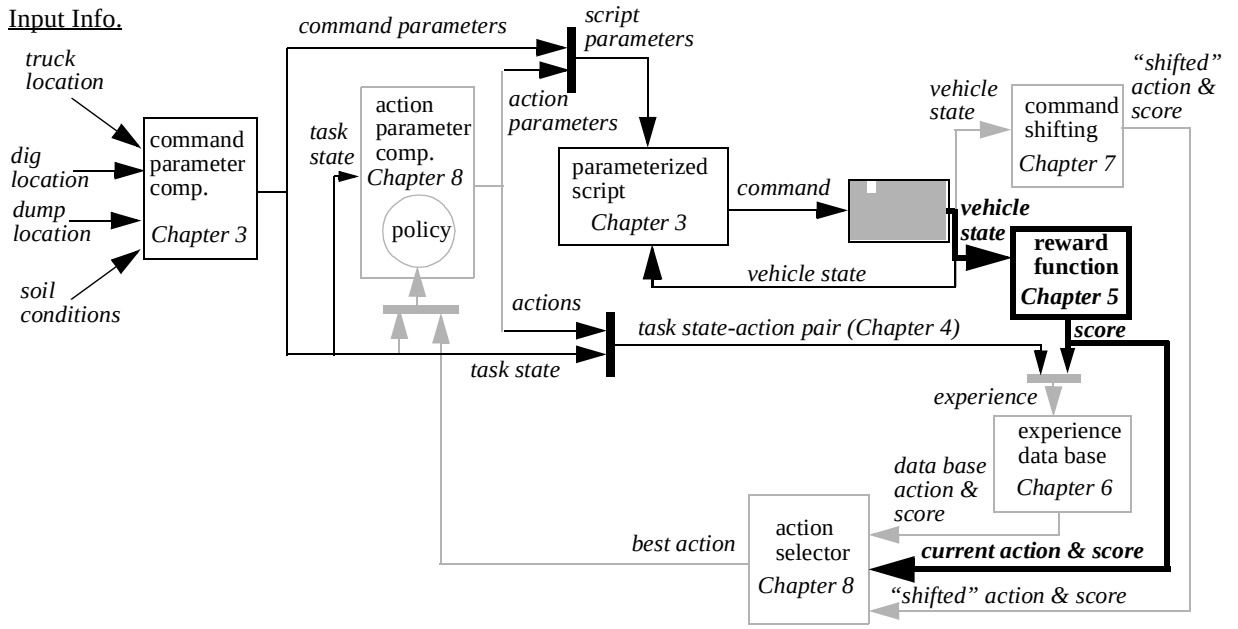


Figure 5.1: Block diagram of the adaptive learning system. In bold are the system components that are described in this chapter.

A pair of time/error scores are computed for each separate task state-action space that was defined in Chapter 4. For the truck loading task, a set of eight reward numbers would be produced, two scores for the four separate task state-action spaces. Rather than combining the two components of the score into one numerical value, the error component of the score is used as a constraint to prune the space of possible actions. *Error thresholds*, which define tolerances on the allowable task constraint error, are used to set these boundaries on the actions. Of the remaining actions, the time score is used to differentiate between good and poor actions.

The combination of a task state-action pair and the corresponding scores is referred to as an *experience*. Each loading pass produces one new experience.

The next two sections discuss the two criteria for evaluating a vehicle motion: execution time and task constraint error. Section 5.3 describes how the two components of the score are used together to help compute future actions. Section 5.4 presents an example of the reward space for the Example Task. Section 5.5 presents some sample reward results from the truck loading task. Finally, Section 5.6 presents a discussion of the reward function.

5.1 Execution Time

One of the goals of the autonomous mass excavation system is to maximize its productivity. One clear way that this can be done is by executing each loading pass as fast as possible. Therefore, the execution time of the task will be used as one element of the total reward.

As mentioned earlier, each individual task state-action space will receive its own execution time score. Recall that one task state-action space represents a smaller portion of the total task. For total task execution time to be a minimum, the times for each subtask should also be at a minimum.

In order for time to be used as a scoring metric, there must be some way of defining at what point a task begins and the point when it ends. Several different types of events are defined that will help in determining the start and end points of a task.

5.1.1 Start Events

Start events are the points in time when a joint begins its motion. This may not necessarily be the same time that the command to move the joint was sent to the joint controller. Long delays can occur because of the computational times involved in the controller software and the relatively large dynamic time constants associated with the hydraulic actuation system. Instead, the time that the joint perceptibly begins to move is desired. An example is shown in Figure 5.2. Notice the 1 second latency between when the command was sent and when the joint actually began to move.

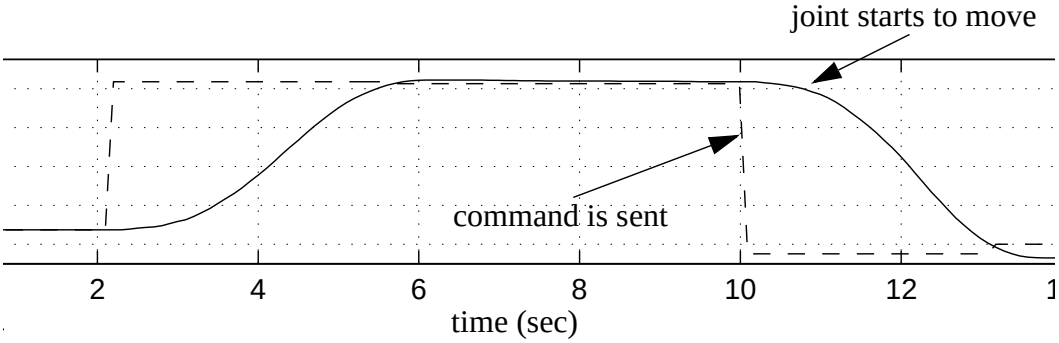


Figure 5.2: Joint trace displaying a start event.

A start event is detected by a joint exceeding a certain velocity threshold. A special case occurs if the range of motion is so short that the joint never reaches the given velocity threshold. If this occurs, then the start event time is set to the time the joint command was sent. One disadvantage of this approach is that explicit velocity thresholds must be provided to the reward function for each section of task motion where a joint begins to move (e.g. swing to truck, swing to dig face, bucket open, etc.). The values of these thresholds may be dependent on the characteristics of a particular vehicle, which, as stated earlier, we have no a priori knowledge of.

5.1.2 Target Events

Target events are the points in time when a joint reaches a certain angular position during a section of its motion. A specific script state, along with vehicle state information, may also be needed to put the target angle in context. For example, the swing joint moves through the same space twice during its motion, once on the way to the truck, and once on the way back to the dig face. If a target angle is specified somewhere between the truck and dig face, then the swing actually encounters the target twice[1]. However, this ambiguity is usually resolved automatically by the separation of the total task state-action space into smaller subspaces with separate time scores.

5.1.3 Goal Events

Goal events are the points in time when a joint reaches its desired goal angle. Determining goal events can be tricky because the joints asymptotically approach their commanded goals. Either position or velocity thresholds can be used to determine when a joint has come close enough to its goal. Figure 5.3 shows the detection of a goal event in a sample joint trace. Like the start events, goal events also require another set of predefined threshold values, which may be machine specific.

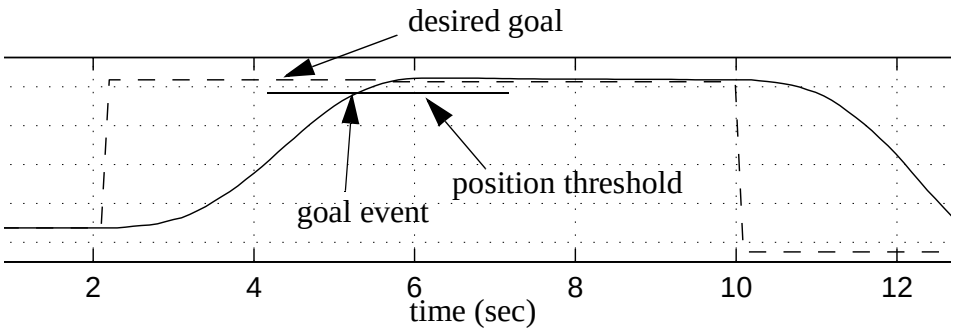


Figure 5.3: Joint trace displaying a goal event.

A failure condition occurs for both the target and goal events if neither the target or goal value is reached during the motion. For example, consider the joint trace shown in Figure 5.4. The swing joint comes nowhere near the desired goal of $90°$. In this case, the event detection portion of the reward function must report that this event did not occur during the motion, and there would be no way to assign a valid time score. However, if a case like this occurs, it is usually indicative of poor motion planning and means that something has gone seriously wrong. The time score can be set to an arbitrarily bad value, and the task constraint error scores, which are the second half of the total reward, will probably suffer as well for this failure condition.

1. An additional constraint could also be the direction of the velocity, but that is not guaranteed to disambiguate any generic target event.

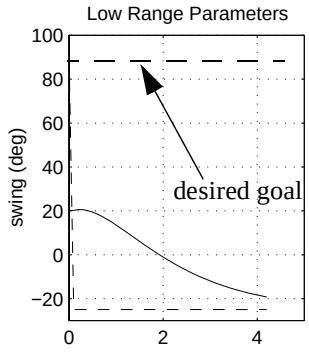


Figure 5.4: Joint trace showing a failure mode for finding target and goal events.

5.1.4 Example Task Time

Since there is only one task state-action space for the Example Task, there will only be one pair of time/error scores. For the Example Task, the start and end points of the motion are well defined. Time will begin once the swing-to-truck command is sent and end when the Example Task parameterized script terminates, which occurs when the swing comes within 1 degree of its goal angle at the dig face. In this case, the start event velocity threshold is 0, and the goal event position threshold is 1°.

Figure 5.5 redisplays the joint traces for the three Example Task motions originally shown in Figure 3.11 of Chapter 3. For the three vehicle motions, the time scores are:

low range parameters = 4.2 secs.
mid range parameters = 8.6 secs.
high range parameters = 13.6 secs.

The fastest vehicle motion is produced by the low range action parameters. However, as will be shown in the next section on the error score component, it does a poor job of completing the desired task, which is to deposit material in the truck. The vehicle motion from the high range action parameters is the slowest of the three, but it definitely fulfills the task description. Not surprisingly, the middle joint trace fall somewhere in the middle of the other two in terms of both time and task completion.

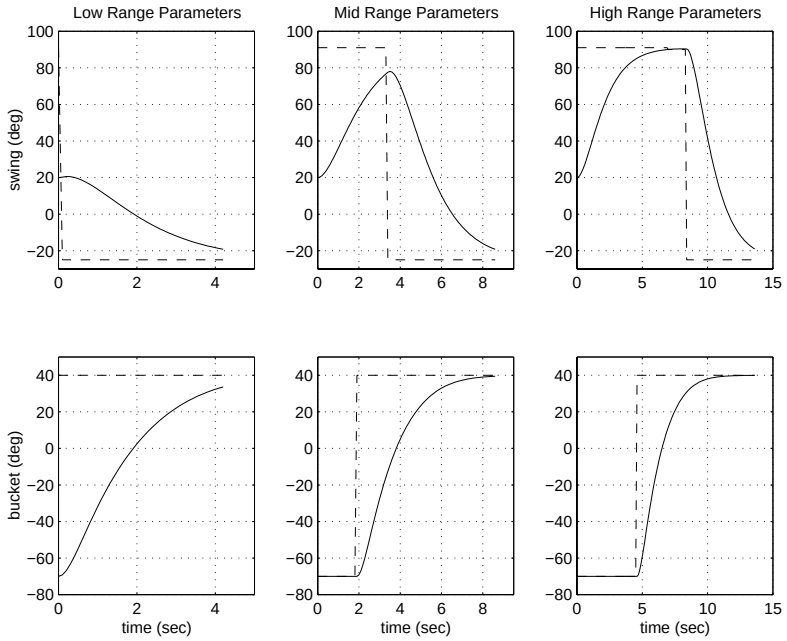


Figure 5.5: Vehicle motions produced by three different sets of action parameter values.

5.2 Task Constraint Error

The second component of the vehicle motion reward captures how close the vehicle motion came to completing the desired task. This is done by specifying *task constraints*, which are discussed in the next section.

5.2.1 Task Constraints

Task constraints specify the desired behavior of the vehicle for the given task. Like the parameterized scripts of Chapter 3, which define the task motion, task constraints also require human input to define the desired task behavior. In many cases it would be possible to use the same parameterized script for a certain task but change the desired task behavior, which may ultimately change the coordination of the scripted motions.

More specifically, task constraints are desired temporal relationships between the motions of two joints during one portion of the task. These joint pairs are related by a script rule, where the motion of one depends on the motion of the other. There are two types of task constraints that are used in the adaptive motion planning system.

Target-target: A target-target task constraint requires both joints to each reach a specific target value at the same point in time during the task motion. The target may be a joint's commanded goal angle or an intermediate angle encountered during its motion to its current goal. Target-target task constraints require two parameters, which are the specific values of the two targets.

For example, in the truck loading task, it is required that the boom joint reach its commanded clearance angle, $dump\theta_{bm}$, by the time the swing passes a certain angle that corresponds to reaching the rear of the truck, $truck\theta_{sw}$ (see Figure 5.6). This constraint is necessary to avoid a collision between the implements and the truck. The parameters for this task constraint are the boom's commanded position and the swing angle that corresponds to reaching the truck.

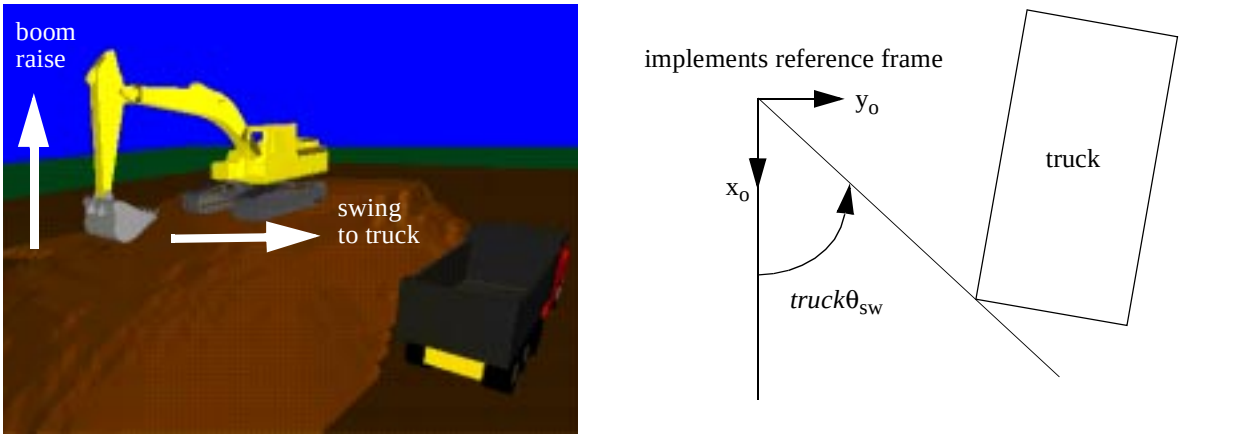


Figure 5.6: Target-target task constraint example.

Target-start: A target-start task constraint requires one joint to start its motion at the time a second joint reaches a certain target. This does not mean that the command to begin moving the first joint is sent at this time because of long delays between commanded and perceived motion. Target-start task constraints need one parameter, which is the value of the target.

An example of this type of task constraint occurs during truck loading when swinging back to the dig face. The boom cannot begin to lower to the desired dig location until the implements clear the truck bed. For this task constraint, the joint that is to begin to move is the boom, and the target parameter for the swing joint is again the $truck\theta_{sw}$ angle.

Even though task constraints are only specified for a pair of joints, constraint relationships can be expressed for any number of joints. In general, if there are n joints, then there need to be n+1 pairwise task constraints to fully constrain all of the joints.

Task constraints do not have to be equality constraints, where two joint angles must equal their respective targets at the same time. It is also possible to have task inequality constraints. This means that it would be acceptable for a joint to reach its target either before or after the other joint in the constraint relationship reaches its target. For example, returning to the above target-target constraint scenario, it would be acceptable to have the boom reach its clearance angle $dump\theta_{bm}$ *before* the swing reaches its truck target angle $truck\theta_{sw}$. This type of motion would also avoid a collision. Similarly, in the target-start constraint example, the boom could begin lowering to the dig face *after* the swing passes the truck target angle.

5.2.2 Example Task Error

For the Example Task, one or more task constraints can be constructed from the task description. Recall that the Example Task is to swing to the truck, open the bucket, and swing back to the dig face. The intended goal of the this task is to deposit the material that is in the bucket into the truck bed. This could imply that the bucket does not begin opening until the truck is reached, which would be a target-start task constraint relationship. Another possible task constraint could require that the bucket open to a certain angle that guarantees all of the material has fallen out by the time the swing reaches its commanded angle at the truck.

This latter task constraint will be the one that is used for the Example Task, although both could be selected. The chosen task constraint is a target-target equality constraint relationship. The bucket target is the angle that guarantees all of the material has left the bucket and is symbolized by $dump\theta_{bk}$. The swing target is the swing goal angle at the truck and is represented by $dump\theta_{sw}$. Both target parameters are also task state variables.

The parameter values for the task constraint targets for this example are:

$$dump\theta_{sw} = 90^{\circ}$$
$$dump\theta_{bk} = 0^{\circ}$$

One way to conceptualize a task constraint is as a point in the space that is formed by the pair of joints that are involved in the constraint relationship. This *task constraint point* acts as a boundary marker in the joint space. The joint space path may be required to pass through the task constraint point for an equality constraint, or remain on one side of the point for an inequality constraint.

The plots in Figure 5.7 show the joint space paths for the three different motions produced by the three different sets of script parameter values shown in Figure 5.5. For the Example Task, the joint space consists of the swing and bucket angular positions. The circle shows the task constraint point at $(90^{\circ}, 0^{\circ})$. This task constraint is a target-target equality constraint, so it is desirable for the path to pass through the point.

The dashed lines show the minimum distances from the joint space path to the task constraint point. This distance will be used as the numerical score for the error component of the total reward. Lower error scores mean better task performance. This distance metric is acceptable for task equality constraints.

For task inequality constraints, however, the joint space path may not need to come close to the task constraint point to still maintain acceptable performance as long as the path remains on the proper side of the task constraint point. In this case a large distance between the joint space path and the task constraint point would not mean poor task performance. The question is how to assign a score. One possibility is to make the error score be the best possible, which is 0°. This

may cause problems, however, with discontinuities in the function that is fit from the collected data that relates the action parameter value to the error score. Instead, the distance between the task constraint point and the path is still used, but is signed depending on which side of the task constraint point it passes.

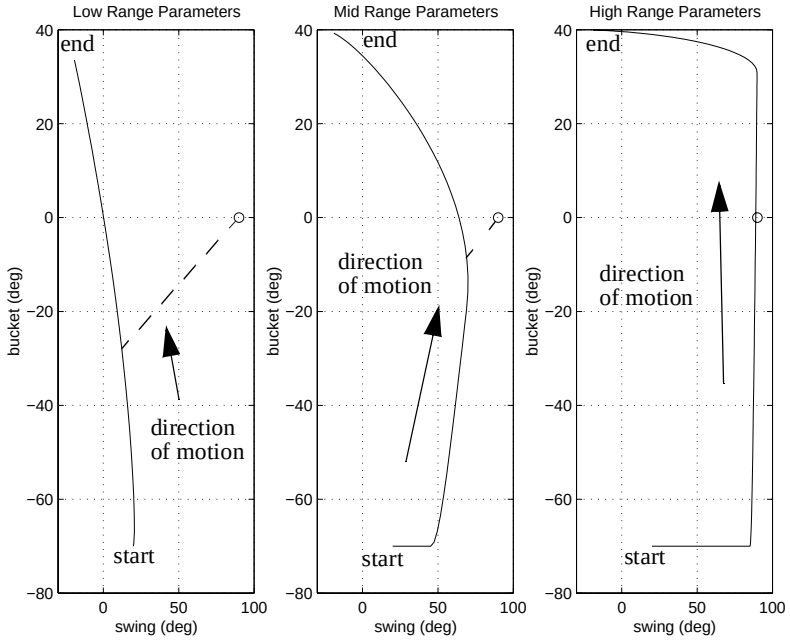


Figure 5.7: Plots showing the vehicle motions in joint space from the joint traces of Figure 5.5 and how close they come to the task constraint point.

For the three plots of Figure 5.7, the error scores are:

low range parameters = 82.6 °
mid range parameters = 13.7 °
high range parameters = 0.5 °

where the distances are measured in units of degrees.

As expected, the action parameter values from the low end of their range do very poorly and the corresponding joint space path comes nowhere near the task constraint point, resulting in a high error score. The middle range parameters are a little better, and the parameter values from the high end of the range produce a path that passes very close to the task constraint point.

Notice that the order of best to worst performance is now reversed when compared to the time component of the score. Clearly there is a trade-off between fast execution time and minimizing the task constraint error. These conflicting goals will be a fact that is true for the truck loading task as well. This trade-off makes sense. Faster execution times means that the vehicle's motion must come as close as possible to the boundaries which separate collision from non-collision and guar-

anteed task completion from motions which just barely meet task behavior requirements.

5.3 Combining Time and Error

At first, attempts were made to combine the two components of the score into one numerical reward that would then be assigned to the appropriate set of action parameter values for each task state-action space. One way to do this is to use a weighted sum shown in the following equation

$$reward = w_{time} time + w_{error} error$$

where w_{time} and w_{error} are adjustable weights.

A problem with this approach is selecting the values of the weights. It is not obvious how many degrees of error is equal to one second of time. Another problem is that very different task performance can result in the same score. For example, consider two different vehicle motions for the same task script. One motion may take 10 seconds to execute and only be 1° off in satisfying the task constraint. The second motion may only take 1 second of execution time, but have 10° of error with respect to the task constraint. This is entirely possible due to the conflict between speed and error. If both weights have a value of 1, then the combined score for both actions is 11. The fast, though inaccurate, motion may be completely unacceptable if 10° of error results in a collision. The slow, though more accurate, motion may be acceptable but just take longer to execute than preferred. Clearly there is a distinct difference between the two motions, but a single score provides no way of differentiating this information.

For the adaptive motion planning system, the time and error components are not combined into a single score. Rather, both numbers are kept separate. The error component of the score is used to prune the space of possible actions. This is done by defining an *error threshold* for the error score component. Error thresholds place limits on the acceptable task constraint error that can be tolerated.

In the portion of the action space that is within the error threshold, the time component of the score is then used as the action selection criterion. As in many constrained optimization problems, the optimal point will usually lie on the error threshold boundary. Thus, the choice of the error threshold value can play a dramatic role in the perceived productivity of the vehicle.

Error thresholds are discussed further in the contexts of searching for the best action among past experiences (Chapter 6), and in analyzing the state history of the vehicle to immediately find a good action (Chapter 7).

5.4 Example Task Reward Space

Let us examine what the time and error reward spaces look like for the Example Task. Figure 5.8 shows a set of contour plots which illustrate the complete time and error rewards over all possible action values for a constant task state. The bounds on the actions are a result of the task state values. The opposing nature of the time and error scores are clearly shown by the plots.

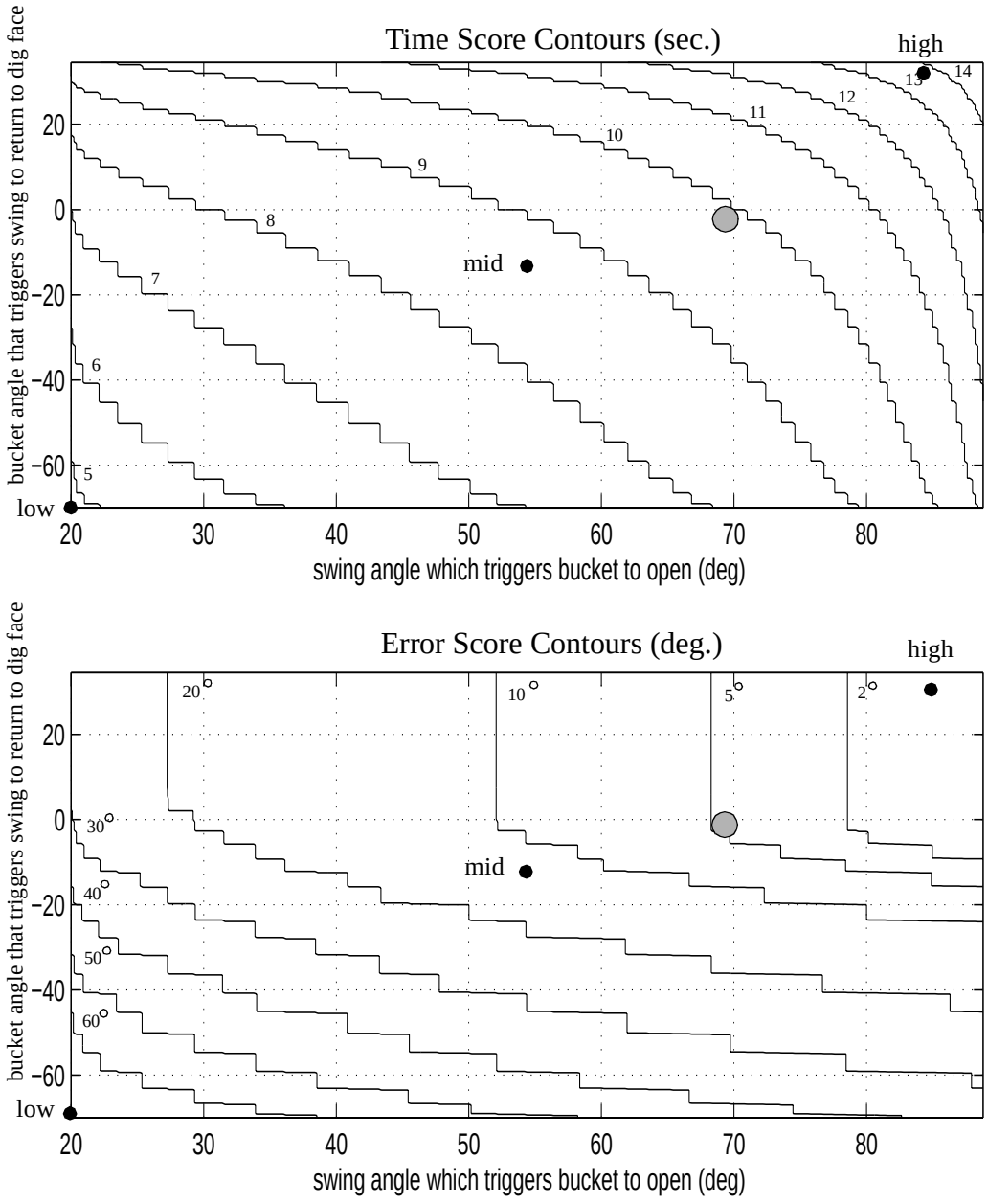


Figure 5.8: Contour plots showing the two dimensional action space for the Example Task for one task state. (Top) Contours of the time score. (Bottom) Contours of the error score.

These plots are shown for illustrative purposes only and it should be made clear that the adaptive motion planning system does not have this information ahead of time. It is this function that the memory-based learning techniques described in the next chapter will attempt to approximate.

The black dots labelled low, mid, and high represent the low, middle, and high range parameter value sets respectively. For sake of discussion, suppose $5°$ is set as the error threshold. In the context of the Example Task, this means that the swing must be within $5°$ of its goal when the bucket reaches the angle at which all of the material falls out. Notice that the choice of this error threshold value eliminates over 80% of the possible action space for this task state. The gray circle shows the area of the optimal action given this error threshold, which is along the $5°$ error contour.

One item of interest is the stair step nature of the contours. This is a direct result of the discrete time planner-controller interface. Vehicle states are received and vehicle commands are sent to the joint controller at a rate of 10 Hz in the current system. In the parameterized script paradigm, a command is sent to the joint controller when a vehicle state is received, because this is when the script rules are reevaluated. Because of this discrete time interface, there are contiguous values of action parameters that result in precisely the same vehicle motions, and thus the same scores.

For example, assume the swing joint is moving with a positive velocity of 50 deg/sec. Vehicle state updates are received every tenth of a second, so the swing will have moved five degrees between updates. Suppose swing angle positions are received with values of $0°$, $5°$, $10°$, etc. We wish to send a new command to the swing when it passes $46°$. However, there will be no difference in the vehicle motion if the command is sent when the swing angle passes $47°$, $48°$, or $49°$. The next chance the planner will have to send a command is at the 50 degree swing position update. Therefore, values of the action parameter between 45 and 50 degrees will result in the same command sequence and the same scores[1]. Higher vehicle velocities means larger areas of the action parameter space that result in the same command sequence, vehicle motion, and score. This is seen as larger plateaus in the middle of the contour plots when the vehicle joints are moving at their fastest. As the communication rate between motion planner and vehicle controller increases, the contours in the reward space will become smoother.

5.5 Truck Loading Task Rewards

This section describes how the time and error scores are computed for the truck loading task and shows some sample loading passes with their corresponding rewards. Recall that each separate task state-action space receives its own pair of time/error scores. For the truck loading task, there are four separate subspaces, which means that each bucket loading pass results in eight scores.

The task constraints for the truck loading task were constructed with the following behavior goals in mind:

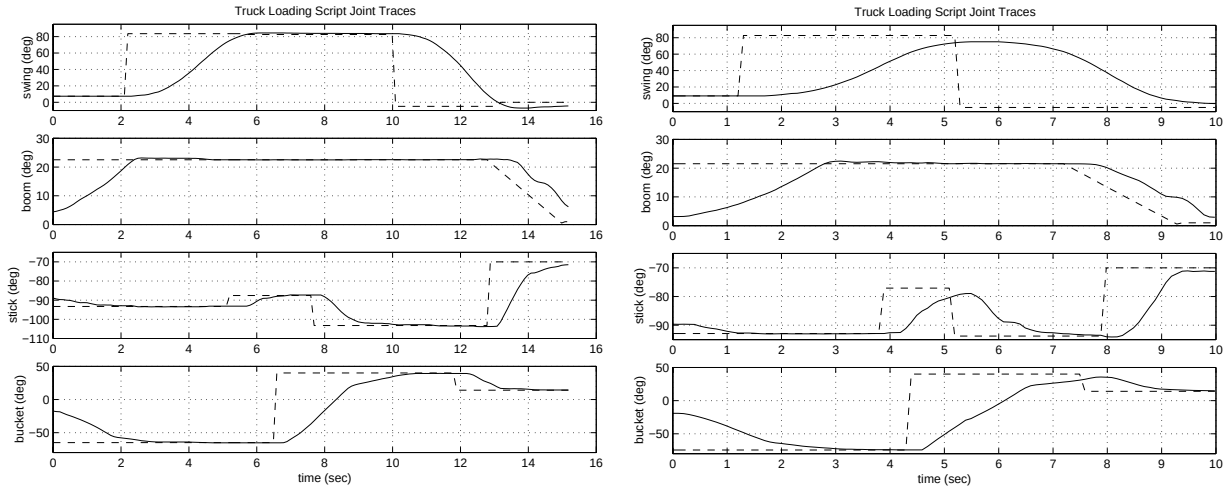
1. Inclusive or exclusive depends on the type of comparison that is used in the script rule.

- avoid all known obstacles such as the truck and unexcavated terrain,
- minimize spillage of the soil when dumping in the truck bed,
- complete each bucket loading cycle in the minimum time.

There are six task constraints in total, each of which will be described in the context of its particular subtask.

For the signed error scores, negative values mean that the joint space path passed on the wrong side of the task constraint point. Positive error scores, no matter their magnitude, are always acceptable.

The next four sections will show some sample time and error plots and scores for two different loading passes. The bucket loading passes have already been displayed in Figure 3.20 and Figure 3.21 from Chapter 3. Figure 5.9 redisplays the two vehicle motions. The one on the left is a slower motion taking nearly 16 seconds to execute. The set of joint traces on the right is a faster motion taking just under 10 seconds. The scores for each task state-action subspace should reflect how the time is saved between the left and right plots, but at the expense of worse error scores.



Bucket Loading Pass 1 Bucket Loading Pass 2

Figure 5.9: Joint traces from two sample bucket loading passes. (Left) Slower bucket loading pass. (Right) Faster bucket loading pass.

5.5.1 *Boom Up* Subtask

The *Boom Up* subtask involves the coordination of the swing and boom joints at the beginning of the motion. Digging has just finished, and the boom is raising to its commanded clearance angle. At some point during the boom's motion, the swing is commanded to move toward the truck.

5.5.1.1 Time

Figure 5.10 shows the time scores for the *Boom Up* subtask resulting from the two sample motions of Figure 5.9. The joint traces are plotted on the same time scale for comparison purposes. The start event for this task is the boom reaching a certain velocity during its raise motion, and the *Boom Up* subtask is completed when the swing passes the angle that corresponds to the rear of the truck, which is a target event. Notice that there is a difference of one second between the two sample motions. This is a direct result of different choices of the $a_{bm\text{-}sw}$ action parameter value, which can be seen by referring to Tables 3.15 and 3.16 from Chapter 3.

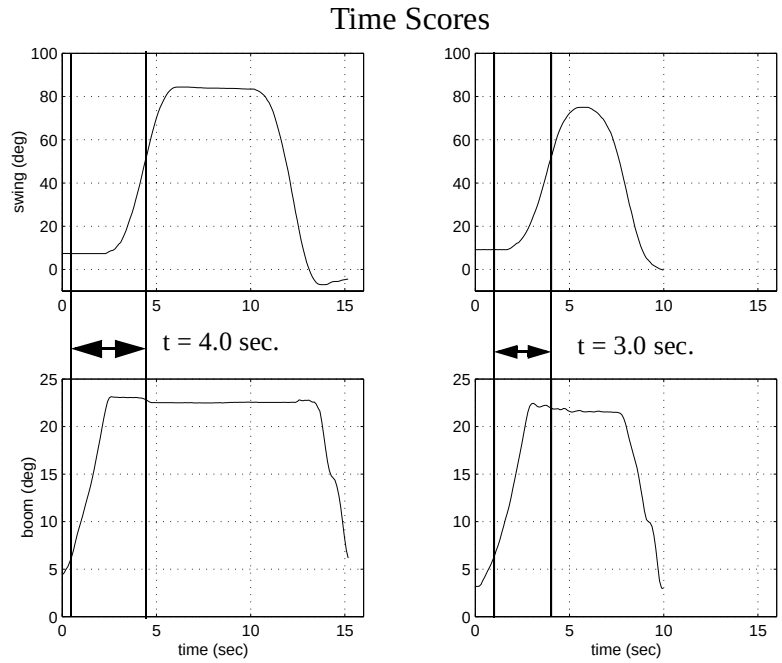


Figure 5.10: Swing and boom joint traces for two different loading passes.

5.5.1.2 Error

Figure 5.11 shows the paths in swing/boom joint space for the two sample loading passes. The circle represents the task constraint point for the particular task state from each motion. For this subtask, there is one task constraint:

- Boom must reach clearance angle $dump\theta_{bm}$ before the swing reaches the angle $truck\theta_{sw}$ which corresponds to the rear of the truck. This is a target-target inequality constraint and is needed to prevent a collision between the implements and the truck bed. Error scores for paths that pass above the task constraint point are acceptable and are given a positive sign. Error scores for paths that pass below the task constraint point are given a negative sign.

For both joint traces, the error is positive because the joint path passes on the correct side of the

task constraint point. The boom is high enough to clear the truck once the swing reaches the truck bed and would not cause a collision with the bucket.

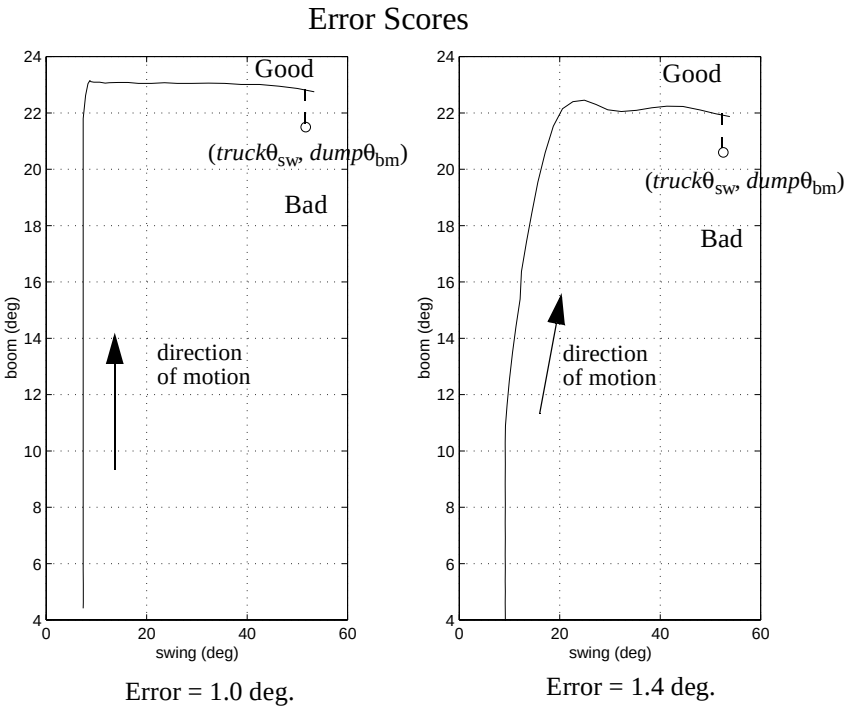


Figure 5.11: Joint space path in swing/boom space for two different loading passes.

5.5.2 *Dumping Motion* Subtask

The *Dumping Motion* subtask is the most complicated of the four separate subtasks. The *Dumping Motion* subtask involves the coordination of the swing, stick, and bucket joints during the dumping phase of the overall truck loading motion. As the swing is moving towards the truck, the stick is commanded to move to its first dump location, and the bucket is commanded to open. Once the bucket has opened beyond a certain angle, the swing is commanded to return to the dig face.

5.5.2.1 Time

Figure 5.12 shows the time scores for the two sample truck loading motions. The *Dumping Motion* subtask begins once the swing begins moving towards the truck. The task is considered complete once the swing begins moving back to the dig face. Both events are start events involving the swing joint. Notice that the vehicle motion on the left of Figure 5.12 takes nearly twice as long as the motion on the right. Again, this is due to the choice of action parameter values, which affect the coordination and timing of the scripted motions.

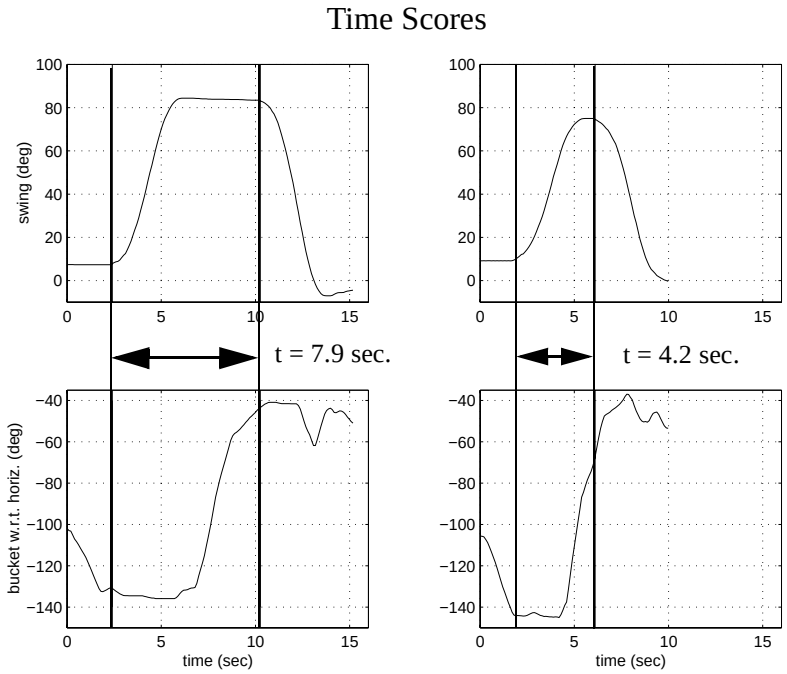


Figure 5.12: Swing and bucket joint traces for two different loading passes.

5.5.2.2 Error

There are three task constraints that are involved in the *Dumping Motion* subtask, however only the first two contribute to the error score. The third task constraint is designed to reduce the loading time, so its satisfaction is directly reflected in the time score. The three task constraints are:

- Stick must reach the first commanded dump angle $dump\theta_{st1}$ before the bucket reaches the angle $spill\theta_{bk}$ at which the material just begins to fall out. This is a target-target inequality constraint and is needed to control the radial direction of the bucket's motion during the dumping phase of the task.

- Swing must reach its commanded goal at the truck $dump\theta_{sw}$ before the bucket reaches the angle $dump\theta_{bk}$ at which the material has completely fallen out. This is a target-target inequality constraint and is needed to control the tangential direction of the bucket's motion during the dumping phase of the task.

- Swing must begin to move back to the dig face after the bucket reaches a certain angle, $dump\theta_{bk}$, at which all of the material has completely left the bucket. This is a target-start inequality constraint and is present to minimize the amount of time the swing dwells at the truck bed.

Figure 5.13 shows the joint space path of the stick and bucket joint pair during the dumping phase

of the motion. This plot illustrates the first task constraint listed above that involves the stick and bucket joints. The slower vehicle motion on the left produces a very accurate error score, while the path on the right sacrifices some accuracy for a savings in time.

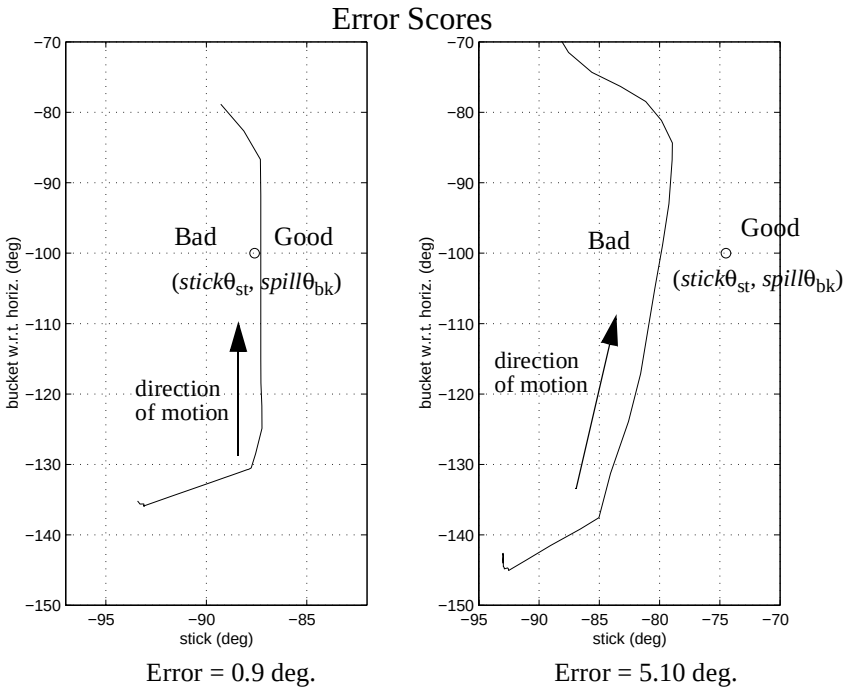


Figure 5.13: Joint space path in stick/bucket space for two different loading passes.

Figure 5.14 shows the joint space path for the second task constraint listed above. This task constraint involves the swing and bucket joints, thus the swing/bucket joint space is shown. Like the plots of Figure 5.13, the faster motion of the right hand plot is less accurate than the left hand plot, but completes the task in a shorter amount of time.

One way that the two error values are combined into one score is

$$error = \sqrt{error_1^{\,2} + error_2^{\,2}}$$

If either error score has a negative sign, then the final error score also has a negative sign. This is the method that is used for this subtask. This might not always be the best method however. An alternative is to keep the error scores separate, just as the time score is kept separate, and have separate error functions and thresholds for both. When pruning the space of possible actions with the error scores, the most constraining error threshold is used. In this case, there would be two thresholds, one to specify the allowable error in the lateral direction during the dumping motion, and one to specify the allowable error in the radial direction.

For the slower motion, the error score is 1.6° . For the faster motion, the error score is -8.33° .

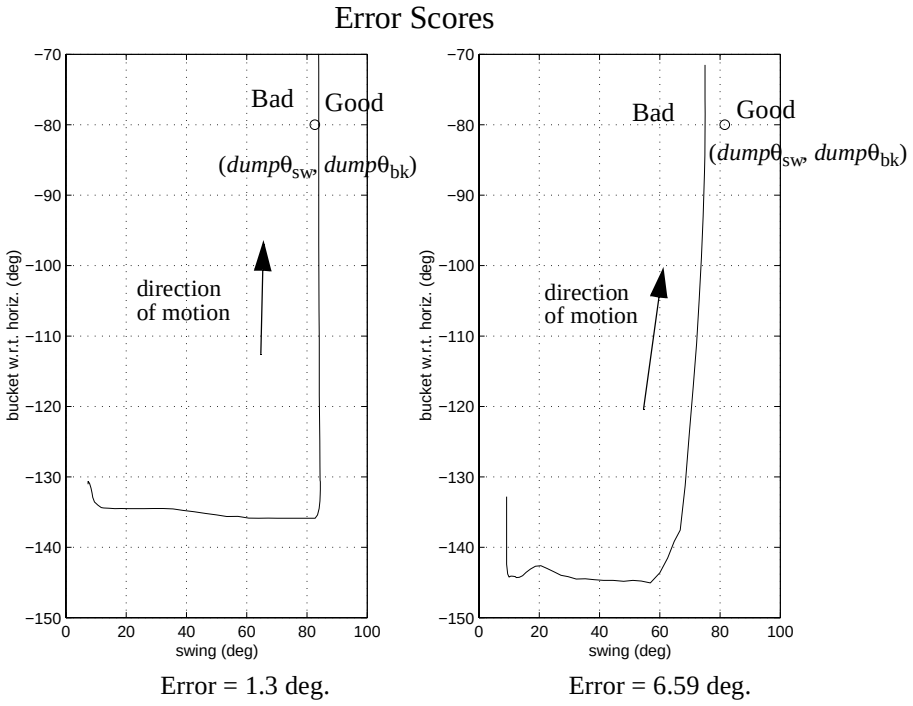


Figure 5.14: Joint space path in swing/bucket space for two different loading passes.

5.5.3 *Boom Down* Subtask

The *Boom Down* subtask involves the boom and swing joints as the swing is returning to the dig face. It is the counterpart to the *Boom Up* subtask, which seeks to avoid hitting the truck with the bucket while raising the boom. The *Boom Down* subtask aims to avoid hitting the truck with the bucket as the boom is lowering to the dig face.

5.5.3.1 Time

Figure 5.15 shows the time scores for the two different bucket loading passes. Time begins when the swing begins to move to the dig face. Time for this subtask ends once the boom begins to lower. Both are start events.

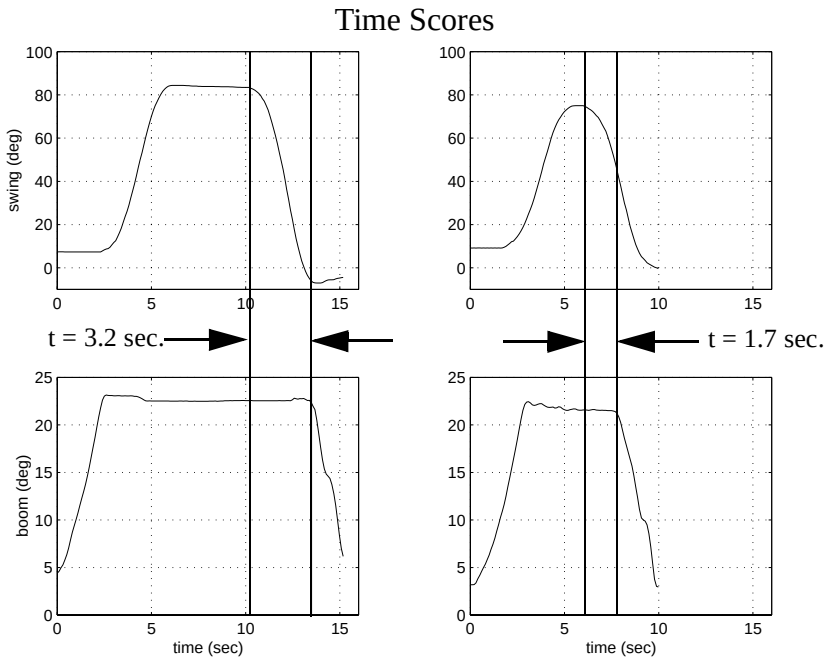


Figure 5.15: Swing and boom joint traces for two different loading passes.

5.5.3.2 Error

There is one task constraint for this subtask.

- Boom begins to lower to its dig goal after the swing has passed the angle $truck\theta_{sw}$ which corresponds to the rear of the truck. This is a target-start inequality constraint and is needed to prevent a collision between the implements and the truck bed. Positive error scores are acceptable. Negative error scores mean the boom began to lower too soon while it was still over the truck bed.

Figure 5.16 shows the paths in swing/boom joint space for the two truck loading cycles. The error is measured as the distance along the swing axis between the task constraint point and the point at which the boom begins to lower. At first, it may appear that the error and time scores do not conflict. However, high positive error scores are not necessarily bad in this case since the boom began to lower on the correct side of the task constraint point. Consider the case when the boom begins to lower immediately once the swing begins to move to the dig face. The time score would be very good, but the error would be a large negative number, which is unacceptable. As the error becomes positive and more acceptable, the time score increases and becomes worse.

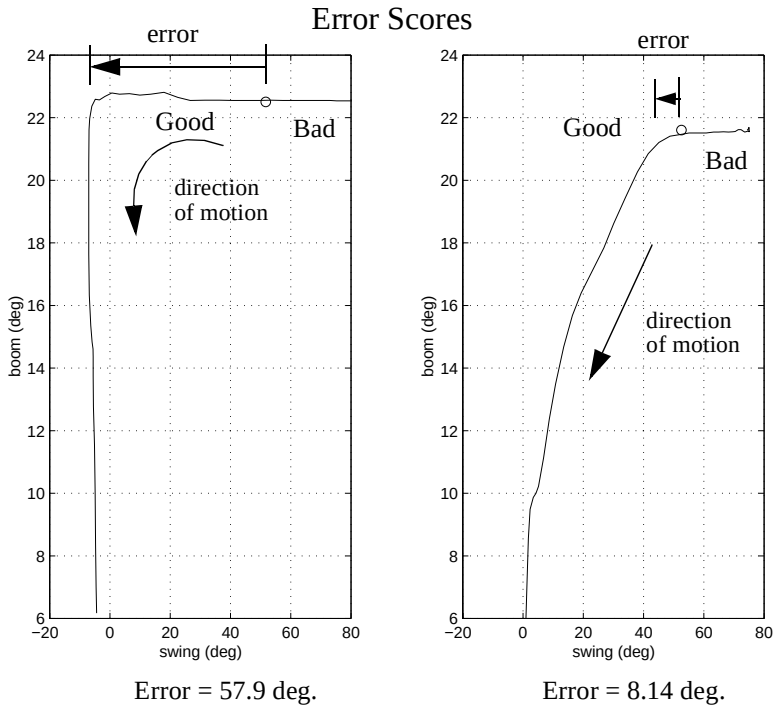


Figure 5.16: Joint space path in swing/boom space for two different loading passes.

5.5.4 *Stick Dig* Subtask

The Stick Dig subtask involves the motion of the swing and stick joints at the end of the overall truck loading task. Both the swing and stick joints are commanded to move to the next desired dig location at the dig face. Both events are termination conditions for the truck loading parameterized script and it would be ideal if both conditions were satisfied at the same time.

5.5.4.1 Time

Figure 5.17 shows the time scores for the two different bucket loading passes. Time for this subtask begins once the swing begins moving toward the dig face, which is a start event. The subtask is finished when both the swing and the stick joints are at their desired positions, whichever comes last. This is a goal event. In both vehicle motions of Figure 5.17, the swing takes the same amount of time to travel from the truck to its dig goal. However, the vehicle motions of the right hand plot are 1 second faster because the stick joint is commanded to move to its goal earlier.

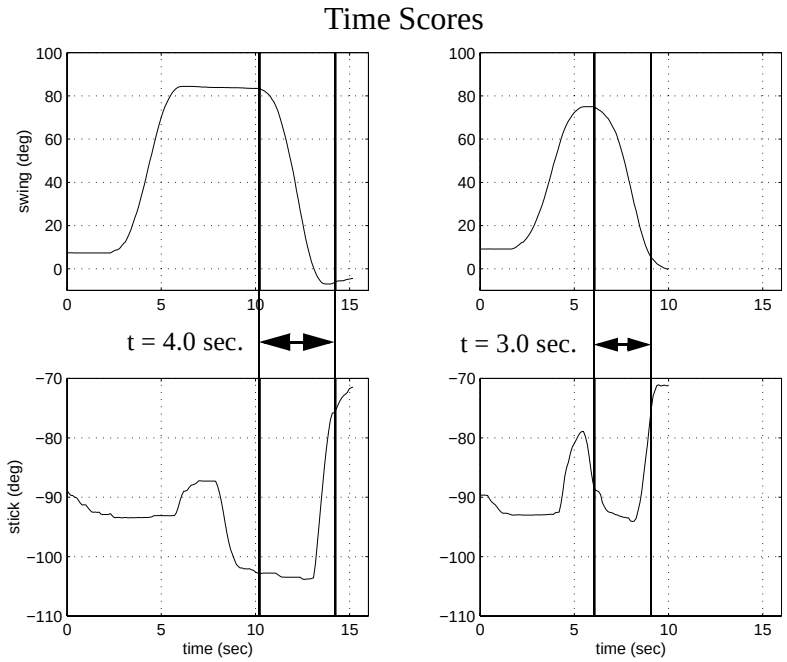


Figure 5.17: Swing and stick joint traces for two different loading passes.

5.5.4.2 Error

There is one task constraint for the *Stick Dig* subtask.

- Stick must reach its commanded dig angle $dig\theta_{st}$ before the swing reaches its commanded dig angle $dig\theta_{sw}$. This is a target-target inequality constraint and is needed to reduce the total bucket loading cycle time.

For this subtask, an error score is not very meaningful. The purpose of the task is to move both joints to their goal positions in the minimum time. Unlike the other subtasks, there are no obstacles to avoid or desired locations to place material that naturally produce an error metric. Therefore, for this subtask there is no error score component. This simply means that there is no pruning of the action space.

5.6 Discussion

The evaluation of the vehicle's motion for each task execution cycle involves not only evaluating speed of execution, but also how well the motion plan met the desired task objectives. Neither one alone is enough to fully evaluate the vehicle's performance.

In a way, the idea of task constraints are a return to the original idea of using via points to specify

a desired path of the bucket as described in Chapter 3. A task constraint, however, only involves the relationship between two degrees of freedom, as opposed to constraining all four joints of the excavator to a particular point in space. Inequality task constraints differ from via points in that the task constraint point does not need to be reached during the vehicle's motion. The path created by the vehicle needs only to stay to one side of the task constraint point. This makes task constraints more flexible than via points.

Plotting the vehicle motion in joint space offers some additional insights into the nature of parameterized scripts and the motion evaluation. Consider the motion of the boom and swing during the *Boom Up* subtask, which is displayed in Figure 5.18. Notice the rather roundabout path, shown as the solid line, which is needed to pass on the correct side of the task constraint point. Why not take a more direct route as shown by the dashed line?

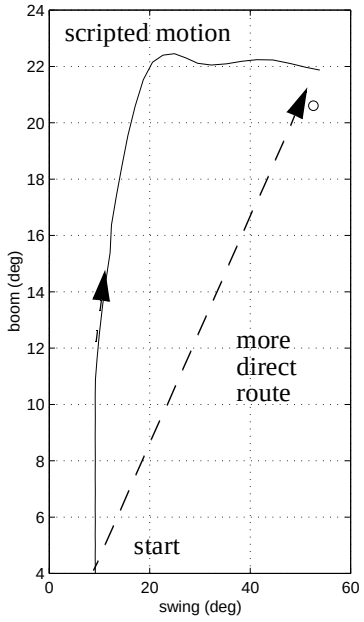


Figure 5.18: Joint space path in swing/boom joint space for the *Boom Up* subtask. (Solid line) Path produced by the parameterized script. (Dashed line) Direct path to the correct side of the task constraint point shown by the circle.

The path that is shown in joint space is both a function of the rules of the parameterized script and the closed-loop dynamic behavior of the excavator itself. The script rule for this particular motion states that the boom must begin moving first, after which the swing may begin to move towards its goal. The characteristics of the boom and swing joint controllers, as well as the hydraulic actuation system, define the relative velocities of each joint to one another, which ultimately define the shape of the path. This, coupled with a variable point in time to start swinging, creates a family of curves which are the only possible paths for this motion.

In order to achieve the motion shown as the dashed line in Figure 5.18, one or both of these elements would need to change. The dashed line represents a true trajectory. There would be no need for a script rule defining the swing motion relative to the boom motion. As stated in Chapter 3, the

controller on the excavator testbed is not a trajectory tracker and would track this trajectory very poorly for any reasonable velocity[1].

To determine if the straight line path is better than the scripted motion, experiments were performed. The autonomous system used the parameterized script to generate a path from start to the task constraint. A human operator followed the straight line path. The difference in time was less than 0.5 seconds, with the more direct dashed line path being slightly faster. Thus, the parameterized scripting approach sacrifices a bit of global optimality for both a reduction in the number of control options and the elimination of an accurate high-speed trajectory tracker for a hydraulic machine. This is a good trade-off considering the nature of the task and the hydraulic excavator. As mentioned in Chapter 1, highly accurate trajectory trackers are hard to obtain for large hydraulic machines with highly non-linear dynamic characteristics. The reduction in the number of control options leads to rapid improvement in performance, since the number of control parameters is small.

The next chapter describes where the task states, action parameter values, and rewards are stored, and how they are used to predict the score of any action taken in any task state.

1. The controller could track the trajectory very accurately if wanted, but at very slow joint speeds.

Chapter 6 Experience Data Base

So far, we have described two major system components: a way of planning and executing the motions of an excavator to perform the task of truck loading, and a method of evaluating the motion plan to assign a set of reward values. The combination of a motion plan and an evaluation is referred to as an *experience*. This chapter presents the next step in the adaptive motion planning system, which involves how experiences are used to predict the performance of untried actions. This capability can then be used to find the best set of actions for a particular task state given the experiences that have already been collected.

In the basic reinforcement learning algorithm presented in Chapter 2, there are four items that must be defined for an application: states, actions, rewards, and a policy update function. We have already encountered three of the four. This chapter, along with the following two, begins to describe the policy update function. Recall that a policy is a mapping from a state to an action, in other words, what to do in any given situation. The policy update function involves using the experiences to update the policy such that the system improves its performance over time.

Figure 6.1 shows the continuation of the system diagram that was begun in Chapter 3 with the new system component described in this chapter shown in bold. As shown in the diagram, each experience is stored in the *experience data base*. The output of the experience data base is a recommended action and a predicted set of scores for the current task state. In other words, the experience data base offers a suggestion of what action should be taken for the current task conditions based on what the system already knows. This is done by having the experience data base generalize across all possible experiences, and then searching for the one which gives the best scores in terms of time and task constraint error.

The next section describes the data base of experiences in more detail. Section 6.2 discusses how

the data base is used to generalize across experiences. Memory-based learning techniques are employed to make predictions of the rewards for any task state and actions. Section 6.3 describes how this prediction capability is used to search for the best action for a given task state. Implementation for the truck loading task is presented in Section 6.4. Finally, Section 6.5 presents a discussion of the use of memory-based learning techniques in the experience data base.

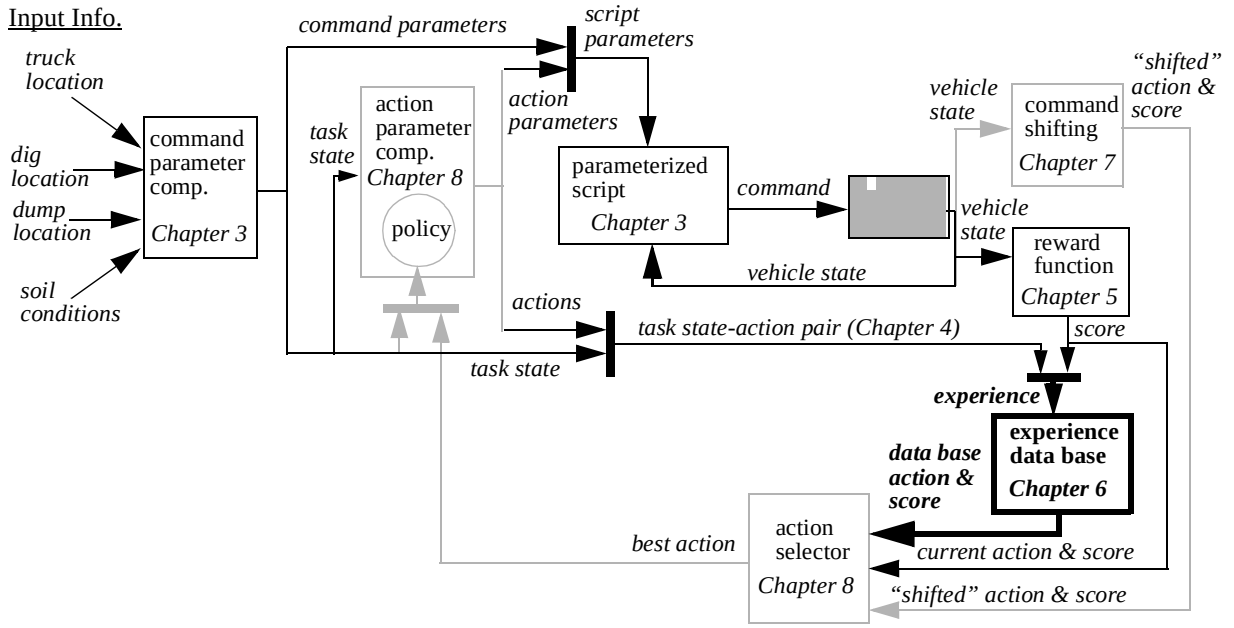


Figure 6.1: Block diagram of the adaptive learning system. In bold are the system components that are described in this chapter.

6.1 Experience Data Base

Figure 6.2 shows a conceptual model of the experience data base. Each experience that is stored in the data base consists of a task state, a set of actions, and a pair of time/error scores. This information is shown as the three axes in Figure 6.2, although there may be several dimensions to the task state, action, and score axes. In order to generalize across experiences in the data base, one could think of a surface in this three dimensional conceptual space. The surface would map task states and actions to a score.

There is a separate data base for each separate task-state action space. For the truck loading task, there would be four experience data bases. Each component of the reward, time and error, can also be considered to be stored in its own data base, as there may be different functions fitted to the time reward data and the error reward data (as shown by the different contour plots of Figure 5.8).

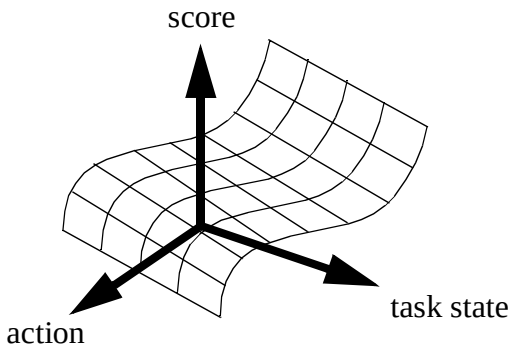


Figure 6.2: Conceptual diagram of an experience data base.

Because the values of the task state for each task execution cycle affect the valid ranges of the actions (see Section 3.3.6 from Chapter 3), there is a great deal of space in the experience data base that is never used. Consider the example shown in Figure 6.3. The x axis shows one dimension of a possible task state, a desired goal for the swing angle. The y axis is one dimension of the action set, which is also a swing angle. Because a value of the action that is beyond the desired swing goal would be meaningless, one half of this task state-action space is invalid. No experience data would be collected in this region because if an action value were selected from the invalid region, the parameterized script would get stuck and fail to terminate.

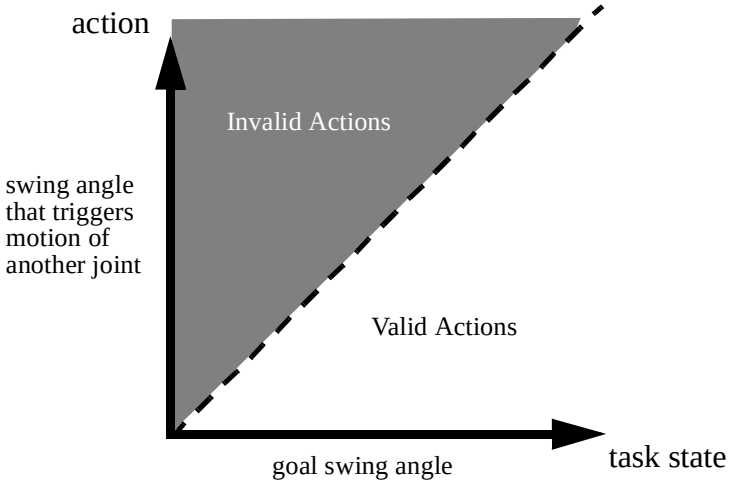


Figure 6.3: Simplified model of a task state-action slice from the experience data base.

Data description	Exp. 1	Exp. 2	Exp. 3
Task state: initial swing angle	20°	20°	20°
Task state: initial bucket angle	-70°	-70°	-70°
Task state: swing angle at truck	90°	90°	90°
Task state: angle of bucket at which material falls out	0°	0°	0°
Task state: bucket open angle	40°	40°	40°
Task state: swing angle at dig face	-20°	-20°	-20°
Action: swing angle that triggers bucket joint to open	20°	55°	85°
Action: bucket ang. triggering swing to move to dig face	-70°	-15°	30°
Score: time	4.2 sec.	8.6 sec.	13.6 sec.
Score: task constraint error	82.6°	13.7°	0.5°

Table 6.1: Data base of three Example Task experiences.

Another characteristic of the experience data bases is that the space is continuous rather than discrete. Task states, which involve starting, intermediate, or final locations of various joints, can take on any value, as can the action parameters which coordinate the vehicle's motions. This characteristic influences the choice of a data storage and prediction strategy.

Discretizing the space into bins and using it as a massive look-up table is one option. However, there are problems with selecting the proper resolution to discretize, and the fact that much of the space of the experience data base is invalid due to the boundaries shown in Figure 6.3 creates a great deal of wasted memory. Furthermore, as mentioned in Chapter 1, the excavator works in a small fraction of its total workspace on any given worksite, and there are a small number of dig and dump locations. This means that the same task states are encountered many times over the course of a work shift, resulting in several small clusters of data in the data bases with much of the space remaining empty.

Several researchers (Santamaria et. al., 98; Atkeson, 91; Moore et. al., 95; Sutton, 88) have advocated using memory-based learning techniques. Memory-based learning offers advantages in dealing with continuous state-action spaces. The space does not need to be arbitrarily discretized. In memory-based learning all of the motion experiences are remembered, so the state-action space records data precisely where it needs it. Memory-based learning uses function approximation techniques to generalize across the space of states and actions and predict the results of untried actions. Memory-based learning techniques are used in the adaptive motion planning system and

are presented in Section 6.2.

6.1.1 Example Task Experiences

So far in the course of developing the Example Task, we have seen three different experiences. These experiences were initially presented in Section 3.3.6 of Chapter 3, which compared the results of action parameter values taken from the low, middle, and high ends of their valid ranges. For each experience, the task state has remained the same. The three experiences that would be stored in the experience data base are shown in Table 6.1. Experience 1 involves action parameter values from the low end of the action parameter range, experience 2 uses the middle range action parameter values, and experience 3 is the result of the high range action parameter values.

6.2 Predictions

The main purpose of the experience data base is to provide predictions of the resulting scores for any task state and action. In order to do this, there must be some mechanism that can generalize the stored data to provide a prediction.

6.2.1 Memory-based Learning

Each and every experience that is collected by the adaptive motion planning system as the excavator performs its task is stored in the experience data base. This strategy of explicitly remembering all past experiences is known as *memory-based learning*. Several different memory-based function approximators were described in Chapter 2.

Locally weighted linear regression is the function approximator used to construct a model of the experiences in the experience data base. Early analysis of the data collected during system development showed that the mapping from task states and actions to scores is fairly linear. This makes sense in the context of the task. For example, recall in the *Boom Up* subtask, the boom angle triggers the swing joint to begin to move toward the truck. Figure 6.4 shows the resulting time and error scores for all values of the boom trigger angle for one constant task state. Boom trigger angles which are closer to the initial boom angle (left end of the action parameter range) result in shorter task times, and boom trigger angles that are closer to the final boom angle (right end of the action parameter range) result in longer task times. The relationship is mostly linear with only a slight change towards the high end of the boom angle range, when the boom joint's deceleration has a large effect on the task time. The error score plot is also linear over most of the range of the boom trigger angle. Similar results are obtained for the task state variables, such as the starting swing angle, versus the time component of the score. For example, swing angles that start farther away from the truck result in longer task times than swing angles which begin closer to the truck. Locally weighted linear regression is described in more detail in Appendix B.

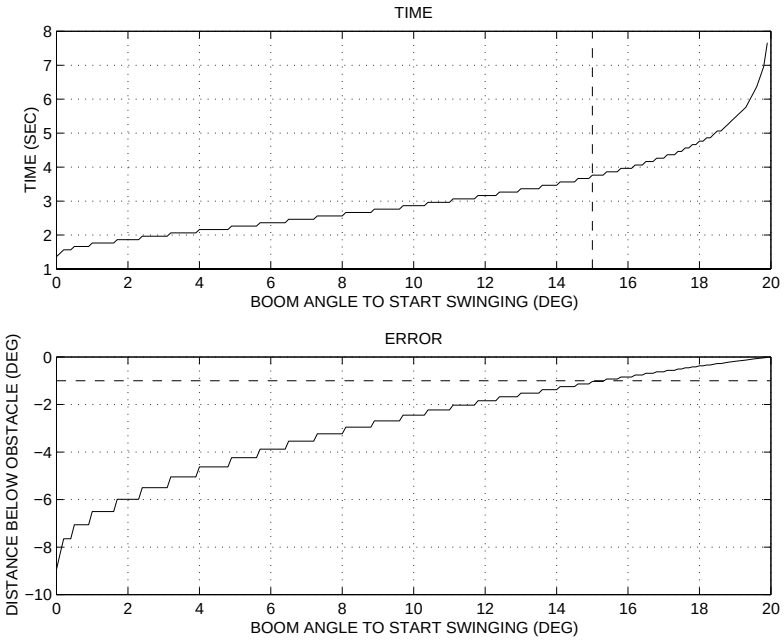


Figure 6.4: Time and error reward plots for the *Boom Up* subtask.

The experience data base cannot make a prediction if there is no data. Also, the prediction data base is not allowed to offer a prediction if there is not enough data either. In order to ensure matrices of full rank, there must be m+1 data points in the data base, where m is the number of input dimensions. This is the number of task state and action variables for a given subtask.

6.3 Finding the Best Action

This section describes how the experience data base is used to find the best action for a given task state. In order to do this, some searching functionality that queries the experience data base for predictions is needed.

After each bucket loading pass has been executed and evaluated, the experience data base suggests the best action for the current task state. In other words the experience data base computes the action for the current task state that results in the best predicted reward values. This means that all of the values of the task state dimensions in the experience data base are fixed. Only the values of the actions need to be searched to find the best one. This search is performed after each and every bucket loading pass.

There are many different search techniques to choose from. Because gradient information is readily available from the locally weighted linear regression, gradient descent may be an appropriate choice. Others could include simulated annealing if local minima are expected to pose a problem, or a downhill simplex method (Press et. al., 88).

For the adaptive motion planning system, a simple multi-resolution brute force search was implemented. The experience data base is first queried at a coarse resolution to find the region that contains the best result. This smaller region of the data base is then queried at a finer resolution to home in on the final answer. This is shown in the one dimensional example in Figure 6.5. Here, the minimum of the curve is sought.

6.3.1 Controlled Extrapolation

Extrapolation into regions of unknown or unexplored spaces of the experience data base can be dangerous in the application of loading trucks with a large, powerful excavator. With this in mind, extrapolation is dealt with very conservatively in the experience data base search.

The multi-resolution brute force search technique does not search over all possible values of each action parameter for a given task state. Rather, it is limited by the minimum and maximum action parameter values that are known so far. As more data points are collected, the boundaries of the search space are expanded, but the search is not allowed to consider any actions that are beyond the bounds of what it has already tried.

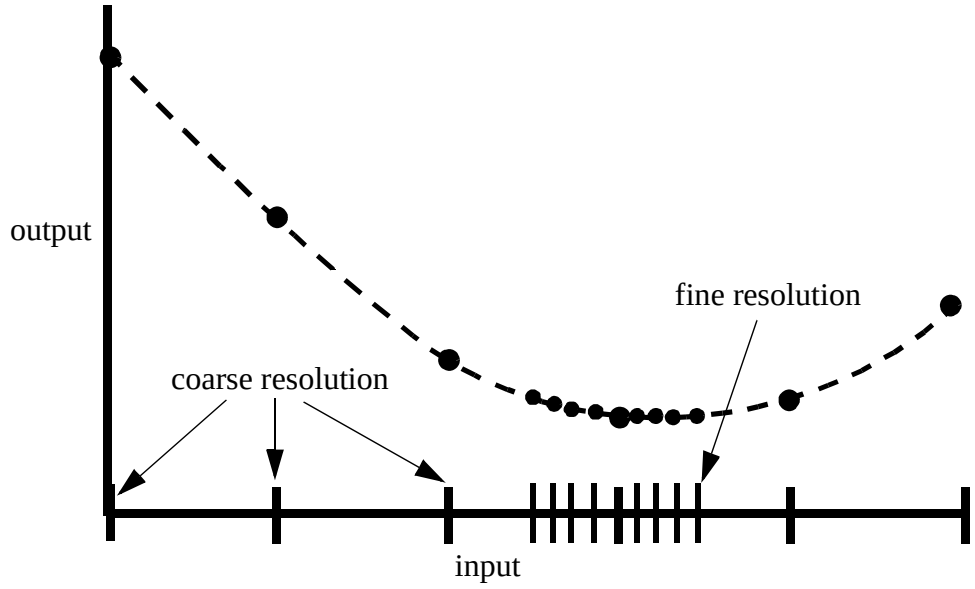


Figure 6.5: Multi-resolution search technique.

For example, consider the contour plot of the time scores for the Example Task previously shown in Figure 5.8 of Chapter 5. The black dots represent the experiences that have been collected so far. The dashed rectangle represents the current bounds of the search space. No actions that are outside of the rectangle are considered during the search for the best action.

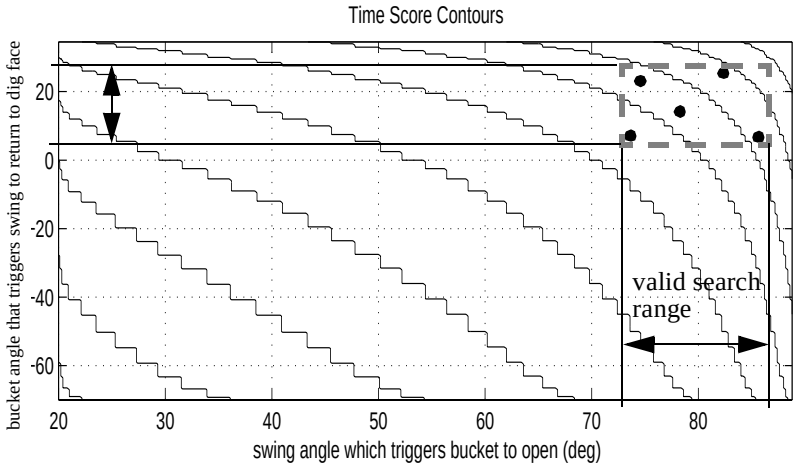


Figure 6.6: Valid experience data base search ranges for the actions are defined by existing experiences.

This may appear as if the search is forever trapped in this box. However, extrapolation is allowed in a controlled manner. If the data base search routine is convinced that the best action lies on the boundary of the rectangle, then a small step in the appropriate external direction is taken. This cautiously expands the boundaries of search space. Unfortunately, just like the velocity, position, and error thresholds that we have seen earlier, this *extrapolation step* for each action dimension is another threshold that must be provided to the system and may be dependent on the nature of the task and vehicle characteristics.

The implications of the controlled extrapolation approach addresses the trade-off between aggressive exploration, which may result in a faster increase in task performance, and safety in choosing new, untried actions. Certainly in an application like truck loading, safety is very important. A learning algorithm that slowly but surely improves its performance, but may take a long time to do so, would be more desirable than a learning system that approaches human expert operator performance in only one or two trucks, but has a very high chance of making a mistake in the process. The fact that many buckets are loaded over the course of the task means that very rapid increases in task performance is not the top priority.

6.3.2 Using Error Thresholds

Error thresholds were briefly introduced in Chapter 5 in describing of how the error component of the reward is used to select an action. Error thresholds define limits on the acceptable amount of error that can be tolerated in satisfying a task constraint. In the search for the best action in the experience data base, error thresholds are used to prune the search space as well.

During the searching process, only those actions whose predicted error score is within acceptable limits are considered further. This pruning process occurs at all levels of the multi-resolution brute force search. Among the remaining actions, the one with the predicted minimum time is returned as the best action to take in the current task state.

The choice of error threshold values can have a dramatic effect on the perceived productivity of the task. As shown in Chapter 5, performing the task in minimal time and satisfying the task constraints are conflicting goals. The error thresholds affect which actions can be used and which can not. Error thresholds that are very conservative will slow task execution down, while error thresholds that are too loose may result in dangerous situations. Thus, the choice of these values is crucial to perceived task performance.

One possibility is to have a human supervisor observe the autonomous machine's behavior, and adjust the error thresholds accordingly. For example, if the supervisor feels that the bucket is coming too close to the truck on each loading pass, he or she can tighten the error threshold that affects that part of the motion (the *Boom Up* subtask, for instance). The experimental results chapter explores the effects of changing the errors thresholds on the fly during operation.

6.4 Truck Loading Experiences

6.4.1 Error Thresholds and Extrapolation Steps

There are three separate subtasks in the truck loading system that receive their own error score. This means that there are three error thresholds are required. The error thresholds that were used for the final adaptive motion planning system are shown in Table 6.2.

Subtask	Error threshold
Boom Up	-1°
Dumping Motion	-10°
Boom Down	0°

Table 6.2: Truck Loading Task Error Thresholds

There are seven action parameters in the adaptive motion planning system. Each action requires its own extrapolation step if the search range is to expand. Table 6.3 shows the extrapolation step values that were selected for the final truck loading system.

Action	Step
$a_{bm\text{-}sw}$: boom angle that triggers swing joint to truck	0.5°
$a_{sw\text{-}st_dump}$: swing angle that triggers stick joint to move to first dump location	1°
$a_{sw\text{-}bk}$: swing angle that triggers bucket joint to open	1°
$a_{st\text{-}bk}$: stick angle that triggers bucket joint to open	2°
$a_{bk\text{-}sw}$: bucket angle that triggers swing joint to return to dig face	5°
$a_{sw\text{-}bm}$: swing angle that triggers boom joint to lower to dig face	1°
$a_{sw\text{-}st_dig}$: swing angle that triggers stick joint to move to dig location	1°

Table 6.3: Extrapolation step values for truck loading actions

6.4.2 Implementational Details

Although the multi-resolution brute force search is very simple, for higher dimensional data bases it is not very fast. Although there is adequate time during the current bucket loading pass to search for the results from the previous pass (approximately 15 seconds worth of time), it was decided in actual implementation to defer the data base search to a later time, such as waiting for the next truck to arrive. Therefore, during the loading of each truck, the experiences are still gathered and stored, but the data base does not suggest an alternative action after each bucket load. Rather, it waits until the entire truck is loaded, and then performs the search for better actions for the task states that it had just encountered. There is no fundamental difference in how the algorithm works, just in when it performs its computation. If a suggested action from the experience data base truly is desired after each bucket loading pass, an acceptable solution may be to have a separate data base search module that runs in parallel with the motion planning system on its own processor.

6.5 Discussion

A goal of the adaptive motion planning system is maximum productivity. This implies some sort of optimization. The optimization portion of the system that achieves this goal is performed in the experience data base component. It forms the essence of any reinforcement learning system. A model from actions to rewards is built incrementally as data is gathered. This model is used to find the best action for a given task state.

The most important result that is presented in this chapter is the representation of the inputs and

outputs to the experience data base. These include a state of the world, an action, and a reward. Once the problem has been expressed in this form, there are many choices of prediction and search methods that can be used. Memory-based learning techniques have been selected as the prediction mechanism for this system component. Certainly a neural network, CMAC, or radial basis function could have been used as a prediction too as well, although memory-based learning does hold some advantages for this particular application.

The next chapter describes another way of suggesting an action that proves to be very useful in the early stages of the task when there are no experiences in the experience data base. This other action suggestion method analyzes the vehicle state history after each bucket loading pass to determine a better set of action parameter values.

Chapter 7 Command Shifting

The experience data base described in the previous chapter is a repository for all of the experiences that are gathered by the autonomous system. These experiences are used to find the best action to take for the current task state. However, there are a couple of problems with the experience data base.

For one, it is useless until there are experiences in the data base. Some method is needed to safely populate the data base with initial experiences that are useful in suggesting better actions later. Another problem with the experience data base is the controlled extrapolation. Recall that controlled extrapolation was implemented to avoid suggesting actions that are in an unexplored region of the data base and could possibly be dangerous. By itself, controlled extrapolation as a means of exploration could take a long time to expand the boundaries of candidate actions. It would be helpful if there was some other means to add experiences to the data base so that the boundaries of the valid actions are expanded quickly and safely.

This chapter presents a powerful heuristic, called *command shifting*, that analyzes the vehicle state history from each loading pass. The vehicle state history is freely available, as it is also needed for the reward function. Like the experience data base, the command shifting function also suggests the best action to take in the current task state.

Figure 7.1 shows a continuation of the adaptive motion planning system diagram. The new command shifting component described in this chapter is shown in bold. The input to the command shifting module is the vehicle state history from one loading pass, and the outputs are a new action parameter set and predicted score. This action, along with the suggested action from the experience data base, goes to the *action selector* module, which is described in the next chapter.

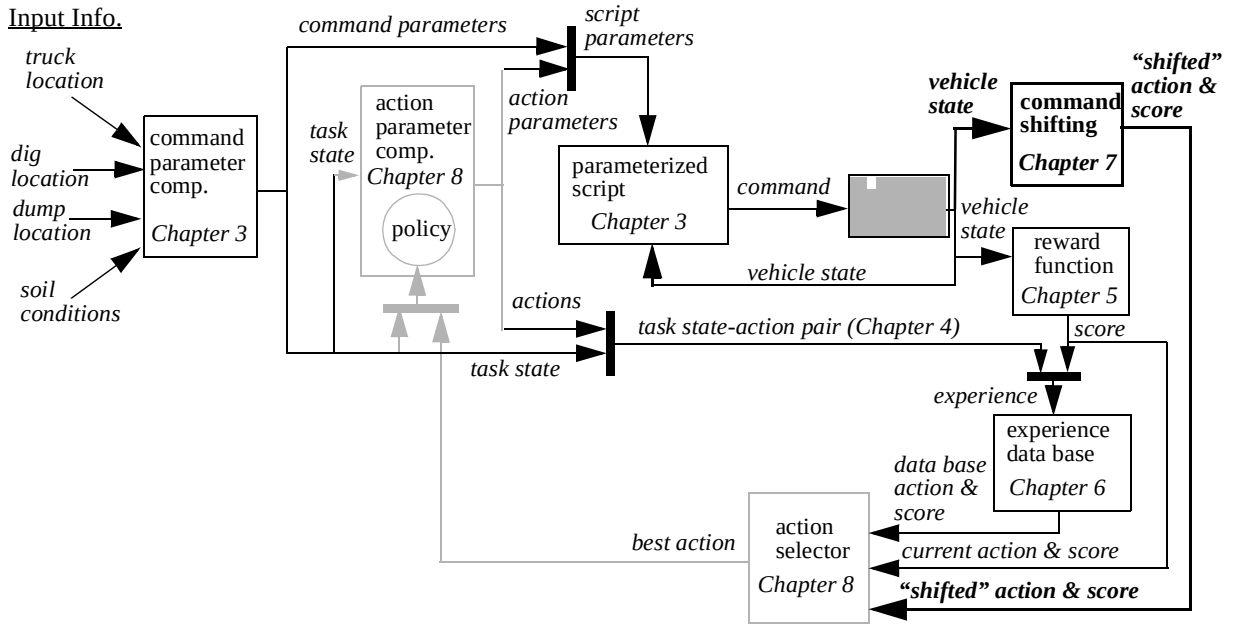


Figure 7.1: Block diagram of the adaptive learning system. In bold are the system components that are described in this chapter.

The next section details the steps of the command shifting function in the context of the Example Task. Section 7.2 presents the information that is required for the command shifting heuristic to operate on joint traces for the truck loading task. Finally, Section 7.3 is a discussion of the advantages and disadvantages of the command shifting heuristic

7.1 Command Shifting

"Command shifting" is indicative of what the command shifting heuristic does to the joint traces of the vehicle state history. The command shifting function shifts the joint traces back and forth along the time axis in order to produce a new vehicle state history that would receive the best score possible in the current task conditions. In doing so, it produces a new coordination of the different joint motions.

In a sense, the command shifting algorithm works in reverse order from the actual script execution. During normal operation, a set of action parameter values are filled in to the parameterized script. The plan is executed resulting in a set of joint traces. The command shifting function starts with a set of joint traces, rearranges them to produce a better vehicle state history, and then uses the new shifted joint traces to back solve for a set of action parameters that will achieve the desired motion.

More specifically, the command shifting function seeks to align pairs of joint traces so that certain

events coincide in time. These events are based on the task constraints that define the desired vehicle behavior.

The command shifting heuristic produces both a new set of action parameter values and a predicted set of scores for the adjusted vehicle motion. An important point is the new set of action parameter values have *not* been executed on the actual machine, it is simply a prediction. Also, the new set of script parameters that are produced by the command shifting heuristic are only valid for the current task state. At first, this may seem to pose a problem since exactly the same task states may never be seen again, but as explained in the next chapter, the new actions can help to find good sets of action parameters for any task state.

There are four main steps to the command shifting function: determining events, shifting the joint traces, finding new values for the action parameters, and predicting a score for the new shifted motion. The following four sections describe each step in the context of the Example Task.

7.1.1 Determining Events

The first requirement for the command shifting heuristic is a set of events that must be detected in the joint traces. These events are similar, and in some cases identical, to the events presented in Section 5.1 of Chapter 5 that are needed for the reward function. Recall that typical events include *start events*, which are the times when a joint begins to move, *goal events*, which are the times when a joint reaches its goal, and *target events*, which are the times when a joint reaches an intermediate target angle during its motion. The specific events that are needed for the command shifting function come from an analysis of the task constraints. Two events come from each task constraint.

Let us return to the Example Task. Recall that the Example Task is to swing to the truck, open the bucket, and swing back to the dig face. For the Example Task, there is one explicit task constraint: the bucket must reach a “deposit” angle, at which all of the material falls out, by the time the swing reaches its goal angle at the truck. The two relevant events that are needed come directly from the task constraint description. The two events are the bucket reaching the “deposit” angle and the swing reaching the desired angle at the truck.

There is also an implied second task constraint that can be used for command shifting purposes. This task constraint was not needed for the reward function, however by defining it, it helps to shorten the total task execution time. The second task constraint specifies that once the bucket does reach its “deposit” angle, the swing can begin to move toward the goal at the dig face. The two events that arise from this new task constraint are the swing beginning to move toward the dig face, and the bucket reaching the “deposit” angle, which is the same event from the first task constraint.

Table 7.1 summarizes the events that are required for the command shifting function. There are three unique events in all, as one is duplicated for the two task constraints.

Task Constraint	Constraint description	Relevant events (event type)
1	Bucket must reach dump angle when swing reaches goal angle at truck	bucket reaches “deposit” angle (target)
		swing reaches angle at truck (goal)
2	Swing must begin to move to dig face when bucket reaches dump angle	swing begins to move to dig face (start)
		bucket reaches dump angle (target)

Table 7.1: Example Task events for each task constraint.

Figure 7.2 shows a set of joint traces that are produced by the Example Task’s parameterized script. The action parameter values came from the high end of the action parameter range resulting in a functional, though slow, task execution cycle. The three events that are needed for the command shifting function have been successfully detected. These are shown by the solid vertical lines in the plots, and the times of each event are given in Table 7.2.

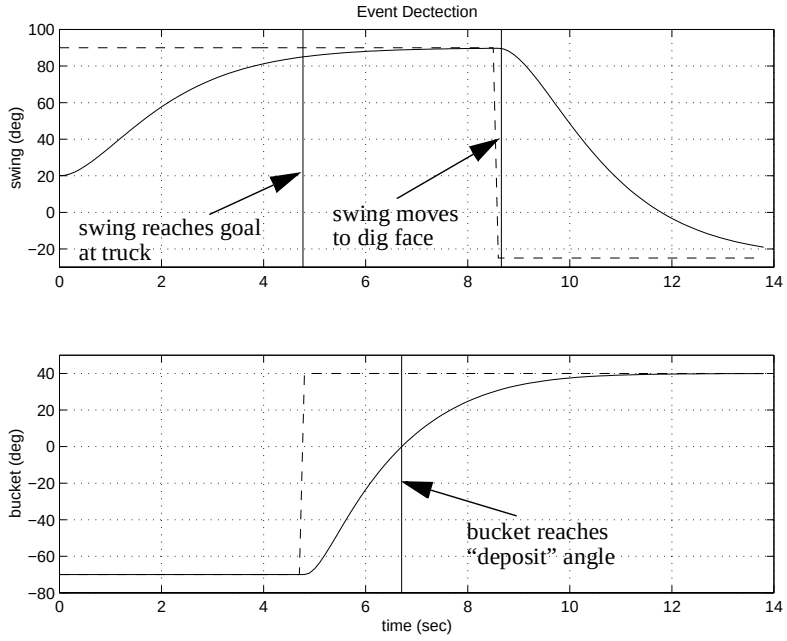


Figure 7.2: The three events that are needed for command shifting.

event	time
swing reaches goal at truck	4.77 sec.
swing starts moving to dig face	8.66 sec.
bucket reaches “deposit” angle	6.71 sec.

Table 7.2: Times of the events for the Example Task joint traces.

7.1.2 Shifting the Traces

The next step in the command shifting heuristic is to shift the joint traces to line up the detected events. The command shifting function seeks to align the events that come from the same task constraint.

For the Example Task, this means that the swing-at-truck event and the bucket-”deposit”-angle event should align, as should the bucket-”deposit”-angle event and the begin-swing-to-dig event. Thus, all three events should occur at the same time.

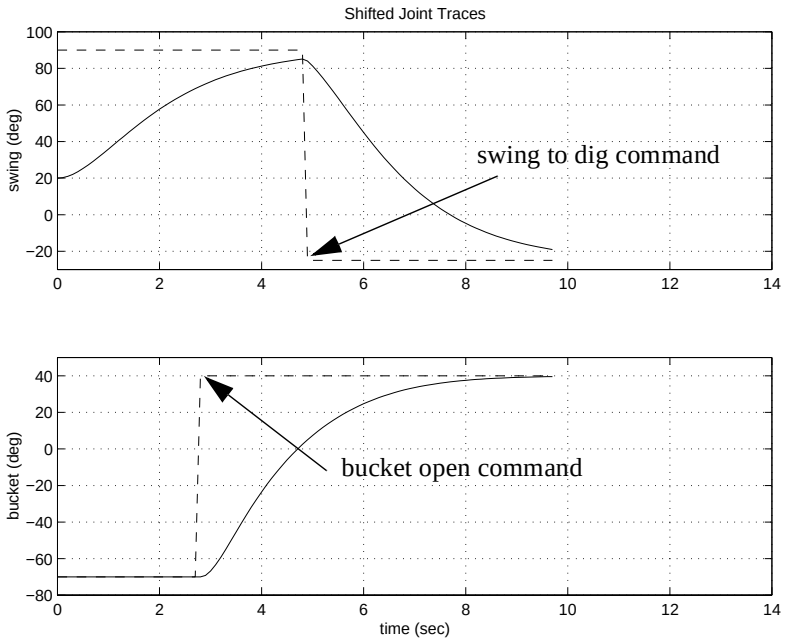


Figure 7.3: Shifted swing and bucket joint traces. The three detected events have been aligned.

Conceptually, shifting the joint traces is equivalent to taking a pair of scissors, cutting the joint traces along the vertical lines that define the events, and gluing the remaining parts of the joint traces back together. Figure 7.3 shows the results of this step of the command shifting function. In the new shifted joint traces, approximately four seconds of time were eliminated from the task

motion by aligning all three events.

7.1.3 Computing New Action Parameter Values

The third stage of the command shifting heuristic uses the shifted joint traces to find a new set of action parameter values. Recall that the action parameters are used in the script rules to determine when to transition between script states, which affects the coordination of the joints during the motion.

For the Example Task, there are two action parameters: the swing angle that triggers the bucket to open $a_{sw\text{-}bk}$, and the bucket angle that triggers the swing to move back to the dig face $a_{bk\text{-}sw}$. The values of these action parameters affect the times that new motion commands are sent to the vehicle's controller. The first step in finding new values of the action parameters is to find the times that the commands are sent in the shifted joint traces. Referring back to Figure 7.3, the bucket-open command is sent at 2.8 seconds and the swing-to-dig command is sent at 4.8 seconds.

Next, the value of the joint that acts as the trigger in the script rule is found from the shifted joint traces at the times that the commands are sent. For example, for the $a_{sw\text{-}bk}$ action parameter, the swing joint determines at what point the bucket is to open. At 2.8 seconds, which is when the bucket-open command is sent in the shifted joint trace, the swing's angular position is $70.27°$. This becomes the new value for the action parameter $a_{sw\text{-}bk}$.

The bucket joint's motion determines at what point the swing is commanded to return to the dig face. At 4.8 seconds, the bucket's angular position is $2.49°$. This becomes the new value for the action parameter $a_{bk\text{-}sw}$.

7.1.3.1 Shifting Rate

The command shifting heuristic is aggressive in choosing new values of the action parameters. It selects the values that it believes will satisfy the task constraints and minimize execution time in one step. In practice, however, it may be desirable to be less aggressive to mitigate the chances that command shifting chooses action parameter values that fall in areas of the action space that are beyond the allowable error thresholds. A *shifting rate* is defined that limits the amount of change from the old action parameter value to the new value. The shifting rate ρ_{shift} is defined to be between 0 (no change) and 1 (most aggressive change).

The shifting rate is used as follows:

$$a = \rho_{shift}(a_{new} - a_{old}) + a_{old}$$

where a_{old} is the previous value of the action parameter and a_{new} is the value from the fully

aggressive command shifting algorithm. If ρ_{shift} was set to 0.5, the final value of each action parameter for the Example Task would be:

$$a_{sw\text{-}bk} = 0.5\ (2.49^\circ - 30.0^\circ) + 30.0^\circ = 16.25^\circ$$
$$a_{bk\text{-}sw} = 0.5\ (70.27^\circ - 85.0^\circ) + 85.0^\circ = 77.64^\circ$$

In this way, the shifting rate allows the action parameter values to approach what are believed to be the best values without possibly jumping past them into an unacceptable region of the action space. The other benefit of the shifting rate is by taking smaller steps, the experience data base is populated with data that can help to map the true shape of the reward function. The disadvantage of the shifting rate is that the rate of task improvement is slowed down. Again, this addresses the trade-off of rapid learning versus safe learning. In an application such as autonomous mass excavation, it is clear that some speed in improving performance can be sacrificed for safe learning.

7.1.4 Predicting the Score

The final step of the command shifting heuristic is to predict the time and accuracy scores for these new, though untried, action parameter values. This is done by sending the shifted joint traces to the reward function module, which was described in Chapter 5, just as if they were real joint traces. Figure 7.4 shows the swing-bucket joint path for the shifted joint traces shown in Figure 7.3. The task constraint point for the current task state is shown by the circle, and the dashed line represents the closest distance from the path to the task constraint point, which is the error component of the score. The predicted time and error scores for the shifted joint traces are:

time = 9.8 sec.
error = 4.9 deg.

The original time and error scores are:

time = 13.6 sec.
error = 0.5 deg.

This results in a time savings of 3.8 seconds while still maintaining acceptable error levels within 5 degrees.

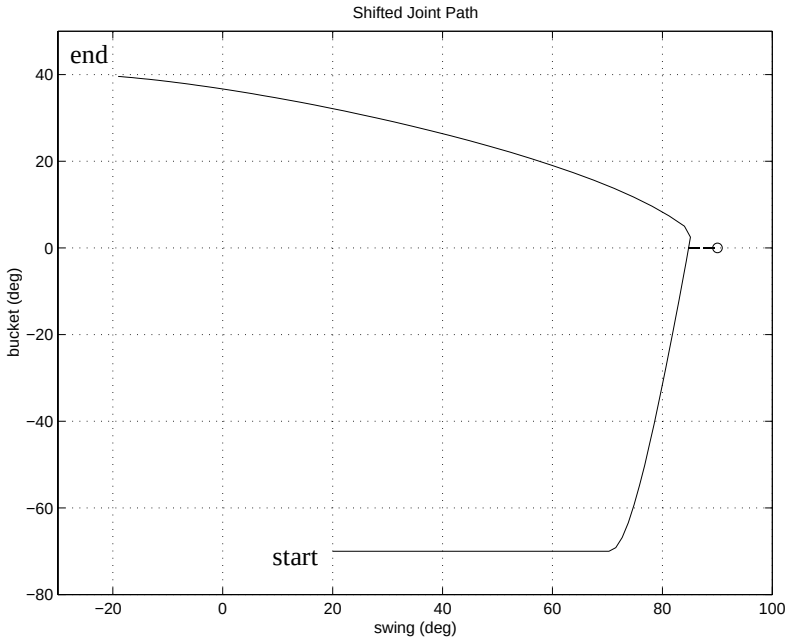
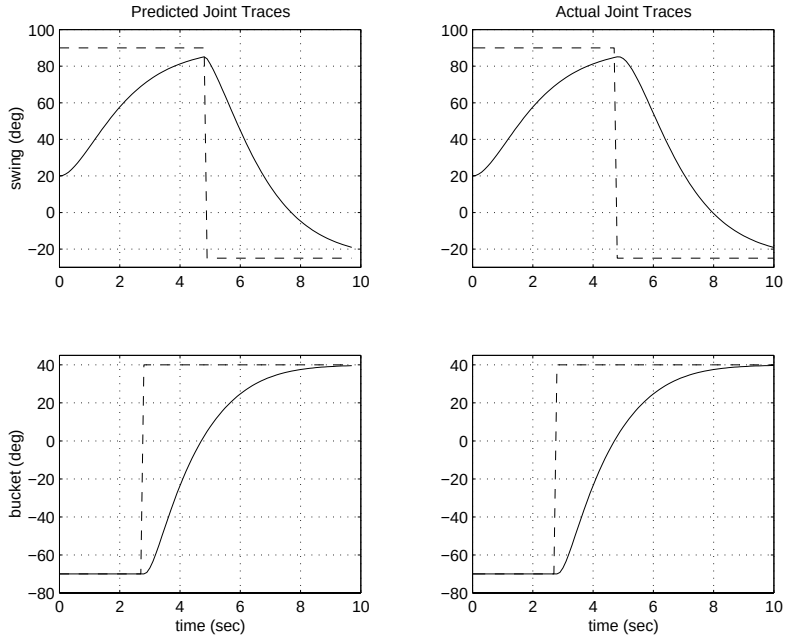


Figure 7.4: Swing/Bucket joint space showing the joint space path of the Example Task motion.



time = 9.8 sec.
accuracy = 4.9 deg.

time =10.0 sec.
accuracy = 4.9 deg.

Figure 7.5: Swing and bucket joint traces for the Example Task. (Left) Predicted joint traces produced by command shifting. (Right) Actual joint traces produced by executing the task motion using the values of the action parameters found by command shifting.

7.1.5 Results

To verify how well the command shifting function did in both finding a better set of action parameter values as well as predicting the score, the shifted action parameter values are filled in to the Example Task's parameterized script, and the motion is executed and evaluated. The plots in Figure 7.5 show a comparison between the predicted and actual joint traces. The joint traces resulting from running the new script parameter values are nearly identical to the predicted joint traces, as are the scores. Minor inaccuracies in time are to be expected as "cutting up" and shifting the joint traces result in motion discontinuities that are smoothed out during the actual vehicle motion.

7.2 Truck Loading Command Shifting

The command shifting function for the truck loading task works the same way as the Example Task, except there are now more task constraints to consider. Recall that for the truck loading task there are six task constraints in all.

7.2.1 Events

Table 7.3 summarizes the events that are necessary for the truck loading command shifting function that arise from the six task constraints.

Task Constraint	Task Constraint Description	Relevant Events (event type)
1	Boom must reach clearance angle before swing reaches truck	boom reaches clearance angle (goal)
		swing reaches truck (target)
2	Stick must reach first dump location before bucket reaches spill angle	stick reaches first dump location (goal)
		bucket reaches spill angle (target)
3	Swing must reach goal at truck before bucket reaches dump angle	swing reaches goal at truck (goal)
		bucket reaches dump angle (target)
4	Swing must begin to move to dig face after bucket reaches dump angle	swing begins to move to dig face (start)
		bucket reaches dump angle (target)
5	Boom begins to lower to dig goal after swing passes truck	boom begins to lower to dig goal (start)
		swing passes truck (target)
6	Stick must reach dig location when swing reaches dig location	stick reaches dig location (goal)
		swing reaches dig location (goal)

Table 7.3: Truck loading events for the command shifting function.

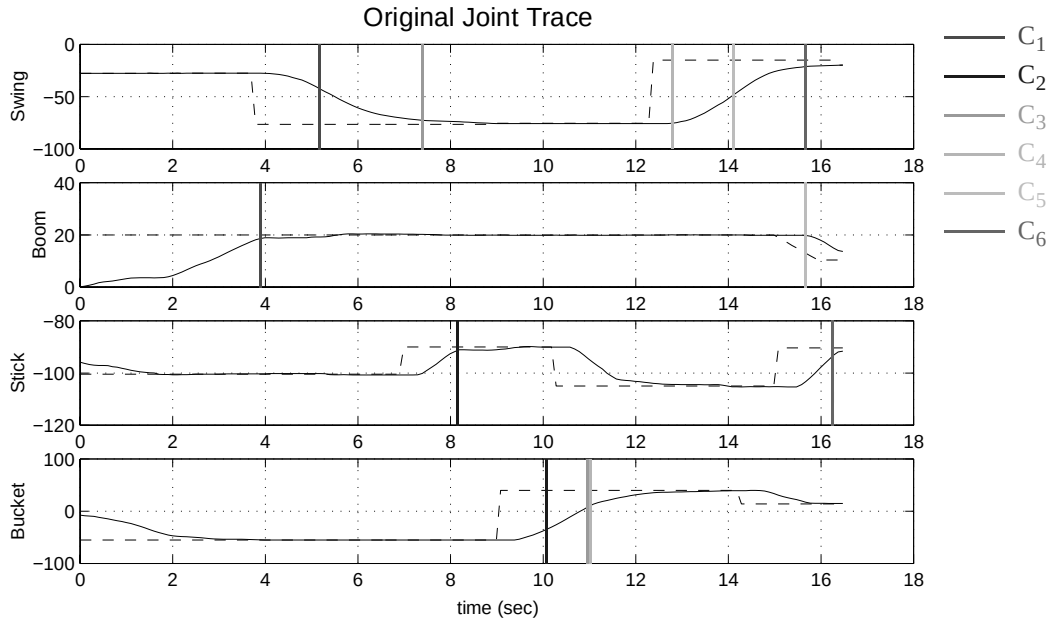


Figure 7.6: Joint traces from an actual truck loading task motion[1]. The 12 events that have been detected are shown as the vertical lines.

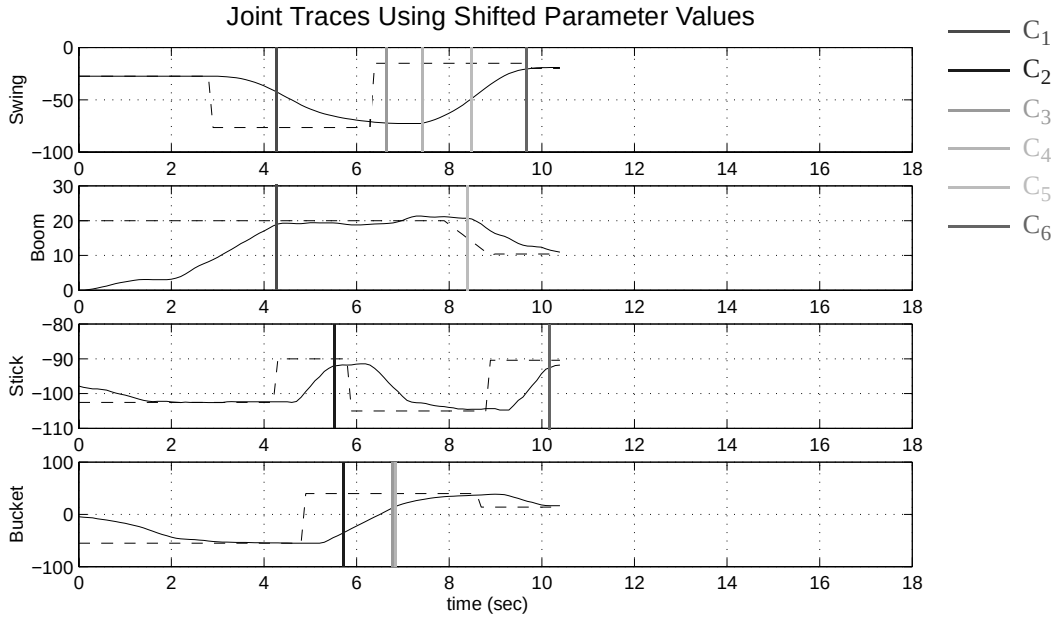


Figure 7.7: Joint traces from an actual truck loading task motion using the action parameter values found from shifting the traces from Figure 7.6.

1. The swing direction is reversed for these tests. Rather than swinging to the left, the excavator swung to the right to deposit the material.

7.2.2 Results

The remaining steps of the command shifting function, shifting the traces, computing the new action parameter values, and predicting a set of scores, is exactly the same as demonstrated by the Example Task. To examine how well the command shifting heuristic does on the truck loading task, an initial functional, though slow, bucket loading cycle was executed. Figure 7.6 shows the joint traces from the loading pass that was executed on the hydraulic excavator testbed. The detected events are shown as the colored vertical lines. Matching colored pairs of vertical lines represent the different task constraints that the command shifting function seeks to align. The numbered legend refers to the task constraint number from Table 7.3.

Figure 7.7 shows the results of the command shifting function. The values of the action parameters that were produced by shifting the joint traces of Figure 7.6 were filled into the truck lading parameterized script and executed. Thus, Figure 7.7 does not show the predicted traces, but the actual traces that were run on the hydraulic excavator using the suggested action parameter values from the command shifting heuristic. A shifting rate of 1.0 was used in this command shifting example. The joint traces of Figure 7.7 shows very good alignment of the task constraints, and a total time savings of 6 seconds, which translates to a 36% increase in task execution speed while still satisfying the task constraints.

7.3 Discussion

The command shifting heuristic is a simple idea that leads to extremely rapid improvement in task performance. The information that it uses about the vehicle state history is freely available, as it is also required to compute a reward. The command shifting heuristic allows the learning system to navigate through the high dimensional space of action parameters and quickly home in on what may be the best set of actions. The command shifting heuristic also helps in initially populating the experience data base with relevant experiences that are in the neighborhood of the best actions to take.

The command shifting heuristic, however, is just that, a heuristic. It makes some big assumptions about the dynamics of the vehicle, which may not be valid in all cases. One assumption that it makes is it can shift the joint traces and still expect the same motion from the vehicle, the only difference being an offset in time. For the majority of the excavator motions that are involved in the truck loading task, this is true. Dynamic coupling effects, such as centripetal or coriolis forces, are minimal and can be neglected. Thus, there is a sense of joint motion independence that makes the command shifting heuristic possible.

The actuator coupling of the excavator's hydraulic system, however, does pose a potential problem for command shifting. Consider the case that involves the motions of a coupled joint pair, the swing and the stick. Recall that these joints share the same hydraulic pump, which is their sole source of power. The task constraint for this motion specifies that both joints are to reach their commanded goal positions at the same time (target-target equality constraint).

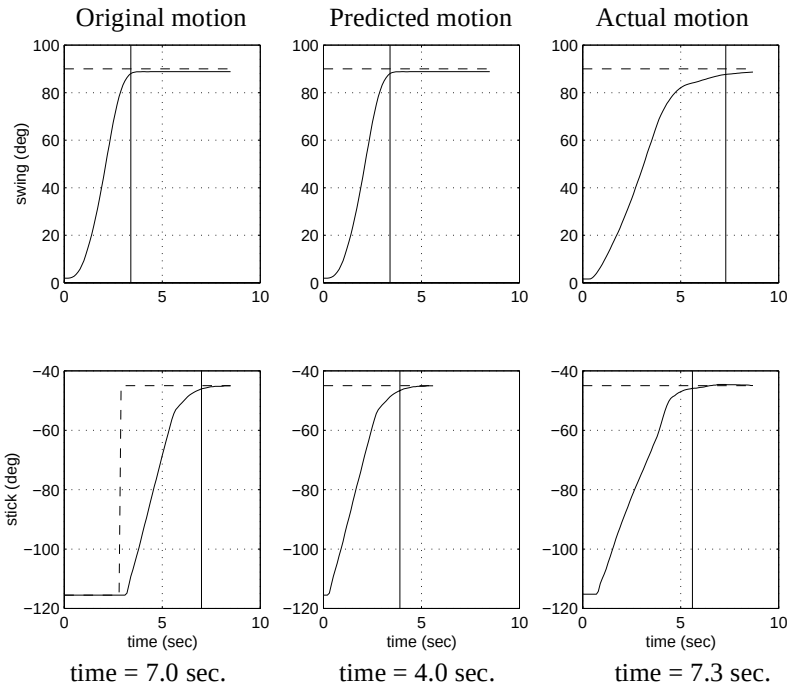


Figure 7.8: Swing and stick joint traces showing problems with command shifting for coupled motion.

Three pairs of plots are shown in Figure 7.8 for this motion example, with the swing joint traces in the top row and the stick joint traces in the bottom row. The first pair of swing/stick plots shows the motions of the two joints when both are run independently. The swing motion starts first followed by the stick motion. The solid vertical lines show the events when both joints reach their respective goals, and the total motion takes 7 seconds to complete.

The second pair of plots shows the results of the command shifting function. The function obviously wants to shift the stick joint trace to the left so that both joint motions run concurrently, and predicts a time of 4 seconds.

The third pair of plots shows what actually happens when this set of action parameter values is run on the excavator. Because of the actuator coupling, the concurrent motion only results in slowing both joints down, and the task is not completed until 7.3 seconds, a worse score than if both joints had moved one at a time.

One solution to this problem is a command shifting monitor. The command shifting monitor would compare the predicted scores from the command shifting heuristic with the actual scores that are produced by running the command shifting's suggested action. During testing and development of the adaptive motion planning system, it was found that the performance was good enough not to warrant the creation of such a monitor, however for motions that violate the independence assumptions, such a monitor would be helpful. It was also discovered that such a moni-

tor was not needed because, after a few experiences had been gathered, the experience data base “took over” and suggested actions which were better than the command shifting function’s suggestions.

The next chapter completes the description of the adaptive motion planning system. It describes how the best action suggestions from the experience data base and the command shifting function are used to find the best actions for any given task state.

Chapter 8 Action Selection and the Policy

With this chapter, the adaptive motion planning system comes full circle. Until now, when describing the components of the adaptive motion planning system, values for the action parameters have simply been given. This chapter describes how the action parameters are computed for each task execution cycle, and how the information that is provided by the *command shifting* function and the *experience data base* is used to select action parameter values for future task execution cycles.

One component that is described in this chapter is the *action selector.* The action selector receives several candidate actions and their predicted scores as input. These action candidates come from the command shifting module described in Chapter 7 and the experience data base described in Chapter 6. The action selector also receives the current action and score as a third candidate action. The action selector decides which of the three candidate actions is best, based on the error and time components of their scores, and sends the winning suggestion to the *policy.*

A policy is a direct mapping from a state to an action, ideally the optimal action to take in the given state. For each task execution cycle, the adaptive motion planning system uses the *policy* to find a set of action parameters for the current task state. The action parameters are then filled into the parameterized script, and the plan is executed.

The policy is similar to the experience data base in several ways. In the experience data base, new experiences are gathered and stored for each bucket loading cycle. Similarly, every new, and hopefully better, action suggestion that comes from the action selector is stored in the policy. Memory-based learning techniques are used to generalize across task states and compute a set of action parameters for any task state. Furthermore, the policy is updated from time to time as more experiences are gathered.

The policy is different from the experience data base in that the experience data base records every experience good or bad, while the policy only stores the best actions to take in a given state. It is not quite correct to say that the policy stores the best experiences because the actions that are in the policy may not have been executed on the real machine yet.

Figure 8.1 shows the completed system block diagram for the adaptive motion planner. The new components presented in this chapter are shown in bold.

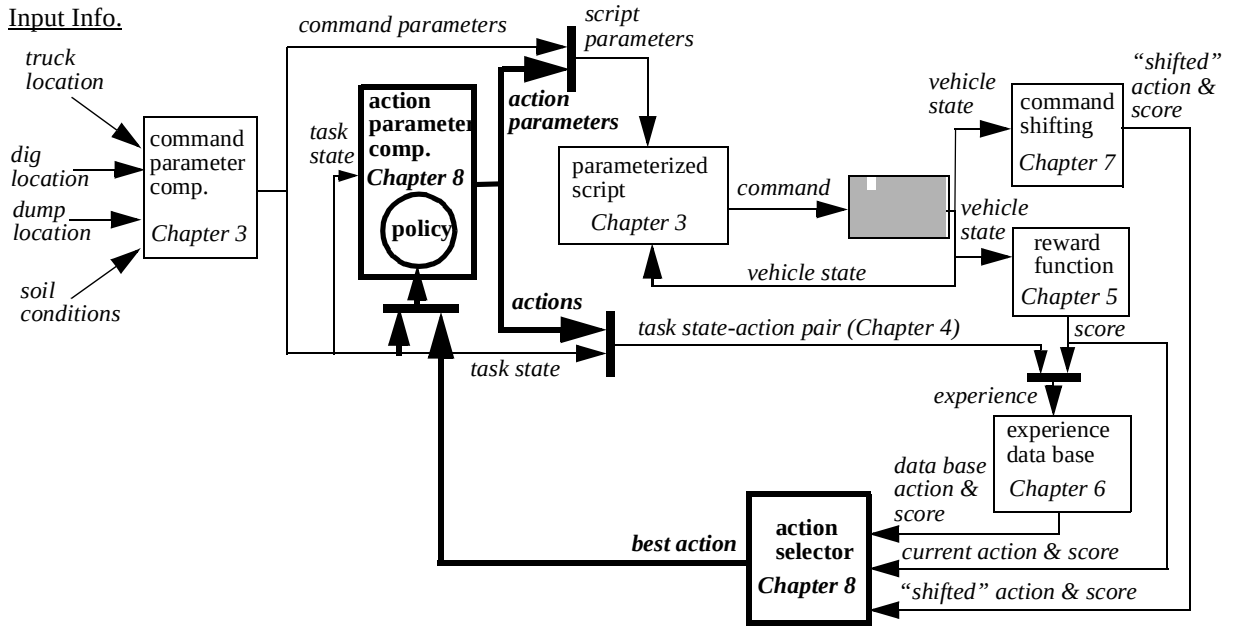


Figure 8.1: Block diagram of the adaptive learning system. In bold are the system components that are described in this chapter.

To summarize the entire system, given information about the worksite and the task conditions, a *task state* is defined. The task state is sent to the *action parameter computation* module which uses the *policy* to find a set of *action parameters* for the current task state. The action and command parameters are filled into the *parameterized script*, and the plan is executed on the vehicle. The corresponding vehicle state history is used to evaluate the plan and assign it a *reward*. The task state, action, and reward, known as an *experience*, is stored in the *experience data base*. The experience data base uses the stored experiences to find the best action to perform in the current task state. The *command shifting* module also suggests an action, which it finds by analyzing vehicle state information. These two action suggestions, along with the current action, are sent to the *action selector,* which chooses the best one. The best action is then sent to the policy where it is stored and used to compute future actions. This cycle continues over the entire lifetime of the task.

The next section describes the action selector module. Section 8.2 describes the policy and also discusses what to do when there is no prior action information. Finally, Section 8.3 describes how

the policy is updated by using the experience data base.

8.1 Action Selector

The action selector module receives any number of suggested actions and scores for the current task state and returns the best action. Currently, there are three inputs to the action selector:

- the current action that was just executed and the actual score,
- the action that is suggested by the experience data base, along with a predicted score,
- and the action that is suggested by the command shifting function, along with a predicted score.

The first stage of the action selection algorithm eliminates any actions that have error scores beyond the acceptable error thresholds. Of the remaining candidate actions, the time component of the score is then used as the selection criterion. The suggested action with the lowest time is considered the best action and sent to the policy.

If all of the action suggestions are beyond the error threshold, then there are two possible remedies. One solution is to simply not return a best action at all to the policy. Unfortunately, this could result in the same poor action being executed over and over again. For example, assume for the moment that an action was selected from the policy that resulted in an error score beyond the error threshold. Let us also assume that the command shifting heuristic and the experience data base are no help in producing actions that are within the error threshold either. Thus, all three options to the action selector are unacceptable. If the action selector does not send an action to the policy, then the same poor action that started the problem remains because the policy is never updated with a better action. If the policy is queried with the same task state, then the same poor action will be returned, since that is all the policy knows.

The other solution, which is the one used in the current system, is to return the action which has the lowest error score of the three and also provide a higher level system warning. This solution may not be very satisfactory, as the error-prone action does influence the action parameter values and may lead to worse actions in the future. However, it may be the case that the controlled extrapolation mechanism of the experience data base eventually moves the action suggestions to a region of the action space that is within the acceptable error limits. Either way, if this situation occurs, it is usually indicative of other problems in the motion planning system, for example beginning the task with a poor action in the first place.

8.2 The Policy

The policy is used to select values for the action parameters of the parameterized script based on the current task state. There is a separate policy for each task state-action space. For the truck loading task, there are four separate policies each returning the appropriate action parameters.

The policy can be considered a look-up table or cache that only stores the best action for a particular task state. There are two advantages to doing this: 1) there is no need for a time consuming search in the experience data base when it is time to compute a set of action parameters and 2) the best actions that are currently stored in the policy can be used to find good actions for previously unseen task states. The second advantage is useful because in a real-world scenario, the same task states may never be seen more than once.

Each and every action suggestion that is received from the action selector module is stored in the policy. This means that there could be entries that are indexed by the same task state. At first, the idea of overwriting the actions for the same task state, or task states that are close together, was considered. However, this requires even more thresholds to determine if task states are close enough. Instead, every new action suggestion from the action selector is stored in the policy, even if it is for exactly the same task state of existing data points. The disadvantage of this approach is that there could be widely varying action suggestions for the same task state. This problem is addressed further in Section 8.3, which discusses updating the policy.

8.2.1 Generalization Across Task States

Like the experience data base, function approximation techniques are used to generalize across different task states. This means a set of actions can be found for any task state even if it has not been seen before. Weighted averaging, or kernel regression, is the function approximation technique that is used in the policy. Kernel regression is described in more detail in Appendix B. Although locally weighted linear regression could also have been used, kernel regression was selected for several reasons. For one, the policy must provide a set of actions for the given task state very quickly. Some of the task state variables are not known until the very beginning of the free motion. Any pause required for computing the action parameters only serves to lower total productivity. Kernel regression is faster than locally weighted linear regression since there is no matrix inversion involved.

Another more subtle reason for selecting kernel regression is the possibility of having several different actions for the same task state, as mentioned earlier. These action suggestions could be very different depending on when they were stored in the policy. For example, early in the task, the actions might not be very good in terms of their time scores. As the task progresses, and new experiences are gathered, the action suggestions for the same task state may improve, but the older worse actions still remain in the policy. Averaging this kind of data, as opposed to linear regression, is a more acceptable solution.

Like the experience data base described in Chapter 6, the policy is not allowed to extrapolate to find actions that are outside the bounds of previously seen task states. Again, the mechanism of controlled extrapolation could be used here, requiring yet more extrapolation step values for the various action parameters.

8.2.2 Default Actions

As in the experience data base, the policy is not useful until action suggestions are stored in it. There must be some way to compute values of initial *default* action parameters. Default action

parameters guarantee functionality of the task, which satisfy the task constraints, but may not result in the fastest task execution cycles.

A way of computing default actions can come directly from the task constraints. For example, consider the *Boom Up* subtask. The task constraint associated with this subtask requires that the boom reach its clearance angle before the swing reaches the truck (see Section 5.5.1 of Chapter 5). Recall that the swing is commanded to move to the truck once the boom passes a certain angular value $a_{bm\text{-}sw,}$ which is the action parameter for this subtask. An obvious default action parameter value for this subtask is the boom clearance angle itself. This guarantees that the swing will not begin moving toward the truck until the boom has raised to its required height, which satisfies the task constraint.

More generally, the values of the default action parameters should be the values of the command parameters for the action parameter's joint. In the above example, the action parameter's joint is the boom, so the default action parameter value is set to the boom's command parameter value, which is the boom clearance angle.

In actual implementation, because of asymptotic effects as a joint reaches its goal, the default action parameter is calculated to be 90% of the distance between start and goal of the action parameter's joint rather than 100%.

Default actions are also used if the function approximation techniques in the policy fail to provide an answer for a given task state. One way this can happen is if the task state is outside the bounds of previously seen task states. In this case, it is best to start with a default action in order to provide at least one safe data point in that region of the policy.

8.3 Policy Updates

At opportune times during the task, for example when waiting for a truck during a mass excavation task, the actions that are stored in the policy are updated with help from the experience data base. This is a simple form of policy iteration, which is prevalent in reinforcement learning (Kaebling et. al., 96).

The policy uses the experience data base to update its entries. For each task state-action data point in the policy, the experience data base is searched to find the best action for the given task state. If the action that is found by the search is better than the action that is currently stored in the policy, the old action is replaced with the new one. This means that the policy must also store the scores (predicted or real) that correspond to each data point in order to perform this comparison.

The search procedure is exactly the same as the one performed in the experience data base at the end of each task execution cycle. The range of valid actions are still bounded by the known actions, and controlled extrapolation is used if the best action lies on the boundary. The actions found by the data base search, of course, must be within acceptable error tolerances.

Why does the policy need to be updated? The answer lies in the trade-offs that were made in order to learn safely. The shifting rate, which controls the rate of change of the action parameters in the command shifting module, and the limits on the range of valid actions in the experience data base, both serve to slow down the overall rate of improvement. As a result, the actions that are considered best and are stored in the policy from the beginning of the task might be very different, and most likely worse, than actions that are stored in the policy later in the task. This point will become clear in the next chapter, which presents an example of the entire adaptive motion planning system in action.

Updating the policy also serves to make those actions similar that are indexed by the same task state. As mentioned earlier, there may be widely varying actions for the same task state in the policy. The policy update routine would return the same action for all instances of the same task state.

The policy updates are done at advantageous times during the task because of the time involved with the search for all of the points in the policy. Another alternative is to have the policy updater run in parallel with the rest of the system replacing actions in the policy. Other possibilities include updating as many policy entries as possible during the digging phase of the truck loading motion, perhaps with some form of prioritization such as oldest entries to most recent entries.

In order to bring the entire adaptive motion planning system together, the next chapter contains an example of the adaptive motion planning system for the Example Task. Chapter 9 also presents the results of the adaptive motion planning system for the target application of loading trucks with a hydraulic excavator.

Chapter 9 Experimental Results

This chapter describes the results of experiments and field tests using the adaptive motion planning system. Like the chapters before it, this chapter is also divided between the Example Task and the truck loading application, presenting results for both. Section 9.1 contains a step-by-step example of the adaptive motion planning system in action for the Example Task. Following this are experiments exploring how the adaptive motion planning system would react given the scenarios of a new, previously unseen task state (Section 9.2), different orderings of task states (Section 9.3) and on-the-fly changes to the error thresholds (Section 9.4).

The next four sections present results of the adaptive motion planning system for the truck loading task. Section 9.5 describes the excavator testbed, both hardware and software, that was used for this research. Section 9.6 presents results from experiments performed on the hydraulic excavator testbed and compares the results to an expert human excavator operator performing the same task. Finally, Section 9.7 shows the results of the adaptive motion planning system in different work site topologies using the simulated excavator.

9.1 Adaptive Motion Planning Example

To get an understanding of how the complete adaptive motion planning system works, this section presents a few steps of the algorithm for the Example Task. There are two different task states involved in this example, one with the swing starting close to the truck and the other with the swing starting farther away from the truck. This example shows how the action parameter values are selected and how the task performance is improved. Subsequent sections demonstrate what the adaptive motion planning system would do in other scenarios for the Example Task including being presented with a previously unseen task state, being presented with the same task states, but

in different orders, and handling on-the-fly changes to the error thresholds, which define the allowable error and the perceived task performance.

9.1.1 Requirements

First, let us summarize all of the required functions, thresholds, and other definitions that must be given to the adaptive motion planning algorithm for a specific task. Cross references to other sections of the document are provided for further information.

- **Parameterized script:** The parameterized script defines the steps of the task, rules to transition from step to step, and a set of parameters that specify the motion goals and motion coordination. Refer to Section 3.3 of Chapter 3 for the Example Task's parameterized script.

- **Command parameters:** Command parameters define the motion goals for each task step. The command parameter computation module transforms information about the task environment into a form that is usable by the parameterized script. For the Example Task, values of the command parameters were simply provided.

- **Task states:** Based on the motion dependencies of the parameterized script, the overall task may be broken down into smaller subtasks. For each subtask, the task state defines the task conditions, such as initial and final joint locations, which are relevant to the subtask. There is only one subtask of the Example Task. Refer to Section 4.1 of Chapter 4 for the task state of the Example Task.

- **Actions:** Actions are the action parameters of the parameterized script. Like task states, different actions are associated with separate subtasks. The main purpose of the adaptive motion planning system is to find actions that result in the best task performance. Refer to Section 4.2 of Chapter 4 for the actions of the Example Task.

- **Start and stop events for evaluating task execution time:** For each subtask, the start and stop events are required for timing purposes. See Section 5.1.1 of Chapter 5 for the start and stop events of the Example Task.

- **Start event velocity thresholds:** The perceptible starting motion of a joint is detected when the joint's velocity surpasses a certain value. A velocity threshold is required for each start event. There is one start event of the Example Task, when the swing begins moving toward the truck, and the velocity threshold is set to 0.

- **Goal event position thresholds:** Goal events are detected by position thresholds that define acceptable margins around desired goals. There is one goal event for the Example Task, when the swing reaches the dig face, which terminates the parameterized script. The position threshold is set to 1°.

- **Task constraints:** Task constraints define the desired behavior of the vehicle for the task and are used to evaluate the accuracy in completing the task. Refer to Section 5.2 of Chapter 5 for the task constraints of the Example Task.

- **Error Thresholds:** Error thresholds define the acceptable error tolerances on the error component of the score and serve to prune the space of possible actions. There is one error threshold required for the Example Task's task constraint, which is set to $5°$.

- **Extrapolation steps:** Extrapolation steps define the allowable distance into unexplored areas of the experience data base. Each action dimension requires its own extrapolation step value. For the two actions of the Example Task, $a_{sw\text{-}bk}$ is set to $1°$, and $a_{bk\text{-}sw}$ is set to $5°$.

- **Shifting rate:** The shifting rate affects the rate of change of the action parameter values in the command shifting function. For the Example Task, the shifting rate is 0.5.

- **Default actions:** In the face of no prior actions in the policy, a method to compute default actions for each action parameter is required. For the Example Task, the $a_{sw\text{-}bk}$ default action is the swing joint's value 90% of the distance between its initial location and its desired goal location at the truck. For the $a_{bk\text{-}sw}$ action, the default is 90% of the distance between the initial and final bucket location for the bucket-open motion.

- **Weighting factors for function approximators:** Both types of locally weighted function approximators that are used require scaling factors. These can be found automatically by cross-validation after several data points have been collected in both the experience data base and the policy. However, in the beginning, initial scaling factors must be provided. For the experience data base, the scaling factor is 1, which is the most global. For the kernel regression of the policy, the scaling factor is very small, which makes the function approximation approach nearest neighbor.

Now that all of the required functions and threshold values have been given, this section walks through the algorithm step by step for several task execution cycles of the Example Task. Two different task states are used, but only one dimension of the task state is altered. The task state dimension that changes is the initial swing angle. The task begins with no prior information in either the experience data base or the policy.

9.1.2 Task Execution Cycle 1

Step 1: Given information about the worksite, compute the values of the task state.

For the Example Task, these values are simply provided. Table 9.1 shows the task state values for the first task execution cycle.

symbol	description	value
$init\theta_{sw}$	initial angle of swing at dig face	20°
$init\theta_{bk}$	initial angle of bucket at dig face	-70°
$dump\theta_{sw}$	desired swing angle at truck	90°
$dump\theta_{bk}$	angle of bucket at which material falls out	0°
$open\theta_{bk}$	angle of bucket to open to (could be the same value as $dump\theta_{bk}$)	40°
$dig\theta_{sw}$	desired final swing angle at dig face	20°

Table 9.1: Values of Variables for Task State 1

Step 2: Compute a set of action parameters.

The task state is used to index the policy. At this point, however, there is no action information in the policy, therefore default actions are returned. The equations below show the default action computation for the two actions of the Example Task.

$$a_{sw-bk} = 0.9(90° - 20°) + 20° = 83°$$
$$a_{bk-sw} = 0.9(40° - ((-70)°)) + (-70)° = 29°$$

Step 3: Fill the command and action parameters into the parameterized script, execute the plan and evaluate. Store the experience in the experience data base.

The time and error scores for the above task state-action pair are shown below. As expected, the error score is well within the given error threshold of 5°, but the time score can be improved.

Actual Score
time = 13.0 sec.
error = 0.89 deg.

The experience data base how has one entry:

	task state	action	reward
Data Base Entry 1:	(20, -70, 90, 0, 40, -20:	83.0, 29.0 ⟶	13.0, 0.89)

Step 4: Search among the valid actions in the experience data base for the best action to take in the current task state.

In this case, the experience data base search fails because there is only one entry. No action suggestion is sent to the action selector.

Step 5: Execute the command shifting function on the vehicle state history to produce another action suggestion.

The command shifting function returns the following action parameter values and predicted score.

Command Shifting

$a_{sw\text{-}bk} = 79.2^{\circ}$

$a_{bk\text{-}sw} = 14.0^{\circ}$

Predicted Score

time = 11.7 sec.

error = 1.72 deg.

Step 6: Select the best action among the suggestions and store in the policy.

Of the two valid action suggestions (command shifting and the actual action), the command shifting suggestion is better. This action suggestion is stored in the policy for the current task state as shown. Notice that the predicted score is also stored. This is needed later for policy updates.

	task state	action	*reward*
Policy Entry 1:	(20, -70, 90, 0, 40, -20 ⟶	79.2, 14.0:	*11.7, 1.72*)

9.1.3 Task Execution Cycle 2

The second task execution cycle makes one change to the task state, the initial location of the swing. The new initial location of the swing is -20°. All other task state values remain the same. The following summarizes the output of the algorithm steps.

Step 1: Task state = (-20, -70, 90, 0, 40 20).

Step 2: This new task state is outside the bounds of previous task states in the policy. Therefore, a default action is computed. Action = (79, 29).

Step 3: The cycle is executed, evaluated, and stored. Time = 13.1 sec. Error = 1.93°

	task state	**action**	**reward**
Data Base Entry 1:	(20, -70, 90, 0, 40, -20:	83.0, 29.0 ⟶	13.0, 0.89)
Data Base Entry 2:	(-20, -70, 90, 0, 40, -20:	79.0, 29.0 ⟶	13.1, 1.93)

Step 4: There is not enough data in the experience data base to generate an accurate prediction yet, so this is not an option.

Step 5: The command shifting function produces the following results.

Command Shifting

$a_{sw\text{-}bk} = 77.0^{\circ}$

$a_{bk\text{-}sw} = 13.9^{\circ}$

Predicted Score

time = 12.1 sec.

error = 2.14 deg.

Step 6: The action selector again selects the command shifted option and stores it in the policy.

	task state	**action**	***reward***
Policy Entry 1:	(20, -70, 90, 0, 40, -20 ⟶	79.2, 14.0:	*11.7, 1.72*)
Policy Entry 2:	(-20, -70, 90, 0, 40, -20 ⟶	77.0, 13.9:	*12.1, 2.14*)

9.1.4 Task Execution Cycles 3 and 4

Step 1: The task state for the third execution cycle is the same as the one for the first cycle (20°), and the task state for the fourth cycle is the same as the second cycle (-20°).

Step 2: There are now actions in the policy for these task states. For the third cycle, the action that is returned by the policy is (79.2, 14.0). For the fourth cycle, the action is (77.0, 13.9). These are the action parameter values that were found by the command shifting function.

Step 3: The results for the third and fourth execution cycles are shown below as the new experience data base entries. Up to 1.5 seconds of time have been eliminated already while still remaining within acceptable error limits.

	task state	action	reward
Data Base Entry 1:	(20, -70, 90, 0, 40, -20:	83.0, 29.0 →	13.0, 0.89)
Data Base Entry 2:	(-20, -70, 90, 0, 40, -20:	79.0, 29.0 →	13.1, 1.93)
Data Base Entry 3:	(20, -70, 90, 0, 40, -20:	79.2, 14.0 →	11.5, 1.96)
Data Base Entry 4:	(-20, -70, 90, 0, 40, -20:	77.0, 13.9 →	11.9, 2.41)

Step 4: Although there needs to be 9 data points in the experience data base before the matrices are full rank, we will make an exception for the purposes of the example. Since there is only one variable task state dimension, and two action dimensions, this means there only needs to be four data points. After entry 4 is placed in the data base, the experience data base can search and offer an action suggestion.

The linear models which are fit to the experience data for each component of the score are:

$$time = (-0.02)ts_1 + 0.17a_{sw-bk} + 0.06a_{bk-sw} - 2.36$$

$$error = 0.01ts_1 - 0.33a_{sw-bk} + 0.01a_{bk-sw} + 27.57$$

where ts_1 is the first task state variable, and $a_{\text{sw-bk}}$ and $a_{\text{bk-sw}}$ are the two action parameters.

The action search range is limited to $a_{\text{sw-bk}}$: {77 - 83} and $a_{\text{bk-sw}}$: {13.9 - 29}. The multi-resolution search of the experience data base predicts that an action of (77, 13.9) will produce the best score. This action is on the boundary of the search region, therefore controlled extrapolation is used to take a small step outside of the action boundaries. The new best action becomes (76, 8.9) and the predicted scores for this action are (time = 11.4, error = 2.68).
Step 5: The command shifting function produces the following results for the two cycles.

Cycle 3	Cycle 4
$a_{sw\text{-}bk} = 77.2^{\circ}$	$a_{sw\text{-}bk} = 76.0^{\circ}$
$a_{bk\text{-}sw} = 6.3^{\circ}$	$a_{bk\text{-}sw} = 6.2^{\circ}$
time = 11.2 sec.	time = 11.6 sec.
error = 2.16 deg.	error = 2.39 deg.

For the fourth task execution cycle, the command shifting suggestion is worse than the experience data base. Which prediction is better is a result of the choice of shifting rate for the command shifting function and the extrapolation steps for the experience data base search. Both affect how current parameter values are changed. However, both action suggestions are improvements over the current action.

Step 6: The action selector chooses the appropriate action suggestions and stores them in the policy. The policy now looks like:

	task state	**action**	***reward***
Policy Entry 1:	(20, -70, 90, 0, 40, -20	⟶ 79.2, 14.0:	*11.7, 1.72*)
Policy Entry 2:	(-20, -70, 90, 0, 40, -20	⟶ 77.0, 13.9:	*12.1, 2.14*)
Policy Entry 3:	(20, -70, 90, 0, 40, -20	⟶ 77.2, 6.3:	*11.2, 2.16*)
Policy Entry 4:	(-20, -70, 90, 0, 40, -20	⟶ 76.0, 8.9:	*11.4, 2.68*)

9.1.5 Policy Updates

At this point, notice that there are two different action suggestions for the same task states. For example, policy entries 1 and 3 are both action suggestions for the first task state, but the suggested actions are (79.2, 14.0) and (77.2, 6.3) respectively. If the policy is queried for an action for the first task state, the policy would average the two actions together and produce an answer of (78.2, 10.15), which would not produce the best result based on what the system has already learned.

Alternatively the policy can be updated. For the policy update procedure, the task state for each policy entry is used as an input to the experience data base search. The experience data base is searched for the best action at that task state. If the data base's answer is better than the score of the action that is currently stored in the policy, then the action is replaced. In this way, actions for similar task states can be made the same value.

Suppose that we did update the four existing entries in the policy. For both task states, the experience data base determines that the best action lies on the current boundary of the search space at

(77, 13.9). With controlled extrapolation, the final suggested action for both task states is (76, 8.9). The fact that the same action is selected for both task states may seem odd, but recall that the experience data base is only allowed to search across the range of actions that it currently knows. Since there are not very many experiences in the data base, the actions have not yet settled on the optimal ones, which would most likely be different for the two different task states. For now, the actions are the same because they are pushing the search space boundary in the direction toward the optimal actions.

For the first task state and the action of (76, 8.9), the experience data base predicts a score of (10.7, 2.95), and for the second task state and the same action of (76, 8.9), the score is predicted to be (11.4, 2.68). The entries in the policy are updated appropriately, and the same action parameter values appear for the same task states.

	task state		**action**	**reward**
Policy Entry 1:	(20, -70, 90, 0, 40, -20	⟶	76.0, 8.9:	*10.7, 2.95*)
Policy Entry 2:	(-20, -70, 90, 0, 40, -20	⟶	76.0, 8.9:	*11.4, 2.68*)
Policy Entry 3:	(20, -70, 90, 0, 40, -20	⟶	76.0, 8.9:	*10.7, 2.95*)
Policy Entry 4:	(-20, -70, 90, 0, 40, -20	⟶	76.0, 8.9:	*11.4, 2.68*)

For the remainder of the example, the policy will be updated after every two trials. This effectively makes the policy have only two entries: an action for the first task state and an action for the second task state. In practice, however, this may not be the case since the values of “similar” task states may never be exactly the same.

9.1.6 Results

The adaptive motion planning algorithm was run for several more iterations. Figure 9.1 displays the time and error scores for all of the task execution cycles. The left column is for the first task state and the second shows the results for the second task state. Overall, the results are good, and the task execution time is improved by approximately three seconds from the initial default actions. Notice, however, that as the algorithm explored its space of actions, some actions resulted in error scores that were beyond the acceptable error threshold. Fortunately, the adaptive motion planning algorithm was able to compensate as it gathered more data and select actions which brought the error scores back within acceptable limits.

Figure 9.2 shows the sequence of the action parameter values that were selected for each task execution cycle. Starting from the default action, the command shifting function provides a “kick start” in the right direction of the action space. After that, the search through the experience data base, primarily using the exploration mechanism of controlled extrapolation, further pushed the actions to the area of action space that contained the best ones. The algorithm then began to dwell in the lower left hand corner of the plots as it continued to gather more experiences.

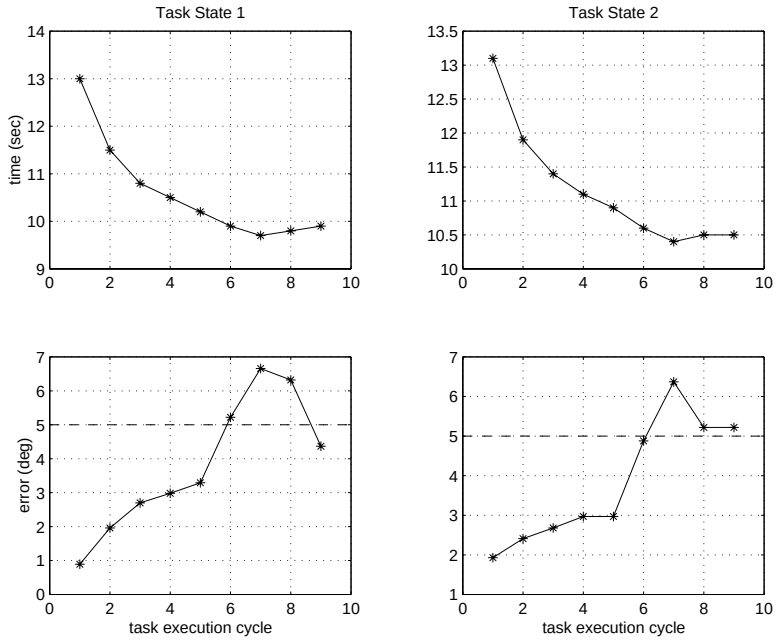


Figure 9.1: Time and error scores for the two separate task states.

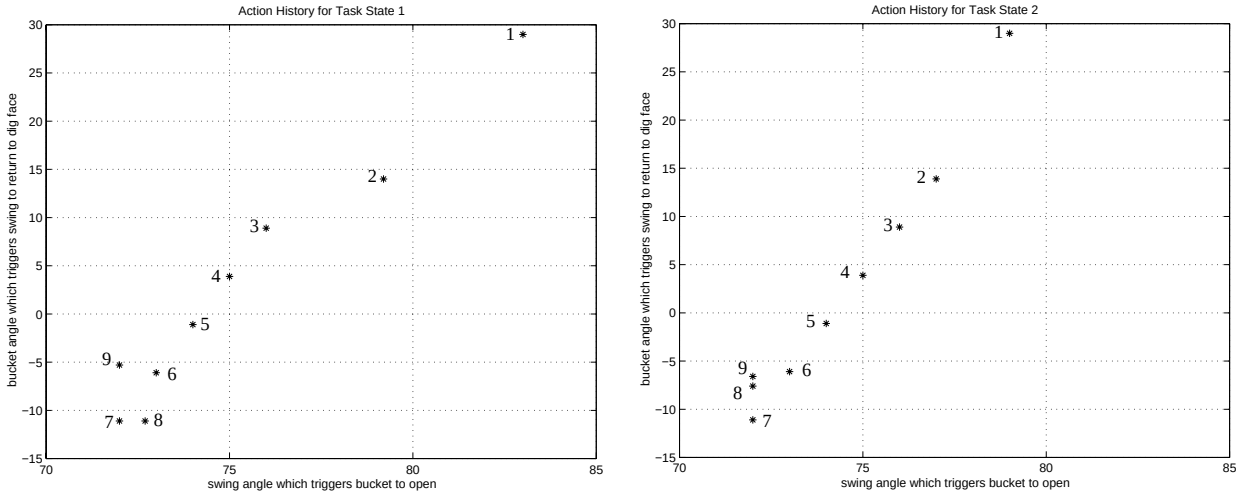


Figure 9.2: Action parameter values for each task execution cycle.

9.2 Example Task: New Task State

This section, and the following two, present some experiments that explore the behavior of the adaptive motion planning algorithm. This first experiment continues where the example from the previous section left off and presents the adaptive motion planning algorithm with a new task

state. The new task state value for the initial swing angle is 0°, which is halfway between the previous task states of 20° and -20°.

After the previous 18 trials, the policy looks like[1]:

	task state		action	*reward*
Policy Entry 1:	(20, -70, 90, 0, 40, -20	⟶	72.0, -6.9:	*9.8, 5.00*)
Policy Entry 2:	(-20, -70, 90, 0, 40, -20	⟶	72.0, -6.6:	*10.4, 5.00*)

The policy must now generalize across the action suggestions of the known task states in order to produce an action for the previously unseen task state of 0°. Recall from Chapter 8 that the generalization function is weighted averaging or kernel regression. There are two extreme cases for computing the action parameter values. The two cases are a function of how much influence, or weight, existing policy entries have on the new task state.

On one extreme, there is pure nearest neighbor, which is the method that was used for the Example Task algorithm. For this case, there is no existing entry in the policy for a task state of 0°, so the action selection algorithm computes a default action for this task state.

On the other extreme, policy entries do exert influence on the value of the actions for the new task state. The amount of influence depends on their distance from the new task state to existing task states. Since the new task state of 0° falls directly in the middle of the two prior task states of -20° and 20°, both existing entries in the policy have equal influence on the new actions.

Let us compare the results from the two methods.

Nearest Neighbor	Global Average
$a_{sw\text{-}bk} = 81$	$a_{sw\text{-}bk} = 72$
$a_{bk\text{-}sw} = 29$	$a_{bk\text{-}sw} = -6.75$
time = 13.1 sec.	time = 10.2 sec.
error = 1.41 deg.	error = 5.34 deg.

As expected, the nearest neighbor option, which used a default action, resulted in a slow, though safe, task cycle. The actions that were found by weighted averaging, however, had much better task execution time score, but were a little beyond the acceptable error threshold of 5°.

1. Redundant policy entries for the same task states have been removed.

It is unclear which technique is better. In the case of selecting a default action for every unseen task state, this does guarantee that there will be a safe data point in that region of the space, but wastes a task execution cycle acquiring it. In the case of weighted averaging, the good actions that have already been found for existing task states can be used to find good actions for unknown task states right away, but at the risk of possible poor task constraint error scores because the true reward function is not known for the new task state.

9.3 Example Task: Interpolation vs. Extrapolation

The next experiment involves the behavior of the policy when given different sequences of the same task state values. The order in which task states are experienced can have a profound effect on the overall task performance.

For instance, suppose that it is the beginning of a new task execution cycle and the task state values have been computed. The policy is now queried for an action to take in the given task state. If the values of the current task state fall outside of the range of previously seen task states values in the policy, then the adaptive motion planning system will use a default action, since it is not allowed to extrapolate beyond the bounds of its known experiences.

If the current task state values fall within the bounds of the known task states, then one of two things would happen depending on which policy generalization strategy is selected. In the case of nearest neighbor, the adaptive motion planning system would most likely select a default action each time, unless the current task state is exactly the same as an existing task state. In the case of weighted averaging, the action that is computed by the policy for the new task state would probably be better than a default action, since it is using information about the current best actions in the policy.

Therefore, for best overall task performance, the minimum and maximum values for each task state variable should be presented to the adaptive motion planning system first so it can use those initial experiences to interpolate actions for subsequent task states. For the truck loading task, for example, it would be ideal to dig the initial buckets of soil from the outer edges of the digging region and then dig in the middle. Similarly, it would be best to dump the material in the front of the truck and the rear of the truck before dumping it in the middle of the truck. The absolute worst case scenario would be experiencing the task state that is in the middle of its range and working from that point outwards in task state space. In that case, all task states would fall outside the realm of previously experienced task states, and the system would constantly use slow, default actions.

This experiment examines just how much of an improvement in performance would be achieved if the adaptive motion planning system did have control over the order of task states. For this experiment, one task state variable from the Example Task, the angle of the bucket at which the material completely falls out, is changed. The other task state values remain constant. In a practical sense, imagine a sensor that could detect the soil conditions and provide the value of this task state variable for every bucket load. A dry, non-cohesive bucket of material may result in a shal-

lower bucket angle, where a wet, cohesive bucket of material might require the bucket to be opened farther in order to allow the material to escape.

This experiment consists of two tests. For each test the order of the task state values is changed. For the first test, the task state values are presented in such a way that no new task state falls within the range of previously encountered task states. Thus, the system is forced to select a default action for each task execution cycle. The second test involves the same task state values, but now presented in such a way that the system is able to interpolate as much as possible. The only default actions are the first two, which define the boundaries of the task state value's range in the first place. For each test, the experience data base and policy begin empty, and 9 task execution cycles of the Example Task are performed.

Table 9.2 shows the task execution times for both tests. Obviously, the total time for the interpolation test is faster than the extrapolation test, simply due to the ordering of the task state values. However, because the policy uses actions that have not been performed on the vehicle to find new actions, the average error for the interpolation test is slightly higher than for the default actions of the extrapolation test.

	Extrapolation (default)	Interpolation
Total task time (sec.)	125.1	99.6
Average error score (deg.)	3.15	6.23

Table 9.2: Task execution times and error scores for the extrapolation and interpolation tests.

Figure 9.3 show the values of the actions that were selected for both tests plotted against the different values of the task state. The order in which the task states were presented to the adaptive motion planning system is also shown. The left hand column is the extrapolation test. As expected, all of the action values are the same, as they are the default actions. The right hand column is the interpolation test. Notice the actions on the boundaries are default actions, but they are the only ones as the policy is able to use existing information to compute different action parameter values.

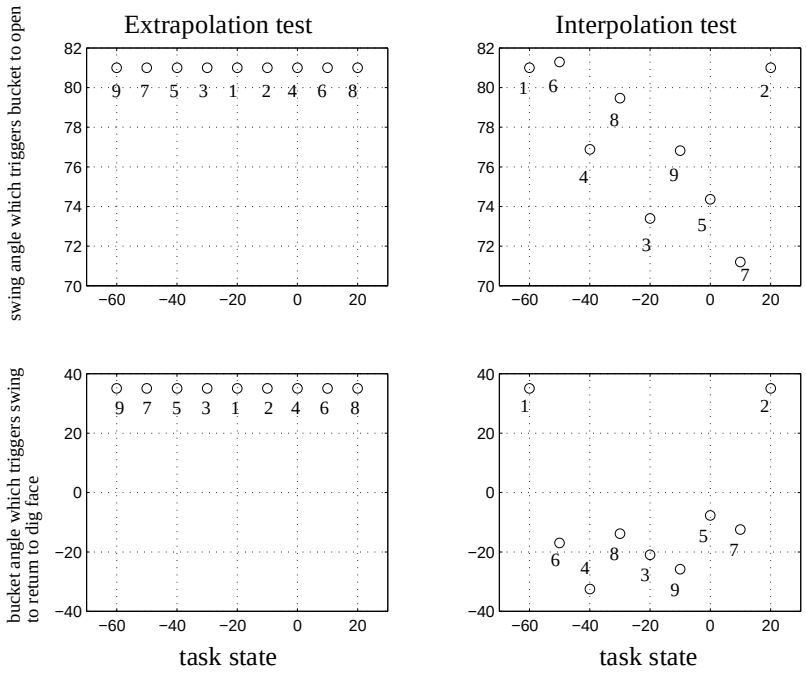


Figure 9.3: Values of the action parameters for different task state values. (Left) The values for the extrapolation test are all default actions. (Right) The values for the interpolation test change because previous actions influence future actions.

9.4 Example Task: Changing Error Threshold

Chapters 5 and 6 explained that the values of the error thresholds can have a dramatic effect on the perceived performance of the vehicle. Selecting error threshold values that are too tight might result in disappointingly slow, though highly accurate, performance. Error threshold values that are too loose may produce an automated excavator performing very fast, though very sloppy loading passes.

A possible remedy to this situation is to have an external observer watch the excavator's progress and adjust the values of these error thresholds on the fly. This experiment explores how the adaptive motion planning system reacts to changing error thresholds.

For the Example Task, there is one error threshold. It defines the acceptable error of the swing joint at its goal over the truck when the bucket deposits its material. For the example from Section 9.1, this was set to $5°$. This experiment involves changing the value of the error threshold, first to $10°$, and then down to $1°$. For sake of example, the task state values remain constant, with the swing starting at $20°$ for each new trial.

The experience data base and policy will begin with the information that was collected during the

example in Section 9.1. The value of the error threshold was 5°, and the actions that are currently stored in the policy were found with that error threshold value. The error threshold value is then changed to 10° and the system is run for 9 more trials. Finally, the error threshold value is changed again to 1°, and the system continues to run for 9 more trials.

Figure 9.4 shows a continuation of the time and error plots from the left hand column of Figure 9.1. Trials 10 through 18 are the new trials with the error threshold value of 10°, and trials 19 through 27 show what happens when the error threshold value changes to 1°.

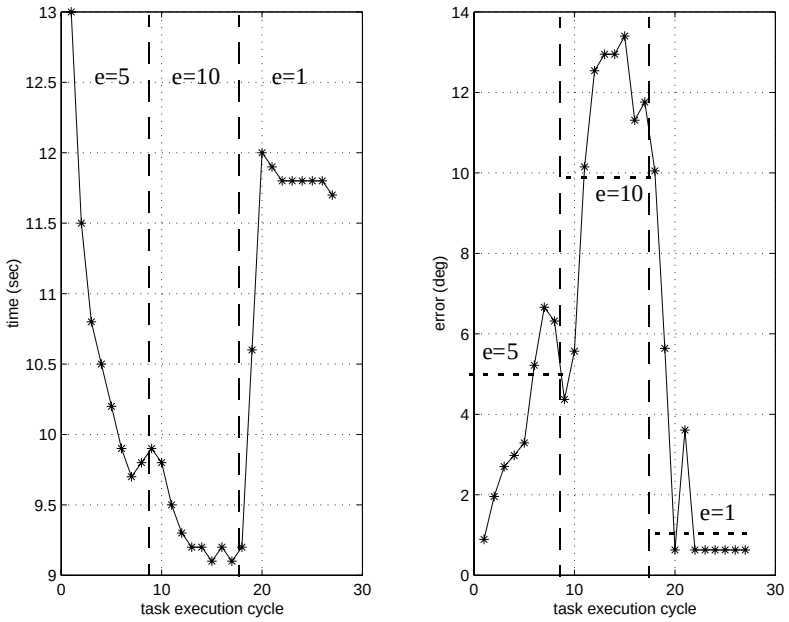


Figure 9.4: Time and error scores from the Example Task algorithm showing the effects of adjusting the error threshold value.

Figure 9.5 shows the progression through the space of actions. Asterisks show the original 9 trials with an error threshold value of 5°, circles are the next 9 trials with an error threshold value of 10°, and pluses are the final 9 trials with an error threshold value of 1°.

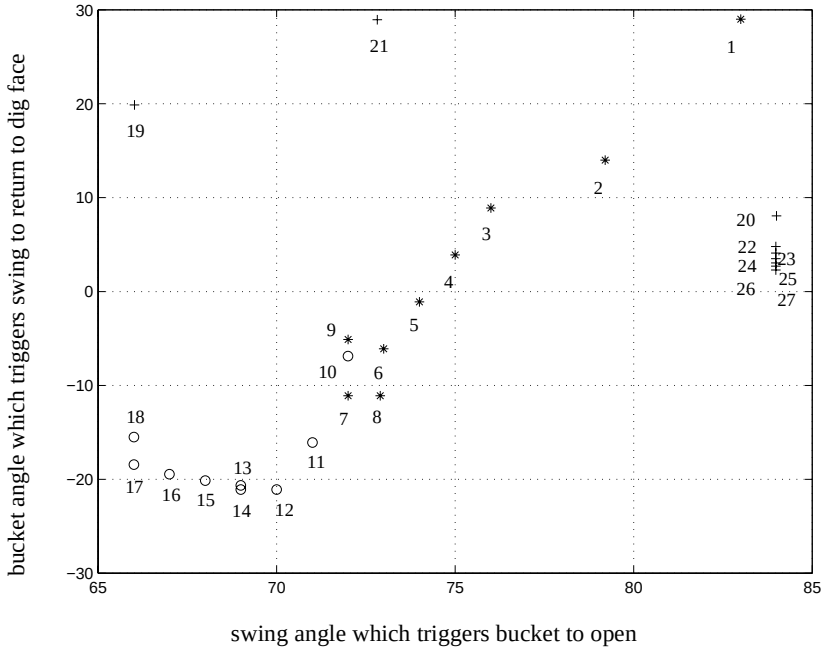


Figure 9.5: Action parameter values for the three different error thresholds.

The changing of the error threshold causes a change in the action parameter search range in the experience data base. New values of the action parameters are found. The system quickly converges to actions that produce the minimum task execution time and are on the boundary of acceptable task error.

For the change from 5° to 10°, the experience data base is now allowed to expand its search region of actions. As a result, controlled extrapolation takes over, the action parameter values are further pushed down and to the left in the plot shown in Figure 9.5. When the error threshold is tightened down to 1°, the boundaries are brought in, but the system does not need to extrapolate. However, there are not many experiences in the data base that have resulted in an error score of 1°, so the system must gather a few new experiences before the actions settle down to the 1° acceptable error tolerance.

The next sections present results of truck loading experiments using a real hydraulic excavator. The experiments include loading several trucks in a row, extended digging tests, and tests performed in different worksite topologies using a simulated excavator. Comparisons against human operator performance are also presented.

9.5 Autonomous Loading System

The research that has been presented here is part of the Autonomous Loading System (ALS) project. The ALS project was a four year robotics project at Carnegie Mellon University's Robotics Institute. The goal was to develop core technologies for automating the mass excavation process. This included perception sensor hardware and software algorithms, vehicle motion planning algorithms, and vehicle control systems as well as a variety of development aids such as simulation and visualization tools. This section describes both the hardware and software components that were developed for the ALS project and that supported this work.

9.5.1 Hardware and Testsite

Figure 9.6 shows the hydraulic excavator that was used to develop and test the adaptive motion planning system. The excavator weighs 25 tons and the bucket has a capacity of 1.5 m^3. The excavator is equipped with joint resolvers, pressure sensors in the hydraulic cylinders, and a PD controller for each joint. All computation is performed onboard the excavator on MIPS processor boards running the vxWorks operating system. A wireless ethernet connection transmits status information back to a workstation for display purposes.



Figure 9.6: Hydraulic excavator testbed.

The excavator is also equipped with two range sensors which are not directly used in this research,

but provide the capabilities to image the truck and surrounding terrain for the purposes of determining where to dig and dump the material on each loading cycle.

The test worksite was constructed to emulate a mass excavation scenario. The excavator sits atop a concrete block approximately 10 feet (3 meters) off the ground. An on-highway dump truck (not shown in the picture) can park to the left side of the excavator for loading. In front of the excavator is a large hillside of material that can be dug with the bucket. The material that is excavated is soft clay-like soil. The clay content in the soil makes its properties change depending on the weather and time of year. In the summer, the soil is very dry and dusty. In the fall and spring, or after a rainstorm, the soil is soupy mud. In the winter, it is usually frozen.

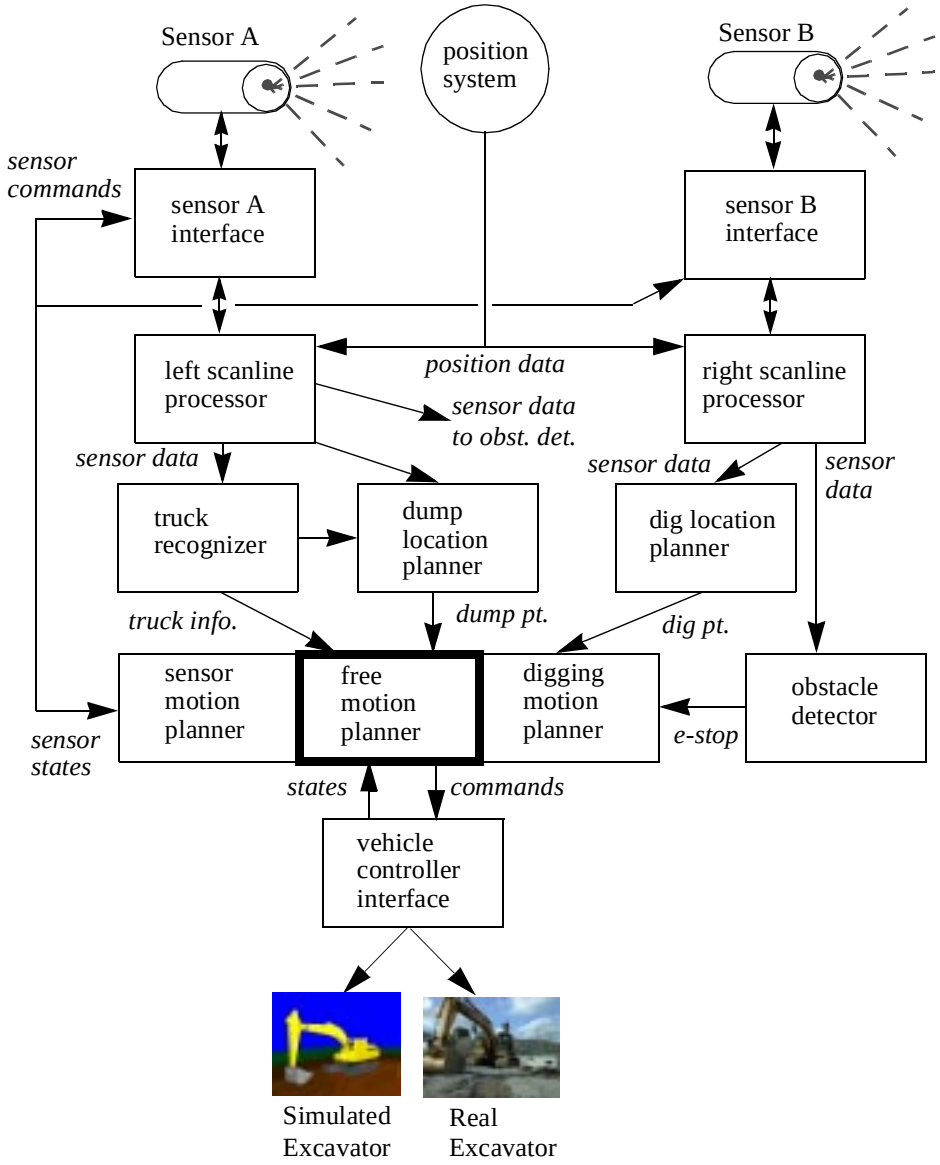


Figure 9.7: Autonomous Loading System software architecture.

9.5.2 Software

Figure 9.7 shows a block diagram of the software architecture that was designed and implemented for the ALS project. Each rectangle represent a separate piece of software that runs concurrently during operation. The one exception is the large rectangle at the bottom of the figure that encompasses the sensor motion planner, dig motion planner, and free motion planner. These are separate functional pieces that exist inside of one software executable.

There are three distinct sections of the software architecture: interfaces, perception, and planning.

9.5.3 Interfaces

The *interfaces* are the outer wrapper of the architecture that connect all of the hardware components to the software. There are interfaces for the vehicle controller as well as each perception sensor. Commands are sent to the vehicle and sensor hardware, and information such as vehicle states or sensor data is received and converted to a standard format that is used throughout the software planning modules. As shown by the vehicle controller interface module, the interfaces also act as a switch between the simulated and real hardware systems. This simplifies the system and allows the larger software planning modules to run in either simulation or on the real testbed without the need for major differences in the code.

9.5.4 Perception Software

The next section of the software architecture deals with the perceptual tasks. As sensor data are received from the sensors, the data points are converted to coordinates in a common global reference frame. This process requires information about the excavator's position and any other degrees of freedom between the sensor and the global reference frame, such as the swing joint. The task of rectifying the sensor data is performed by the *scanline processors*. A packet of sensor data is referred to as a *scanline*. Scanlines are processed by the scanline processor one at a time as they are received from the sensor interfaces. The converted sensor data are then made available to the software modules that require it.

The software modules that require data from the perception sensors are known as the *perception data consumers*. There are four perception data consumers as shown in the architecture. Each of these software modules performs a very specific perceptual task that is needed for autonomous mass excavation. They are similar in that they each receive sensor data from the appropriate region of interest of the workspace and send information about the excavator's environment to the motion planning module.

The *truck recognizer* (Lay, 98) is tasked with sensing, measuring, and locating the truck to be loaded in the excavator's workspace. It receives sensor data from the general region of the workspace where the truck is expected to be, and returns the coordinates of the four top corners of the truck bed, the truck dimensions, and the truck's position and orientation relative to the excavator. In the current system implementation, the truck is recognized only once, upon the arrival of each new truck, however it could be re-recognized on each bucket loading pass in case the truck or excavator had shifted their positions slightly.

The *dump location planner* receives sensor data in the area of the truck in order to plan where the next load of soil should be placed. This software module also requires the information about the truck location from the truck recognizer module so that it can filter out sensor data that are outside the truck bed. Sensor data are received after each load is deposited in the truck, and the coordinates of the next desired dump location are sent to the motion planner.

The *dig location planner* (Singh and Cannon, 98) receives sensor information from the digging region and plans the next location to dig a bucket of soil. Like the dump location planner, the dig location planner also receives this sensor information after each bucket of soil has been dug. The dig location planner does not return a 3-D coordinate to the motion planner. Rather, because it also must take terrain shape and the angle of attack of the bucket teeth into account, it sends the desired joint angles to the motion planner.

The *obstacle detector* (Leger et. al., 98) has a two-tiered approach to preventing possible collisions with unexpected objects, such as other construction vehicles, that may enter the excavator's workspace. A short range obstacle detector receives sensor information for some distance in front of the excavator's implements in its direction of motion. It determines if the area is still clear and decides if the implements will collide with any sensed object. If this is the case, an emergency stop signal is sent to the motion planner. The second level of obstacle detection is long range sensing. Sensor data are received over the natural course of the sensor's and the vehicle's motion. The long range obstacle detector's region of interest is the area beyond the excavator's immediate reach. It monitors this area for abrupt changes in the sensed terrain elevation, which may indicate the presence of an approaching object, and also sends an emergency stop command if warranted.

9.5.5 Motion Planners

The third section of the software architecture is the suite of motion planning modules. There are three distinct motions that must be planned: 1) motion of the vehicle during digging, 2) the free motion of the vehicle which is the topic of this document, and 3) the motions of the sensors to direct their fields of view. This functionality is tightly coupled, which is why there is a single motion planning module.

The sensor motion planner decides where to aim the sensor. This is done by panning the sensor about an axis that is parallel to the excavator's swing axis. The swing's rotational motion acts as a second pan table, so the combined swing and sensor pan angles determine the sensor's orientation with respect to a fixed reference frame. The sensor motion planner also has the responsibility of informing the perception data consumers when to start and stop receiving sensor data. The sensor motion planner does this because it knows precisely which part of the loading cycle motion the excavator is currently executing.

The dig motion planner (Rocke, 94) is a fuzzy logic based algorithm which uses cylinder pressures to measure the forces on the bucket and determines what to do next with a set of pre-existing look-up tables.

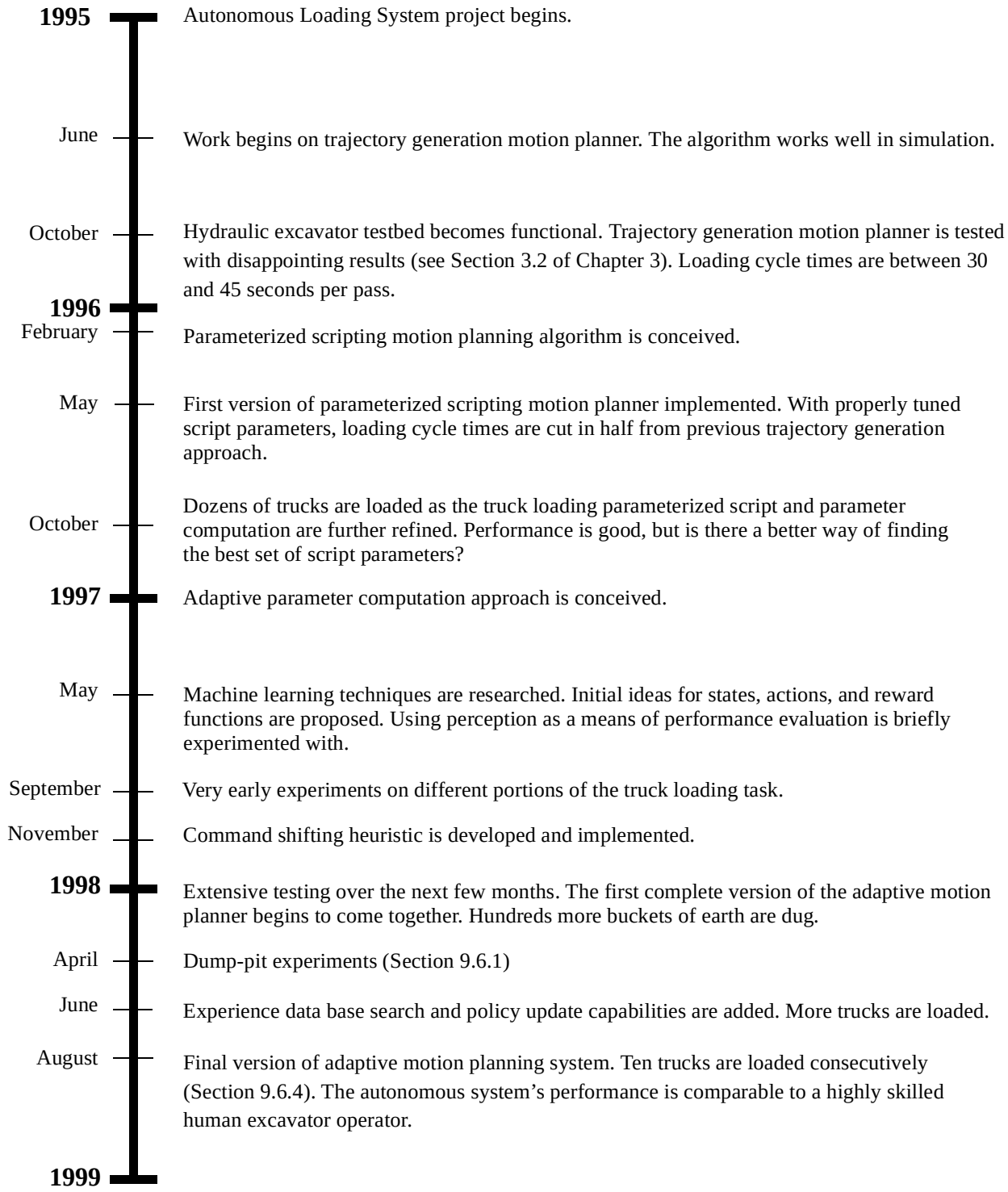


Figure 9.8: Time line showing the evolution of the adaptive motion planning system.

9.6 Experimental Results

This section highlights some of the key results that were obtained during experimentation of the adaptive motion planning system on the hydraulic excavator testbed.

9.6.1 Adaptive Motion Planning System Timeline

The free motion planner for the Autonomous Loading System has evolved over the course of nearly four years of work. During that time, hundreds of trucks have been successfully loaded and thousands of buckets of soil have been dug. Figure 9.8 displays a timeline of the evolution of the free motion planner noting certain key events during the process.

9.6.2 Dump-Pit Experiment

This section describes some of the results that were obtained using early versions of the adaptive motion planning system. The results of these early tests confirmed that the idea of using information about past task performance can improve future task performance. It also helped to identify problems that led to improvements to the final adaptive motion planning system.

The excavator's task for this experiment was to dig several dozen buckets of soil from fourteen dig locations and deposit them at a single location over a large pit in the ground. The dig and dump locations were provided to the system beforehand. Dumping the buckets in the pit allowed the excavator to load many buckets continuously without the need to stop and unload the truck. The single dump location also simplified the task states because many task state values remained constant. Figure 9.9 show the excavator in different stages of the task.



Figure 9.9: (Left) Excavator digging a bucket of soil. (Right) Excavator dumping the bucket of soil in the pit.

There were several differences between this early version of the adaptive motion planning system and the final system that has been presented.

- This version did not search through the experience data base as another means of selecting an action. Rather, it solely relied on the command shifting heuristic function.

- The command shifting was much more aggressive with a shift rate of 1.

- Each of the seven action parameters constituted their own separate action space rather than being coupled together, as some action parameters are in the final system. Also, some of the task states variables were slightly different. Some task state variables were later considered irrelevant and removed.

Figure 9.10 shows the results of this version of the adaptive motion planning system performing the given task. Total accumulated time versus number of buckets is plotted. Forty-five buckets were dug from the bench and deposited in the pit. As a baseline, the dash-dot line shows the results of using default actions only. In this case, no optimization is taking place and the same actions are taken over and over again. The solid line shows the results of the adaptive motion planning system. Notice that for the first five or six buckets, the adaptive system's curve follows that of the default action's curve, but then begins to break away as better actions are found and performance is improved.

Two human test subjects were also asked to perform the same test, but only for approximately thirty buckets. Their results are shown as the two dashed lines.

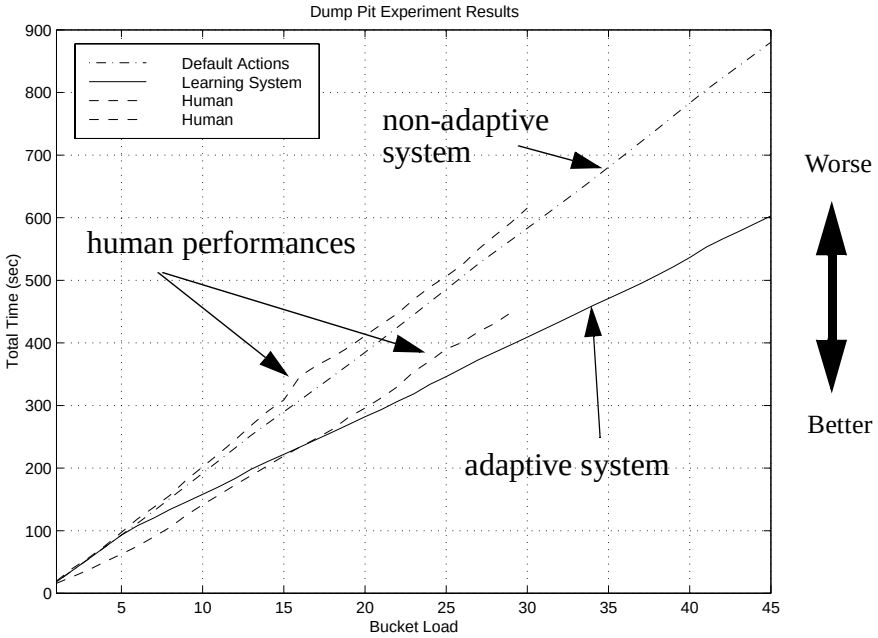


Figure 9.10: Plot comparing automated system performance with human performance for the dump pit task.

9.6.3 Discussion

These early results demonstrated that a motion planning system which can modify its actions based on gathered experiences could allow the excavator to do as well as or better than a human

operator[1]. It is sometimes difficult, however, to compare performance. Different soil conditions and digging styles could affect the overall task cycle times. The accuracy of the human performance in satisfying the task constraints could not be measured very well other than by qualitative means. Observations from these tests showed that the humans could not compete with the accuracy that the autonomous excavator could achieve.As for the task cycle times, one test subject mentioned that, while he was able to stay on pace with, or even out-perform, the automated excavator for a short period of time, there would be no way he could keep that up over the course of an entire day. It would be like running a marathon at a sprinter's pace. In fact, Figure 9.10 shows that one human subject does begin slightly faster than the adaptive system, however after about fifteen buckets, his performance starts to degrade. The automated excavator's performance, when factored over an entire day, would out-perform any human expert operator.

9.6.4 Truck Loading

This section describes the results of the final version of the learning system loading real dump trucks with the real excavator testbed. The excavator's task was to load buckets of soil from six pre-specified dig locations and deposit them in a dump truck that was parked on the left side of the excavator. After each truck was loaded with the six buckets, the truck was unloaded, the soil was placed back where it had been dug, and the truck was parked as close to the same location as possible. The excavator's tracks were not moved. Ten trucks were loaded in a row for this experiment. Figure 9.11 shows the excavator in various stages of the truck loading task.



Figure 9.11: (left) Excavator after digging a bucket of soil and moving towards the truck. (right) The truck is loaded with six buckets of soil.

The same six dig locations and the same dumping pattern were used for each truck. The first bucket was dumped in the middle of the truck bed, the second bucket in the front third of the truck bed, and the third bucket in the back third of the truck bed. The fourth, fifth, and sixth buckets fol-

1. The test subjects in this experiment are members of the Autonomous Loading System project team. While both do have some experience with excavation machinery, they are not professional excavator operators.

lowed the same pattern: middle, front, back. The primary reason for doing this was to present the same task states to the adaptive motion planning system. This way, the improvement in task performance could be studied for a small number of task states without having to perform hundreds of loading passes.

In practice, because each test is a destructive one, the conditions are never exactly the same for each truck. Figure 9.12 shows how repeatable the tests were. The actual dig locations are shown for each of the six nominal dig locations. Four of the ten truck locations that represent the most diverse parking locations are also shown.

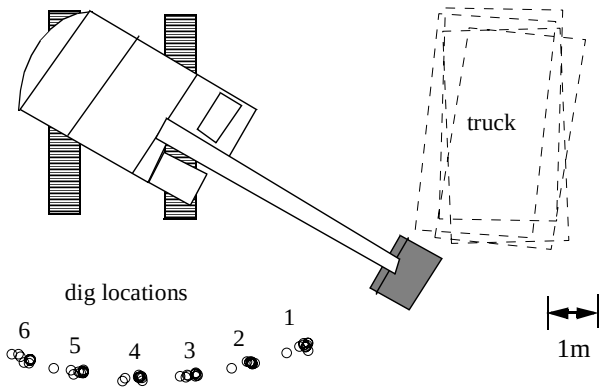


Figure 9.12: Diagram showing the different dig locations and truck parking locations for the truck loading experiments.

9.6.4.1 Task Execution Time

The right-hand plot in Figure 9.13 shows the total truck loading times for each of the ten trucks. The plot on the left displays the times to load each truck with the times for the digging half of the total motion removed. The adaptive motion planning system only adjusted the free motion for each bucket loading pass. Notice that around the third or fourth truck, the graph begins to level off to the final truck loading time. Minor differences of a second or two can be accounted for by slightly different digging times, and different dig and truck locations.

The total productivity increase is 38% for the total truck loading times and 56% when only considering the free motion[1] over the initial default actions.

1. Productivity percentages were calculated by dividing the initial time by the final time and subtracting 1. Thus, 38% increase in productivity means the adaptive system can load 138 trucks in the time it takes the default non-adaptive system to load 100 trucks.

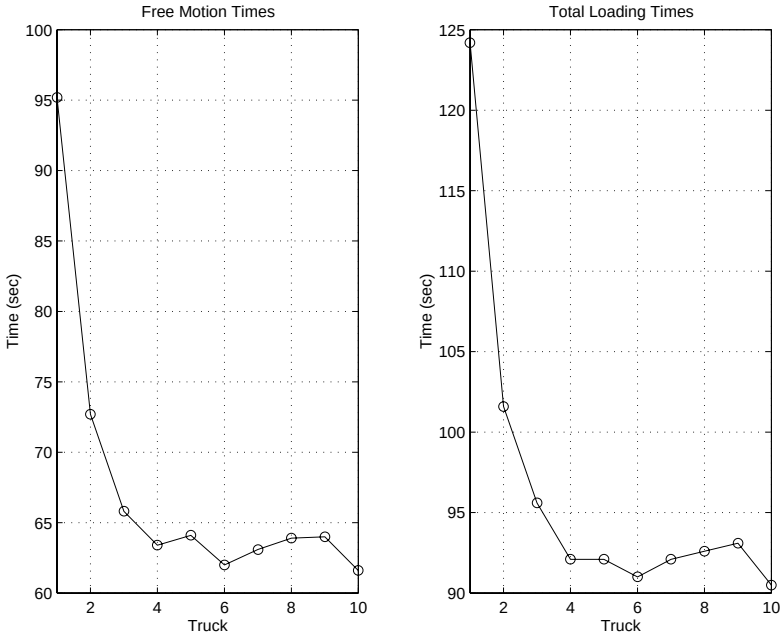


Figure 9.13: (left) Free motion time for each truck. (right) Total truck loading time including dig times for each truck.

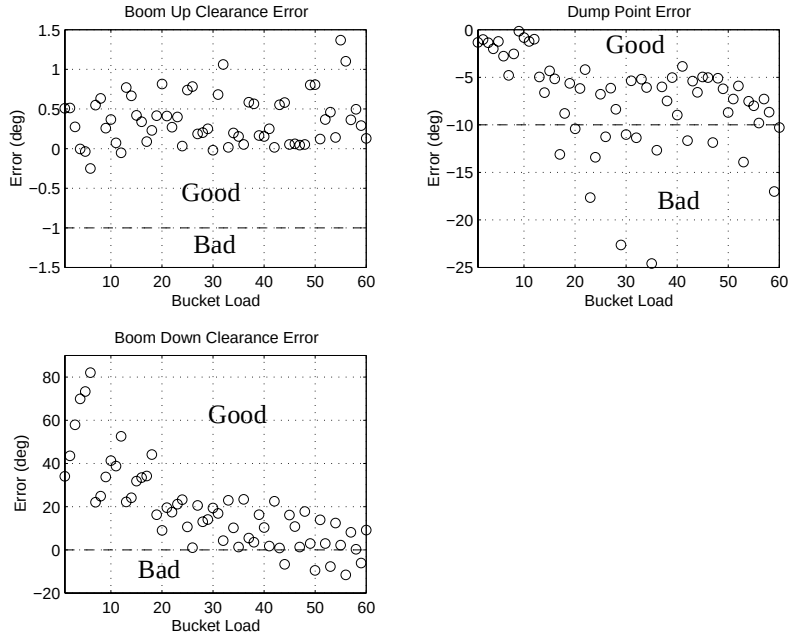


Figure 9.14: Plots of error scores for each bucket loading cycle.

9.6.4.2 Task Constraint Error

Figure 9.14 show how well the adaptive motion planning system did in satisfying the task constraints. The error scores for each of the sixty bucket loads is shown. The horizontal lines represent the error thresholds. The plots are labeled above and below the thresholds as to which error values are good and which are not.

For the *Boom Up* clearance error, the boom always cleared the truck and never came close to the -1° error threshold. This is because in the topology of the test worksite, the boom was already high enough to begin swinging right away even at the dig location that was closest to the truck. The boom always reached its desired angle by the time the bucket reached the truck.

The error for the *Dumping Motion* begins well within the -10° error threshold. As the adaptive motion planning system explores its options, the errors grow. Some even surpass the -10° threshold. These always occur when the swing moves through its largest range, thus high swing speed and controller overshoot contribute to the high error values. Notice however, that as the bucket loading cycles continue, the errors do start to converge to the error threshold.

Similarly, for the *Boom Down* error, high positive values mean the boom begins to lower far away from the truck. As the cycle times get faster, and boom begins to lower closer to the truck at the given error threshold of 0°. Safety margins built into the command parameter computation prevented the excavator's actions that produced the negative errors to collide with the truck.

9.6.4.3 Vehicle State History

Figure 9.15 show the joint traces for the first truck and the tenth truck that was loaded in this experiment. The graphs are plotted on the same time scale for comparison purposes.

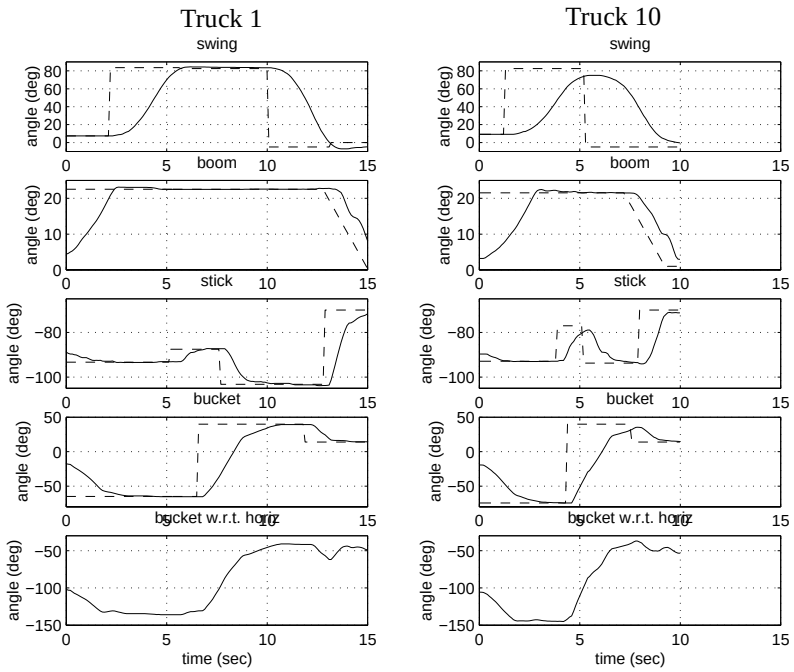


Figure 9.15: Joint traces of the excavator joints for the first truck, using default actions, and the last truck, using the best actions found by the adaptive motion planning system.

A comparison of the joint traces reveals how the time was saved. For the tenth truck, the swing does not stop at all when at the truck, as opposed to the first truck where the swing joint comes to a stop to wait for the other joints to act. In the case of the tenth truck, the swing also begins to move earlier in the cycle than the first truck. During the swing's motion to the truck and back, the stick and bucket joints are able to recoordinate themselves such that the bucket is able to open sooner for the tenth truck.

9.6.4.4 Action Parameter Values

Figure 9.16 shows how the seven action parameter values changed for each of the ten trucks. The first truck begins with the safe, though slow, default values of each action parameter. The general trends of motion can be easily seen for each action parameter plot. Sharp changes in direction can be accounted for when the error thresholds are exceeded and a new way of selecting an action takes over. Slight changes in truck location and dig location can also affect the parameter's values, since they are a function of the task state, which was never exactly the same for each truck. Towards the end of the test, some parameters seem to converge to the same value.

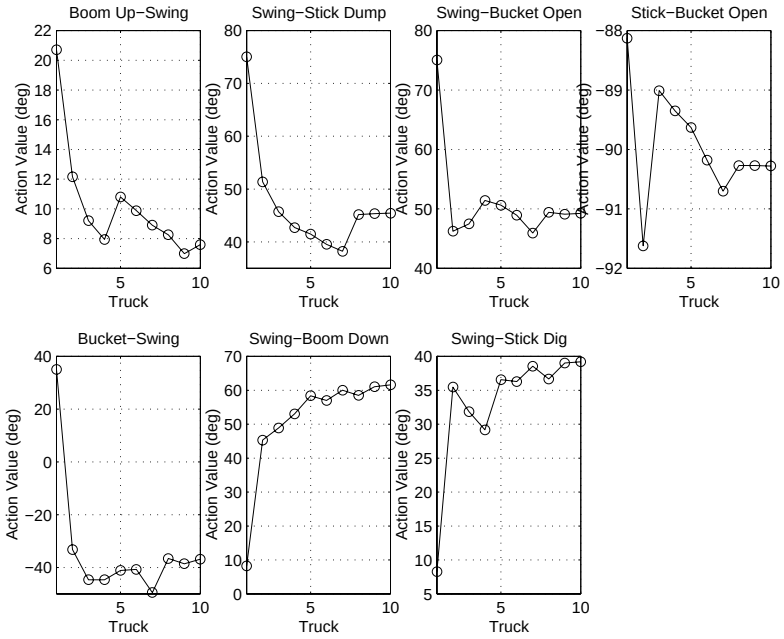


Figure 9.16: Graphs of action parameter values for one loading cycle over the course of ten trucks.

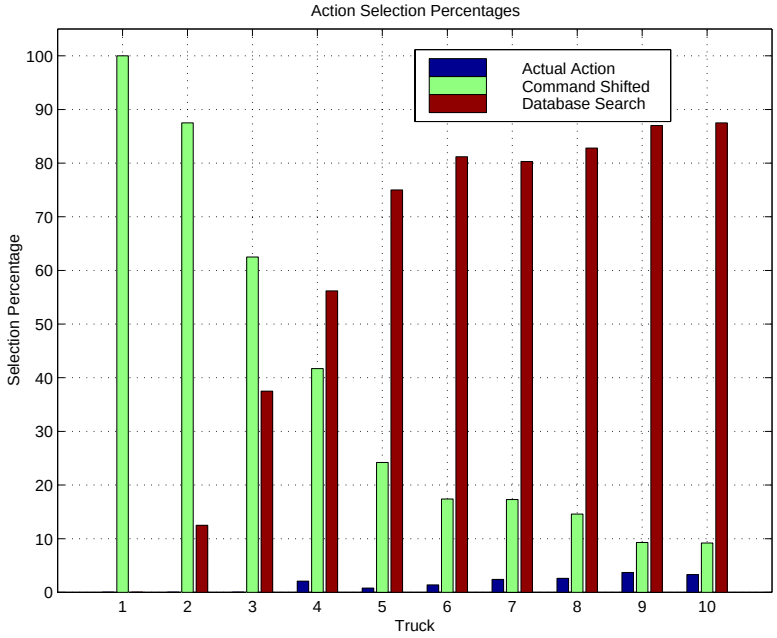


Figure 9.17: Chart showing what percentage of total actions for the policy were chosen by the different action selection methods.

Figure 9.17 displays how the actions that are stored in the policy were chosen among the three

action selection options. Recall that the three action options for a given task state are: 1) the actual action that was just executed, 2) the action that is generated by the command shifting function, or 3) the action that is found with the experience data base search.

After the first truck is loaded, 100% of the actions in the policy were found with the command shifting function. This makes sense, since there is not enough experiences in the experience data base to use that option. As more trucks were loaded, the reliance on the command shifting function decreases, and the experience data base search provides the majority of the actions for the policy. For the later trucks, a few of the actual actions taken survive as well.

9.6.5 Comparison to Human Expert Operator

To determine how well the adaptive motion planning system's performance compares to a human, a human expert excavator operator was asked to load trucks at the test site. Like the automated system, six buckets were loaded into each truck which was parked in roughly the same location.

Table 9.1 summarizes the human operator's performance in loading three trucks.

Human Operator	Truck 1	Truck 2	Truck 3
Loading Time	98.1 sec.	86.2 sec.	100.4 sec.

Table 9.3: Human operator's truck loading times.

The truck loading learning system's fastest truck loading time, as shown in Figure 9.13, was 91 seconds, which is comparable to the human operator's performance.

It is difficult, however, to directly compare bucket loading cycles head to head simply because the testing conditions cannot be duplicated exactly for the human and automated systems[1]. Differences in soil conditions, digging styles, truck location, and overall loading strategy affects the time it takes to load a truck. For example, if the human operator decides to dig all six buckets of soil very close to the truck, rather than at the automated systems dig locations, the human's times will most certainly be faster. It is also difficult to completely separate the digging portion from the free motion portion of the loading task during the human operator's tests in order to directly compare just the free motion times.

However, some general comparisons can be made. For one, it appears that the adaptive motion planning system's performance is on par with a human expert operator who is working at peak performance. Over the course of a work shift, the human's productivity would certainly decrease with fatigue and required breaks.

Figure 9.18 shows the joint traces for both the automated system and the human operator for one

1. Several months separated when data was taken for the human operator and the automated system.

truck, consisting of six bucket loading cycles. In the swing joint trace, the peaks correspond to the desired dump location over the truck and the valleys are the times during digging. Notice that the automated system's dig locations are sequential as it digs laterally across the digging region, where the human's dig location are widely scattered. The human does not spend as long digging as the automated system does. It is encouraging to see that the swing traces from the learning system do have the same shape as the human's swing joint traces. The swing does not dwell at the truck. The shapes of the other joint traces are similar between automated system and human as well.

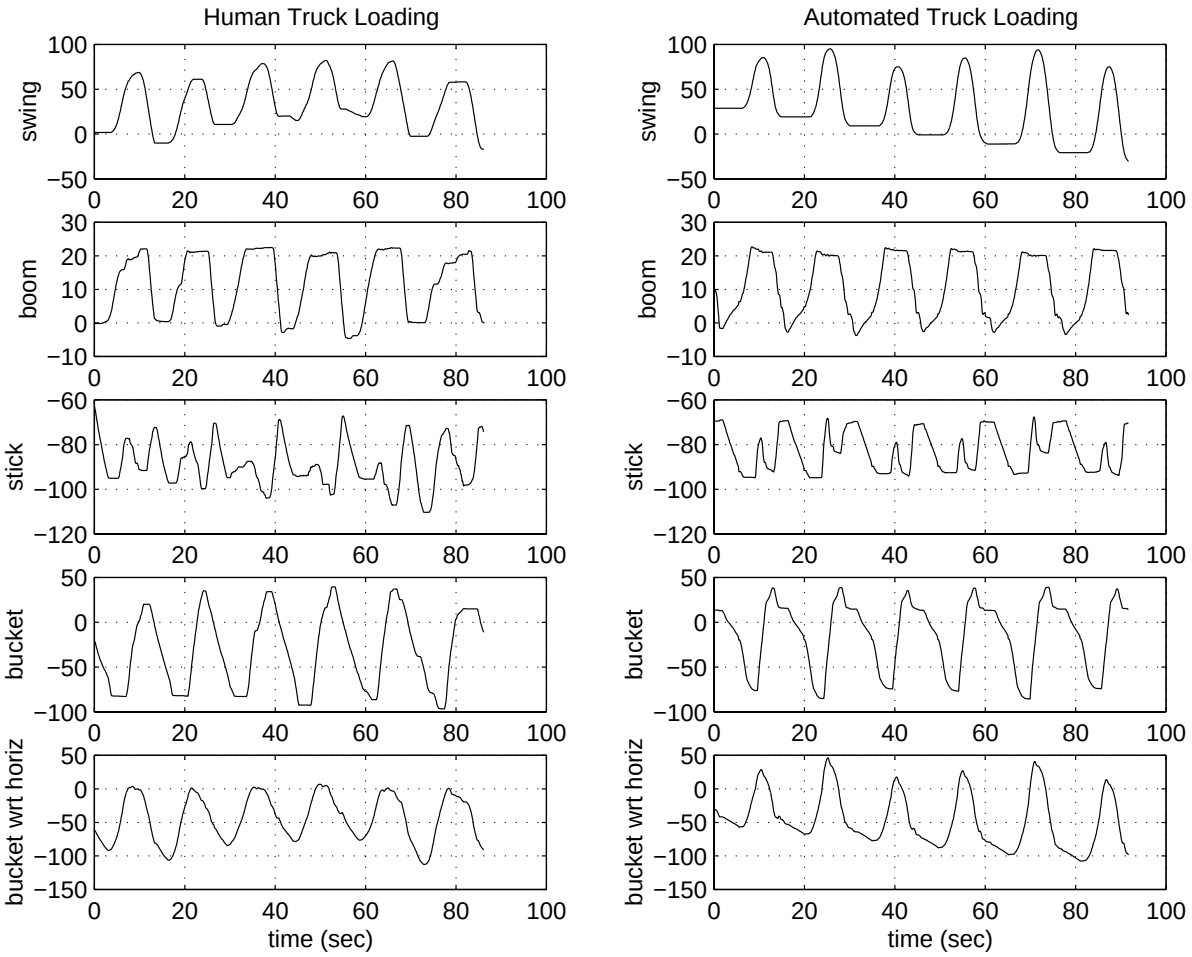


Figure 9.18: Excavator joint traces loading one truck with six buckets. (left) Human expert. (right) Adaptive motion planning system.

9.7 Simulation

This section describes other experiments with the adaptive motion planning system using the simulated excavator. Working with a simulator offers a number of advantages including the ability to load many more trucks in a very short period of time, and the ease in changing the topology of the work site to test system performance. The simulated excavator also has different dynamic charac-

teristics than the real excavator testbed. The biggest difference is lack of actuator coupling in the simulated excavator. This means that the adaptive motion planning system may produce slightly different motion plans than the real machine.

9.7.1 Side Loading

The first loading configuration was the same side loading configuration that is used for the real machine tests. Figure 9.19 shows a picture of the simulated work site for this loading configuration.

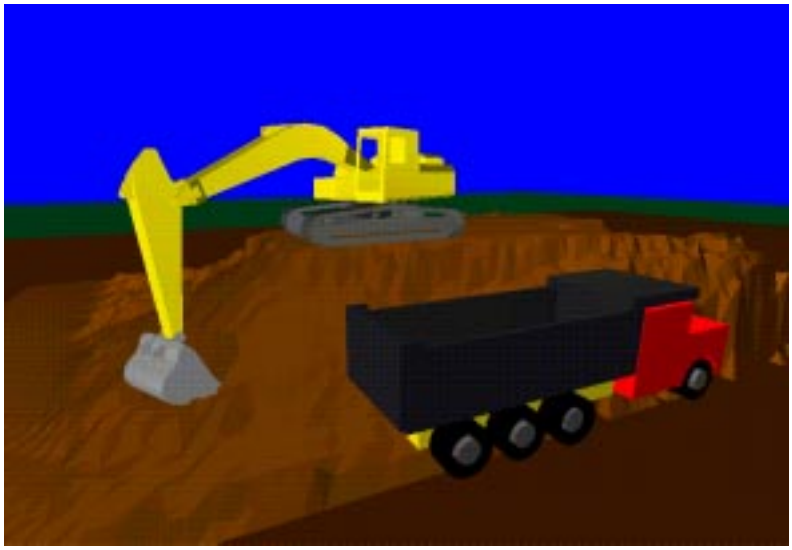


Figure 9.19: Simulated work site set up for side loading configuration.

As with the experiments with the real excavator, six buckets were loaded into each truck, whose parking location did not change. The soil was then reset to its original state and a new truck was loaded. However, forty trucks, instead of ten, were loaded with the simulated excavator.

Figure 9.20 shows the times for the free motion portion of the task for each truck. Notice that it takes a little longer to converge than for the real truck loading system. The free motion times dropped from 120.9 seconds to 67.6 seconds, a 79% increase in productivity.

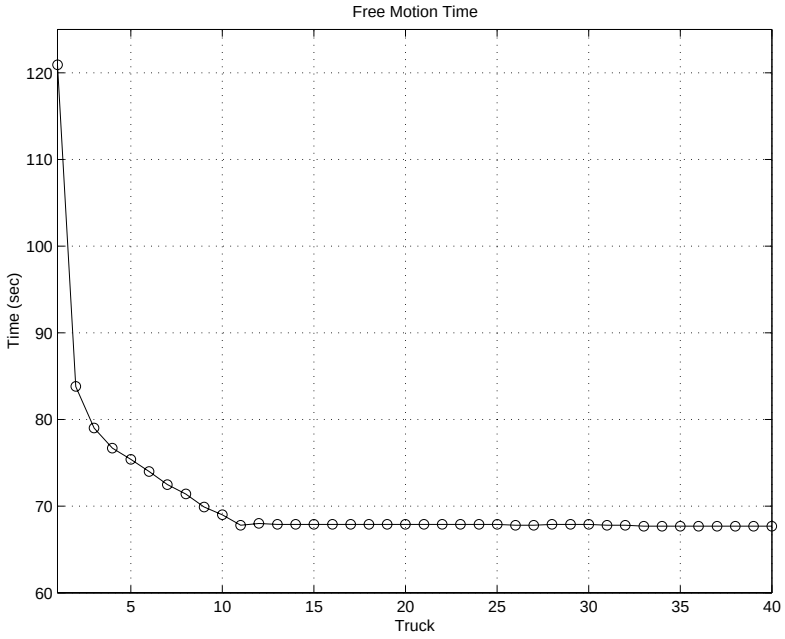


Figure 9.20: Graph of free motion loading times for each truck for simulated side loading.

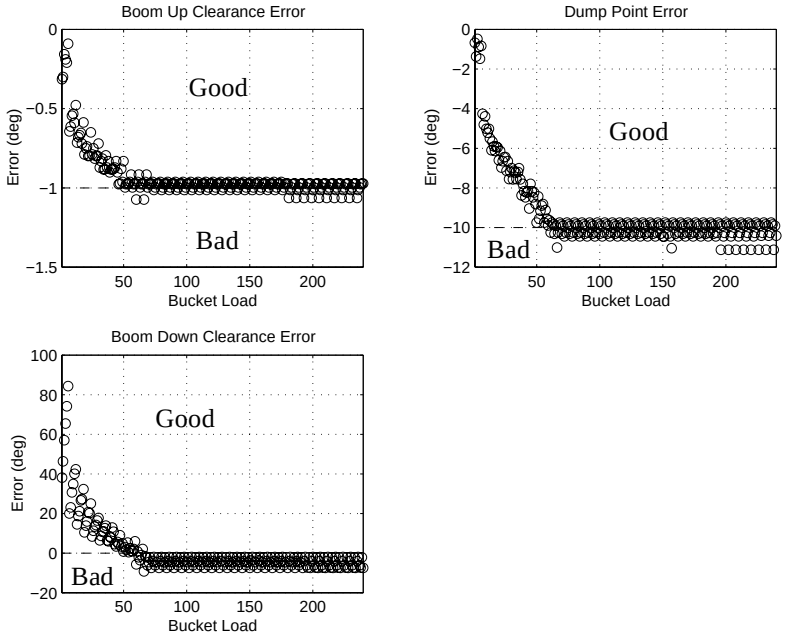


Figure 9.21: Task constraint errors for simulated side loading.

Figure 9.21 displays the error scores for each bucket loading cycle. Notice that the adaptive

motion planner with the simulated excavator can select actions which produced much better behaved task performance. This is because there is perfect machine control in the simulated system, with no disturbances or controller overshoot to cause poor error scores for what the system believes are reasonable actions to take.

9.7.2 Other Worksite Configurations

Two other worksite topologies were created and tested in simulation. Same-level loading is shown in Figure 9.22 and end loading is shown in Figure 9.25. In these new worksite configurations, the adaptive motion planning system also performed well with increases in productivity between 70-80%. Plots of the time and error scores for both configuration are shown in Figures 9.23, 9.24, 9.26, and 9.27. For both configurations, the overall loading times were longer than the side loading case. FOr same-level loading, the boom must be raised approximately 20 degrees higher, resulting in longer loading cycle times. For end loading, the error threshold on the dumping motion must be made very tight since the truck bed is very narrow.



Figure 9.22: Same-level loading using simulator

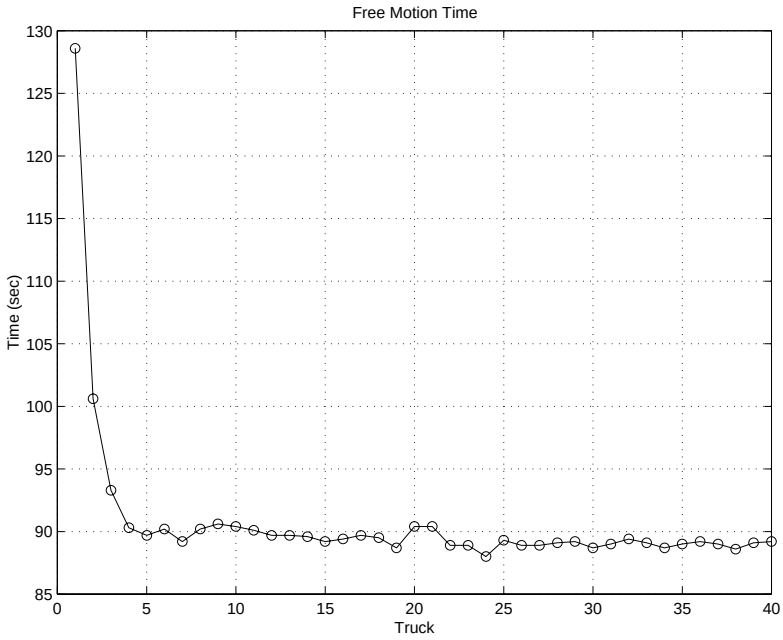


Figure 9.23: Graph of free motion loading times for each truck for simulated side loading.

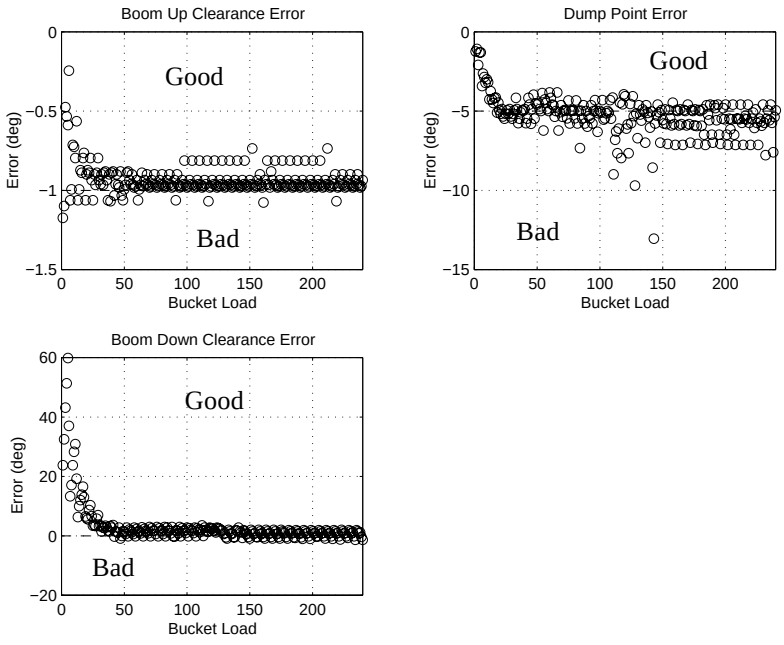


Figure 9.24: Task constraint errors for simulated side loading.



Figure 9.25: End loading using simulator

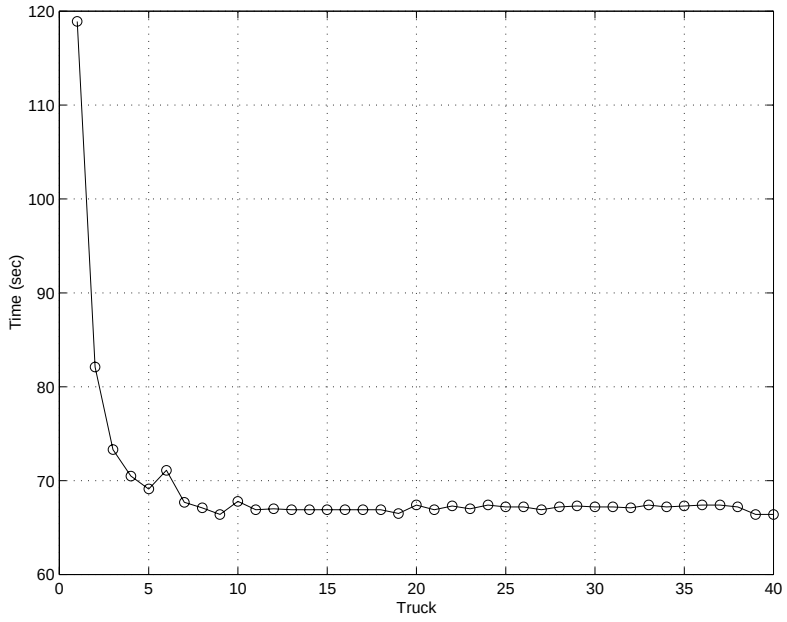


Figure 9.26: Graph of free motion loading times for each truck for simulated end loading.

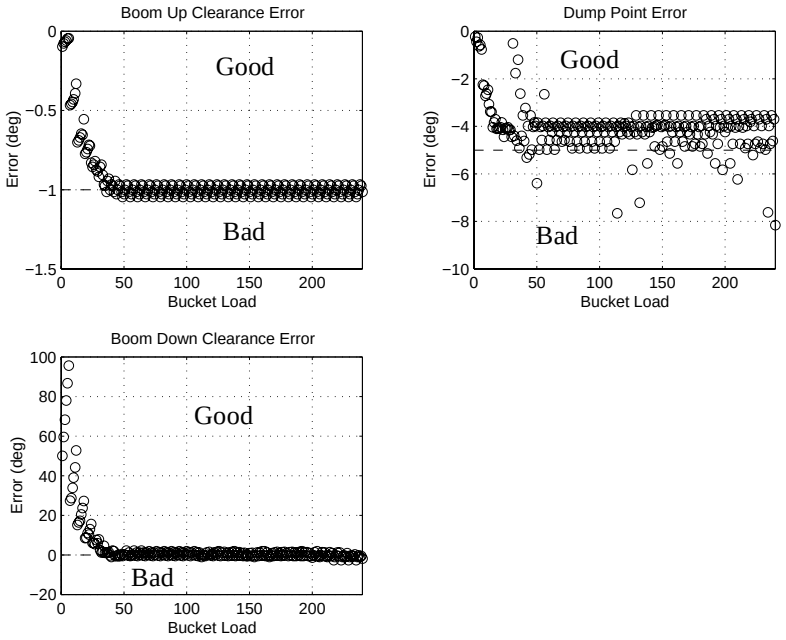


Figure 9.27: Task constraint errors for simulated end loading.

The next and final chapter presents some general conclusions, contributions and future directions of this work.

Chapter 10 Conclusions, Future Work, and Contributions

This chapter presents general conclusions from this research. Many conclusions about individual system components can be found in the Discussion sections of their appropriate chapters. Section 10.2 presents some directions of future research and extensions to the current adaptive motion planning system. Finally, the contributions of this thesis are presented in Section 10.3.

10.1 Conclusions

The goal of this research was to develop an algorithm that planned the free motion of a hydraulic excavator performing the task of truck loading. The desired goal was to achieve maximum productivity in a wide variety of operational conditions. One way to measure productivity was to compare the autonomous loading system's performance to that of a highly skilled human operator working in similar conditions.

The approach involved optimizing for execution time in the space of actions that are constrained by tolerances on acceptable task error. The data used in the optimization are collected on-line as the excavator performs its task. This provides operational flexibility as the system can be used on any type of excavator in any working environment. The motion planning system can automatically adapt to its specific working conditions and achieve top task performance.

10.1.1 Research Issues Revisited

Chapter 1 listed several research issues that were explored in this thesis. This section presents conclusions concerning these research issues.

10.1.1.1 Large state and action spaces

A high dimensional action space makes the task of finding the best set of actions challenging. One way that this was dealt with was by separating the actions into smaller dimensional spaces. This could be done because of the sequential nature of the scripted motion.

Even with this simplification, however, searching through high dimensional action spaces could still take too much time. Instead, a policy was used to store only the best actions. The actions in the policy could be used to find other good actions by interpolation or other generalization techniques. This makes finding the best action to take in any situation a very quick process.

10.1.1.2 Continuous state and action spaces

Memory-based learning techniques are ideal for dealing with continuous state and action spaces. Because it automatically allocates space where needed to store the information, no arbitrary discretization is needed.

10.1.1.3 No vehicle models

Because the modeling process is difficult for the hydraulic excavator, it was decided that no a priori models of the vehicle dynamics would be used in the motion planning system. If a model was available, however, the autonomous excavator could use it to initially search for the best actions given the current worksite topology and soil conditions. This way, the excavator would not need to begin with slow, though functional, actions and improve from there. Instead, it could begin to work very quickly, and still use on-line information to fine tune its internal data representation that maps actions to rewards and achieve even higher levels of productivity that the models alone could not produce.

Another possibility is the use of human data to populate the space of good actions. A human operator could operate the excavator in the given worksite for several loading cycles. The autonomous system would record the task state, and extract the action parameter values from the vehicle motion. This approach would circumvent the need for a search since it is assumed that the human operator is performing the best actions right from the start.

10.1.1.4 Efficient Use of Available Data

One of the keys to initial performance increases and rapid task improvement is the command shifting function described in Chapter 7. Rather than reducing the total vehicle's performance to a small set of numbers, the actual vehicle state history is used to find better actions very quickly. In

many domains, including search and reinforcement learning, heuristics provide a tremendous advantage in realizing rapid improvement.

10.1.1.5 Safety

There is a trade-off between improving task performance quickly and improving task performance safely. In this research, because of the nature of the application, steps were taken to ensure safe performance improvement. This included limiting the rate of change on newly suggested actions, as well as controlling the boundaries of the actions that are considered during the optimization phase. In practice, it appeared that this did not cause a tremendously disappointing decrease in the rate of task improvement. For instance in the truck loading experiments, by the fourth truck the task performance had nearly reached optimality.

10.1.2 Learning

Many observers of this work have commented that the excavator is "learning." This is true, although one must be careful in defining exactly what is being learned.

For many learning systems, the actions themselves, or higher level combinations of primitive actions are being learned. This is not the case for this research. For the task of loading trucks, the types of actions and their general sequence is already known.

Instead, this learning system is really an optimization system. It is finding the best values of a small set of action parameters that specify the coordination of the action sequence. In this way, it can be considered to be learning a skill. The notion of "learning" is valid, however, since the data that it is using to optimize the action parameter values is gathered during actual task execution, as opposed to coming from a predefined model.

10.1.3 Magic Numbers

A goal of the adaptive motion planning system, or any system which claims to learn, is to eliminate the need to tune so-called "magic" numbers. In early versions of the truck loading parameterized script, the values of the action parameters were the "magic" numbers, or rather they were computed by "magic" equations. It was unclear if these equations were computing the best set of action parameters. The adaptive motion planning system took advantage of the motion repetition and availability of large amounts of data to find the best values for the action parameters automatically.

However, it seemed that for every magic number that was eliminated, two more appeared to take its place. While the adaptive motion planning system is able to automatically compute good sets of action parameter values, it still required information about position and velocity thresholds, error thresholds, weighting factors for the function approximators, etc. Many of these numbers may be vehicle or task dependent. much like the original action parameters. In other words, it still required some a priori knowledge in order to get the learning system to operate as if it had no a priori knowledge.

Magic numbers are a fact of life. There really is no way to eliminate every number, threshold or value that is needed in a complex system. The quest now is to be able to supply these numbers intelligently rather than in an ad hoc manner. In some cases, it may be possible to find these numbers automatically, like the weighting factors for the locally weighted function approximators, which can be found by cross validation of the data. Observations from human supervisors may be another way if the human supervisors are presented with an intuitive interface for adjusting these numbers.

10.1.4 Controller Issues

The joint controller that was provided with the excavator test bed was both a necessity and a limitation. The joint controller provided a clear separation between the problems of machine control and the problems of planning, the latter being the focus of this research. By using a controller, the set of commands are desired positions for each joint, and it was assumed that the controller would accurately achieve these joint positions.

In terms of achieving the goal of maximum productivity, however, the excavator's performance is ultimately limited by the controller. For instance, a very overdamped controller, where the joints are very slow in reaching their commanded goal positions, would result in an autonomous machine that could not compete with the speed of a human operator no matter how "optimal" the planning. The perceived smoothness of the excavator's motion is also a function of the controller. The motion planner developed in this research has no influence over these aspects of machine performance.

The good news, however, is because the adaptive motion planning system uses information that is gathered on-line, the controller's characteristics are not a concern. This means that if the machine control was changed, the adaptive motion planner would not need to change. No new machine models would have to be derived or any initial machine data collected. Instead, during task motion, the adaptive system would adjust the joint coordination to take advantage of the changed machine capabilities such as increased joint speed.

10.1.5 Maximizing Productivity

An explicit goal of the motion planning algorithm that was developed for this research was to achieve maximum task productivity. The success of this goal depends on what criteria are used to define productivity.

There are many aspects to the productivity of a mass excavation task. These include the global excavation plan for a work shift, the number, size, and frequency of the trucks that are loaded, the type of material that is being excavated, and the speed of the excavator for each bucket load, just to name a few. Many of these aspects are out of the control of an autonomous system and are best left to a human expert. But there are aspects of the task where an automated system has the advantage. One of these aspects is the problem which was tackled in this research, the free motion planning of the excavator. The adaptive motion planning system did maximize productivity for this aspect of the overall mass excavation task.

10.2 Future Work

10.2.1 Automatic Script Generation

Currently, the parameterized scripts that are used by the adaptive motion planning system are constructed by hand from a thorough study of the task and input from human experts in the task domain. For some tasks, this may be a very tedious process. There is also a question on the correctness and completeness of the scripts. Could parameterized scripts be generated automatically?

The knowledge that is encoded in the script is a sequence of primitive actions to perform a given task. For example, the truck loading parameterized script encodes both what joints to move, their directions of motion, and also the rough sequence of actions by specifying dependencies on the motion.

Perhaps these primitive actions and action sequence could be derived from human examples. For instance, a human expert in the task could perform a few task execution cycles. The automatic script generator could watch and record the types and sequence of actions. From there, it may be possible to construct a parameterized script complete with script steps, script rules, and even script parameters.

10.2.2 Dynamic Scripts

In the current adaptive motion planning system, the primitive actions and their sequence do not change, although the motion is quite flexible because of the parameterization of the script. However, a future extension of the parameterized scripting approach could have the script steps change dynamically based on the current worksite conditions.

For example, in the current parameterized script, the boom must begin raising after digging before the swing is allowed to move to the truck. This sequence is set. However, it may be the case that in certain situations higher productivity is achieved by moving the swing first. The parameterized script should be able to change the order of the script steps, the motion dependencies, and the task constraints which define this motion. This ability would move the system one step closer to truly “learning” how to load trucks optimally.

10.2.3 Confidence Intervals

Memory-based learning techniques can supply levels of confidence on the returned prediction. This capability could be very useful to the inherently dangerous mass excavation task. Both the experience data base and the policy could employ their use. For example, the search for the best action in the experience data base could be limited by a minimum acceptable confidence on the prediction of the score. That way, the predicted scores of the suggested best actions would be closer to the true scores.

The policy could use confidence intervals when computing an action set for a given task state. If the confidence on the action is too low, then the policy could default to the slow, though functional, default actions.

The downside of confidence intervals is that they again require even more numbers provided by the system designer for acceptable confidence levels. The other implication is that the rate of performance improvement would be less aggressive, as the actions would tend to favor those closer to known experiences rather than jumping to new, though untried, actions that may be better right away.

10.2.4 Controller-less Scripts

The current parameterized script algorithm returns desired position commands for each joint. This requires a low level joint controller which, as stated earlier, can have an effect on total task productivity. An alternative to the script's output could be open loop velocities of each joint, or even raw joystick signals that bypass the controller completely. As in optimal bang-bang control, the joystick signals would most likely be full on and full off. The problem is finding the exact time to make the transition so that the joint arrives at the right location.

The adaptive motion planning system would then learn these switching times, which again are the action parameters. There is a little more danger involved in this type of script. For one, it is unclear how to start with a safe, default action that does the right thing. For example, suppose we wish to have the swing joint move 90 degrees. It may take several tries of different on/off switching points to determine the proper command sequence, and in the meantime the swing joint may overshoot or undershoot its goal. However, the potential benefits would be vehicle behavior that is only limited by the dynamics of the vehicle itself.

10.2.5 Using Perception

A distinct advantage of using an adaptive motion planning approach is the ability to easily deal with changing conditions. One condition that could change over the course of a work shift is the soil conditions. Recall that the soil conditions affect the bucket angles at which the soil remains captured and which it completely falls out of the bucket. Incorrect choices for these bucket angles could mean spilled soil or wasted time in opening the bucket too far. Could these angles, and perhaps other command script parameters, also be found automatically?

This requires some other means of evaluating the performance of the vehicle. The vehicle joint traces are not enough to determine where the soil was actually deposited. Another possible evaluation tool could be the use of perception. For example, the truck bed could be scanned with a ranging sensor after each bucket load is deposited, and the location of the soil could be found. This could then be compared to the desired location and an error score can be computed.

This idea was briefly explored early in this research. However, due to difficulties in detecting both when the soil has fallen out of the bucket and where it had fallen with the current ALS sensors, it was decided not to pursue it further. Perhaps other sensing modalities such as cameras would be better for computing this information.

If this information was available, it could be used to compute error scores, which would serve to place limits on the space of possible actions, as well as adjust the values of the task state. For

example, if it was determined that the bucket was opening too far for the current soil conditions, then the bucket angle, which is a task state variable, would be changed to a new value. The adaptive motion planning system would find appropriate action parameter values taking this new task state value into account. For example, if the bucket does not need to open as far as it had been, this may result in the swing returning to the digging region sooner, resulting in faster loading cycle times.

10.3 Contributions

This research has developed a motion planning algorithm, parameterized scripting, that can plan the motions of autonomous machines that perform highly repetitive tasks. The parameterization of the script offers flexibility to handle a wide variety of worksite topologies and working conditions. It exists at a level above simple teach-playback systems as it is able to plan motions at the task level.

A key to the success of parameterized scripts is selecting good values for the script parameters. This research has developed an adaptive approach to computing the values of the script parameters based on the machine's own performance in its current working conditions. This aspect enhances the operational flexibility of the system, as it can be used on any machine without the need for a priori dynamic models. To satisfy the goal of maximum productivity, this research has developed a performance evaluation system that is used to optimize the script parameters. The result of this approach has been a motion planning system that can compete with a highly skilled human in the short term and outperform him in the long term.

This work is a major piece of the overall Autonomous Loading System. The Autonomous Loading System project definitively demonstrated that it is possible to automate machines such as excavators for tasks such as mass excavation and truck loading. It is hoped that the research that has been developed for the project will one day be used to aid operators, with enhanced sensing or training capabilities, for machines that are in use today.

The target application for this research was the task of loading trucks with a hydraulic excavator. However, there exist a universe of other construction, mining, forestry, and industrial machines, shown in Figure 10.1 and Figure 10.2, which possess many of the same characteristics as the hydraulic excavator.

They all perform highly repetitive, deliberate tasks. All are desired to work at peak performance. These machines have a large number of controllable degrees of freedom, some being more complicated than an excavator. Accurate modeling of the dynamics of these machines is also very difficult making them ideal candidates for an on-line adaptive motion planning approach.



Dragline



Excavator with electromagnet



Giant hydraulic front shovel



Mulcher



Stripping shovel



Bucket excavator

Figure 10.1: Other types of construction, mining, forestry or industrial machines where the use of the adaptive motion planning system would be beneficial. (pictures taken from (Bruun & Keith, 97))



Crane



Trencher



Tree chipper



Underground coal miner



Wheel loader



Forestry machine

Figure 10.2: More construction, mining, forestry or industrial machines where the use of the adaptive motion planning system would be beneficial. (pictures taken from (Bruun & Keith, 97))

Some of the machines are very similar to the excavator but possess different end-effectors, such as an electromagnet or a gripping device, as in the mulcher and tree chipper machines. Other machines have different kinematic configurations and are much larger, possibly resulting in significantly different dynamic characteristics such as the hydraulic front shovel, stripping shovel, dragline, and crane. The forestry machine also has several more controllable degrees of freedom associated with the cutting head. Other machines are more of a departure from an excavator, but the nature of the tasks that they do are similar in that they are highly repetitive and can be expressed as a series of steps, such as with the trencher, underground coal miner, and wheel loader.

Finally, this research has also explored and handled many of the challenges of applying machine learning techniques, in particular robot skill learning and memory-based learning, to a real-world application. This research serves as another data point for the learning community. It also serves as proof that giving machines the intelligence to modify their own behavior does indeed result in higher performance and productivity.

Appendix A Command Script Parameters

This appendix presents the kinematic equations for computing several of the command script parameters. These include the swing, boom, and stick angles which are computed using knowledge of the truck and dump location. The diagrams in these sections will use a simplified kinematic model of the excavator's implements shown in Figure A.1. l_{sw}, l_{bm}, l_{st}, and l_{bk} are the swing[1], boom, stick, and joint link lengths respectively.

A.1 Boom Clearance Angle

This command parameter is the angle of the boom joint that allows the bucket to safely clear the truck. It is a function of the truck height, truck location, and excavator link lengths.

A simple way of computing this command parameter is shown in Figure A.2. By taking the combined link lengths of the stick and bucket, plus a safety margin between the end of the implements and the top of the truck, this boom angle guarantees clearance of the truck for any stick and bucket angles. The equation shows how this angle is computed.

1. The swing link length is a small offset between the swing pivot axis and the origin of the boom joint.

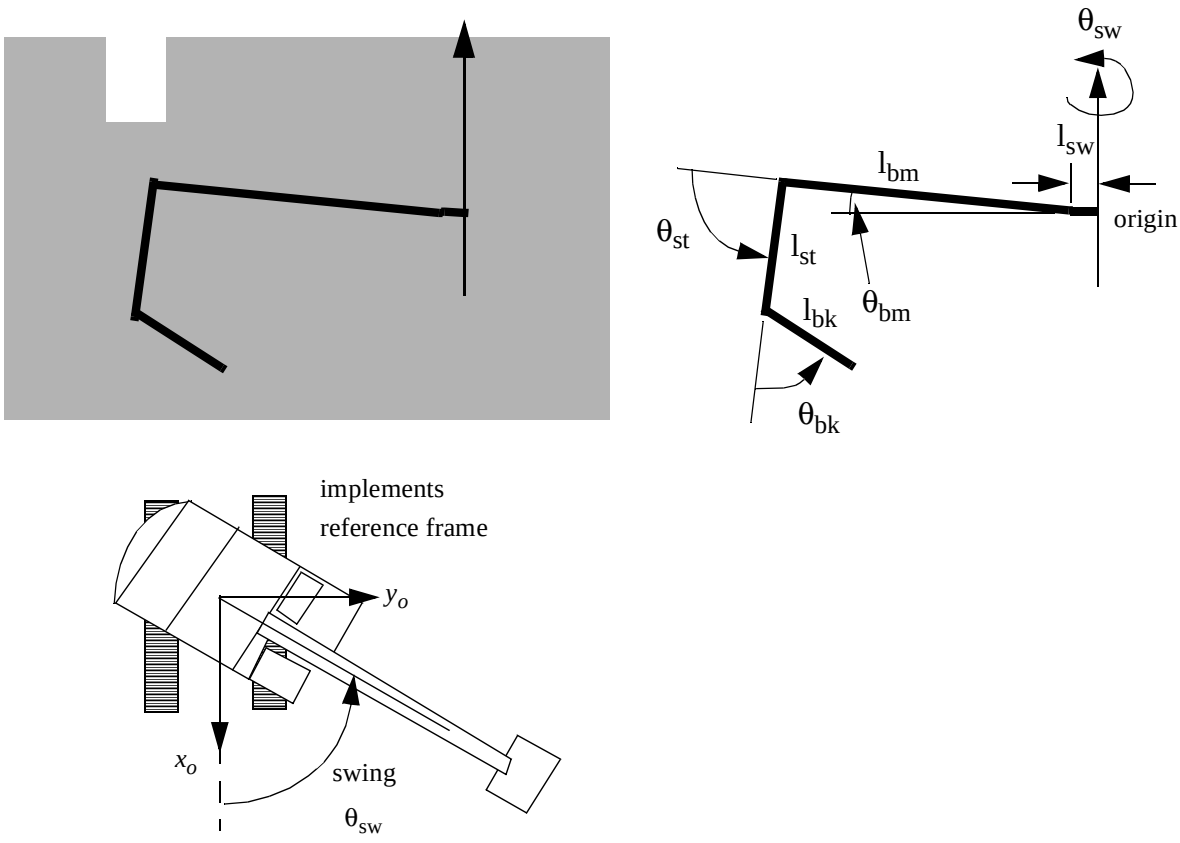


Figure A.1: Schematic of simplified excavator kinematic model that is used in the following command parameter computation sections.

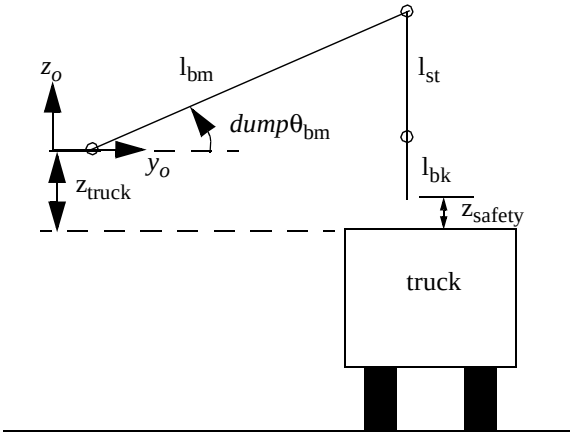


Figure A.2: Schematic showing one method of computing the angle of the boom that guarantees clearance of the truck.

$$dump\theta_{bm} = \text{asin}\frac{(l_{st} + l_{bk} + z_{safety}) - z_{truck}}{l_{bm}}$$

One problem with this method, however, is if the truck is parked far away from the excavator, the boom clearance angle might be higher than necessary. Because the boom is not lowered over the truck, the soil could be dropped from too high an elevation and potentially damage the truck.

A more sophisticated way of computing the boom clearance angle takes the truck's lateral position into account. First, the point on the near truck bed sidewall that is closest to the excavator is found. The closest point from the implement reference frame origin to the truck is found at the intersection of two perpendicular lines, one running along the near truck wall and the other originating from the implement reference frame which is shown in Figure A.3.

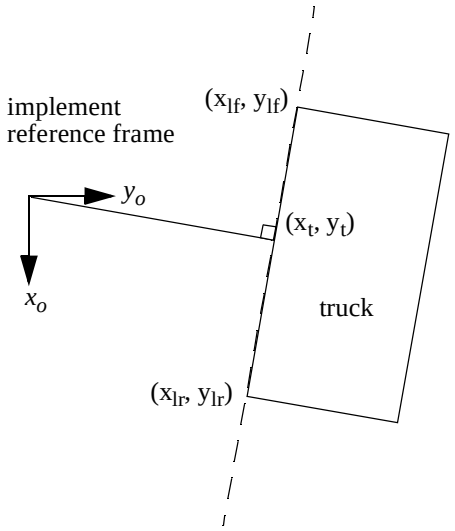


Figure A.3: The closest distance from the implement reference frame to the truck is a line that is perpendicular to the truck bed wall.

The equation for the line along the truck bed wall is given by

$$(y - y_{lr}) = m(x - x_{lr})$$

where the slope m is

$$m = \frac{y_{lf} - y_{lr}}{x_{lf} - x_{lr}}$$

The equation for the other line is found from the negative reciprocal of the slope.

$$y = \frac{-1}{m}x$$

Solving for the point of intersection (x_t, y_t), the equations are

$$x_t = \frac{(mx_{lr} - y_{lr})m}{1 + m^2}$$

$$y_t = \frac{-1}{m}x_t = \frac{y_{lr} - mx_{lr}}{1 + m^2}$$

If the intersection point lies outside the extent of the truck bed, then the coordinates of the closest truck bed corner are returned as the nearest point to the excavator.

The boom clearance angle can now be found by using the inverse kinematics of the implements. This is done in a two dimensional sense in a plane of the implements that is determined by the line between the reference frame origin and (x_t, y_t). The intersection point is transformed to the plane of the implements by

$$x'_t = \sqrt{(x_t)^2 + (y_t)^2}$$

z_t is the z coordinate of the truck bed, which is the distance below the reference frame origin to the truck. Figure A.4 shows this method of computing the boom clearance angle.

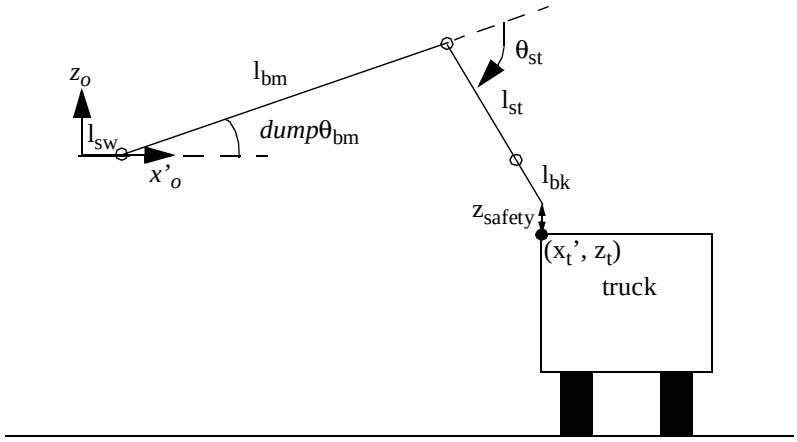


Figure A.4: Schematic illustrating a second way of computing the boom clearance angle that takes the lateral position of the truck into account.

The inverse kinematic equations that solve for the stick and boom angles are

$$\theta_{st} = -\text{acos}\frac{(x'_t - l_{sw})^2 + (z_t + z_{safety})^2 - l_{bm}^2 - (l_{st} + l_{bk})^2}{2l_{bm}(l_{st} + l_{bk})}$$

$$dump\theta_{bm} = \text{atan2}((z_t + z_{safety}),(x'_t - l_{sw})) - \text{atan2}((l_{st} + l_{bk})\sin\theta_{st},(l_{bm} + (l_{st} + l_{bk})\cos\theta_{st}))$$

The *acos* term is negated to return the elbow-up inverse kinematics solution. If the argument to the *acos* function is greater than 1 or less than -1, that means the truck is beyond the reach of the excavator and an error is reported.

A.2 Swing Dump Angle

The swing dump angle is calculated using the desired dump location coordinates. Figure A.5 and the equation below show how this is computed.

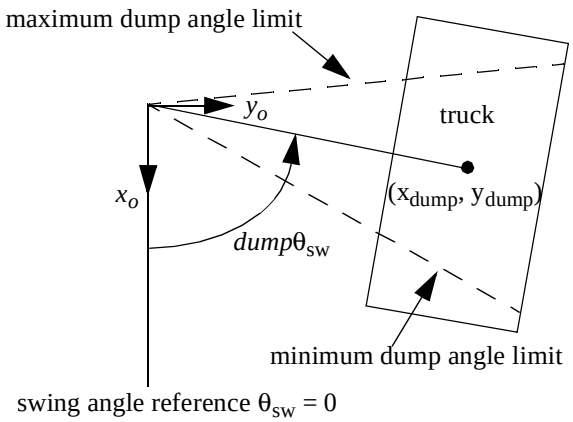


Figure A.5: Schematic showing the swing angle computed from the desired deposit location.

$$dump\theta_{sw} = \text{atan}2(y_{dump}, x_{dump})$$

In order to prevent dumping soil in the front and rear walls of the truck bed, minimum and maximum swing dump angles are computed taking the width of the bucket into account. If the computed swing dump angle is beyond these limits, it is reset to the appropriate limit.

A.3 Stick Dump Angles

The two angles that define the stick goals for the dumping motion are also computed using the inverse kinematics of the implements, but without using the bucket link length and using the boom clearance angle $dump\theta_{bm}$ found previously.

For this computation, two lines are drawn down the length of the truck bed. The far line represents the desired location of the bucket (wrist) joint over the truck bed for the first stick dump angle. The near line represents the desired location of the bucket joint for the second dump angle. Thus, these two lines define the range of motion of the stick during the dumping motion. The height of the bucket joint over the truck bed is not set; it is a result of the solution to the inverse kinematics equations. Figure A.6 shows the two truck bed lines. Intersecting these lines is another line drawn from the origin of the implements reference frame to the desired dump location. The intersections of this line with the two truck bed lines are shown as points (x_1, y_1) and (x_2, y_2).

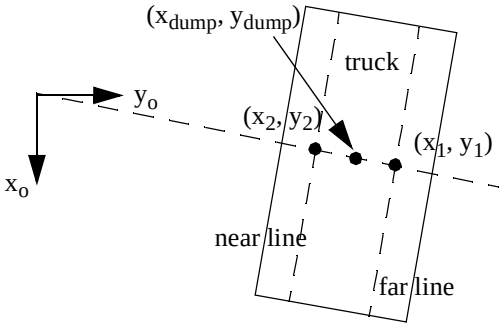


Figure A.6: Two imaginary lines are drawn down the length of the truck bed to position the wrist joint for the two-step dumping maneuver.

Since the boom angle is already set, computing the two stick angles is relatively simple.

$$dump\theta_{st1} = -(dump\theta_{bm} + \text{atan2}(\sqrt{l_{st}^2 - (x'_1 - x_{bm})^2}, (x'_1 - x_{bm})))$$

where

$$x_{bm} = l_{sw} + l_{bm}\cos(dump\theta_{bm})$$

$$x'_1 = \sqrt{(x_1)^2 + (y_1)^2}$$

If the term under the radical in the equation for $dump1\theta_{st}$ is negative, that means the wrist joint cannot reach the desired point over the truck. In this case the stick joint is set to its maximum angle and a warning is returned.

Similarly, for the second stick dump angle, the same equation holds, only now x'_1 is replaced with x'_2 where

$$x'_2 = \sqrt{(x_2)^2 + (y_2)^2}$$

and

$$dump\theta_{st2} = -(dump\theta_{bm} + \operatorname{atan2}(\sqrt{l_{st}^{2} - (x'_{2} - x_{bm})^{2}}, (x'_{2} - x_{bm})))$$

Appendix B Locally Weighted Learning Techniques

This appendix describes the two locally weighted function approximation techniques that were used in the adaptive motion planning system: locally weighted linear learning and kernel regression. More information on the many advantages of these techniques can be found in the literature (Atkeson, 91)

B.1 Locally Weighted Linear Regression

In locally weighted linear regression, the output y is a linear function of the inputs x.

$$y = \sum_{i=1}^{m} \alpha_i x_i$$

where m is the number of terms which form the query, x_i is the ith term of the query, and α_i is the coefficient on the ith term of the query. The coeffiecients α_i are what are computed in the regression calculation. A 1 is assumed to be appended on the end of the query so the linear model does not need to pass through the origin (thus, there are actually m+1 coefficients that are solved for).

Two requirements for locally weighted linear regression is a distance function and a weighting function. The distance function computes the distance between the query and the data points in the data base. The weighting function takes as input the distance and computes a weight for each data point in the data base. Larger weights mean that the data point is close to the query and has more influence on the resulting output.

The distance function that was chosen for this system is a standard Euclidean distance. The weighting function is a decaying exponential

$$w_i = e^{\frac{-D_i}{k}}$$

where w_i is the weight of data point i, D_i is the Euclidean distance between the query and the ith data point (or its square), and k is a scaling factor which determines how global or local the weighting is. Lower values of k result in higher weights for more distant data points resulting in a more global fit of the data. The k that gives the best model fit can be found automatically by cross validation of the data base.

For each prediction, the weight for each and every data point in the data base is computed. In applications where there is a large number of data (on the order of thousands of points), this may result in prediction times which are too slow. If this is the case, more sophisticated data structures and retrieval routines such as k-d trees can be used.

Each input term of the points in the data base is normalized between 0 and 1 to prevent scaling differences from distorting the distance calculations. The terms are scaled based on the minimum and maximum ranges on the input terms that have been seen so far. Thus, the first data point is both at its minimum and maximum range. As new data points, with different values, are added to the data base, the data are rescaled as the minimum and maximum ranges change.

The regression equation which solves for the coefficients of the linear expression can be written in matrix form

$$X\alpha = y$$

where X is an n x m+1 matrix of the input terms of each data point in the data base, α is an m+1 x 1 vector of the coefficients of the linear expression, and y is an n x 1 vector of an output attribute, such as time or error. n is the number of data points that are in the data base and m is the number of input attributes, such as task state and action dimensions, so each row of X and y represent the input and output terms of one data point.

The weights of each data point can be placed on the diagonal of an n x n diagonal matrix W. Thus, the weight associated with the first data point would be in the upper left corner of the W matrix. This matrix is then premultiplied by X and y to weight the inputs and outputs of each data point in the data base.

$$WX\alpha = Wy$$

This does not change the dimensions of the original matrix equation.

The coefficients α can then be solved. For this system, the coefficients were found by singular value decomposition. The coefficients are then used to find the predicted output for the given query.

B.2 Kernel Regression

Kernel regression is another name for a weighted average. Like the locally weighted linear regression described in the previous section, each data point in the data base is assigned a weight based on its distance to the query.

$$w_i = e^{\frac{-D_i}{k}}$$

where w_i is the weight of data point i, D_i is the Euclidean distance between the query and the ith data point (or its square), and k is the scaling factor, or *kernel*, which determines how global or local the weighting is.

The value of the output is simply the sum of the weighted outputs divided by the sum of the weights or

$$y_{pred} = \frac{\sum w_i y_i}{\sum w_i}$$

References

(Aboaf et. al., 88) Aboaf, E.W., Atkeson, C.G., and Reinkensmeyer, D.J. 1988. “Task-Level Robot Learning”, in *Proceedings of the International Conference on Robotics and Automation*.

(Aboaf et. al., 89) Aboaf, E.W, Drucker, S.M., and Atkeson, C.G. 1989. “Task-Level Robot Learning: Juggling a Tennis Ball More Accurately”, in *Proceedings of the International Conference on Robotics and Automation.*

(Atkeson, 91) Atkeson, C.G. 1991. “Using Locally Weighted Regression for Robot Learning”, in *Proceedings of the International Conference on Robotics and Automation*.

(Atkeson et. al., 97) Atkeson, C.G., Moore., A.W., and Schaal, S. 1997. “Locally Weighted Learning”, *Artificial Intelligence Review* Vol. 11, pp. 11-73.

(Bisse et. al., 94) Bisse, E., Hemami, A., Boukas, E.K. 1994. “Optimal Excavation Path Planning for Scooping by a Bucket”, in *Proceedings of the 6th Canadian Symposium on Mining Automation*.

(Bruun and Keith, 97) Bruun, E., and Keith, B. 1997. “Heavy Equipment: Giant machines that crush, cut, dig, dredge, drill, excavate, grade, haul, pave, pulverize, pump, push, roll, stack, thresh, and transport big things”, Black Dog & Leventhal Publishers, Inc.

(Bullock et.al., 90) Bullock, D.M., Apte, S., and Oppenheim, I.J. 1990. “Force and Geometry Constraints in Robot Excavation”, In *Proceedings Space 90: Engineering, Construction, and Operations in Space*.

(Craig, 86) Craig, J.J. 1986. “Introduction to Robotics Mechanics and Control”, Addison-Wesley Publishing Co.

(Huang and Bernold, 93) Huang, X.D., and Bernold, L.E. 1993. “Robotic Rock Handling During Backhoe Excavation”, *Automation and Robotics in Construction.*

(Kaelbling et. al., 96) Kaelbling, L.P., Littman, M.L., and Moore, A.W. 1996 “Reinforcement Learning: A Survey”, *Journal of Artificial Intelligence Research* Vol. 4, pp. 237-285.

(Kositsky et. al., 98) Kositsky, M., Flash, T., and Ullman, S. 1998. “A Cluster Memory Model for Learning Sequential Activities”, *Neural Information Processing Systems.*

(Krishna and Bares, 99) Krishna, M., and Bares, J. 1999. “Constructing Hydraulic Robot Models using Memory-Based Learning“, to appear in *ASCE Journal of Aerospace Engineering*.

(Lawrence et. al., 95) Lawrence, P.D., Salcudean, S.E., Sepehri, N., Chan, D., Bachmann, S., Parker, N., Zhu, M., and Frenette, R. 1995. “Coordinated and Force-Feedback Control of Hydraulic Excavators”, in *Proceedings of the International Symposium on Experimental Robotics.*

(Lay, 98) Lay, N.K. 1998. “Just-In-Time Object Recognition and Localization using Range Sensors on a Mobile Platform”, PhD Thesis Proposal, Robotics Institute, Carnegie Mellon University.

(Leger et. al., 98) Leger, C., Rowe, P., Bares, J., Boehmke, S., and Stentz, A. 1998. “Obstacle Detection and Safeguarding for a High-speed Autonomous Hydraulic Excavator”, in *Proceedings of SPIE*, Vol. 3525.

(Lever et. al., 94) Lever, P., Wang, F., and Chen, D. 1994. “A Fuzzy Control System for an Automated Mining Excavator”, in *Proceedings of the International Conference on Robotics and Automation*.

(Lever and Wang, 95) Lever, P.J.A., and Wang, F. 1995. “Intelligent Excavator Control System for Lunar Mining System”, *Journal of Aerospace Engineering*, Vol. 8, No. 1.

(Mahadevan and Connell, 92) Mahadevan, S., and Connell, J. 1992. “Automatic Programming of Behavior-based Robots using Reinforcement Learning”, *Artificial Intelligence*, vol. 55, Nos. 2-3. pp. 311-365

(Mataric, 94) Mataric, M.J. 1994. “Reward Functions for Accelerated Learning”, in *Proceedings of the Eleventh International Conference on Machine Learning*.

(Moore, 91) Moore, A.W. 1991. “Efficient Memory Based Robot Learning”, PhD Thesis, University of Cambridge.

(Moore et. al., 95) Moore, A.W., Atkeson, C.G., and Schaal, S.A. 1995. “Memory-Based Learning For Control”, Carnegie Mellon Technical Report CMU-RI-TR-95-18.

(Moore et. al., 97) Moore, A.W., Atkeson, C.G., and Schaal, S. 1996. “Locally Weighted Learning For Control”, *Artificial Intelligence Review* Vol. 11, pp. 75-113.

(Parker et. al., 93) Parker, N.R., Salcudean, S.E., and Lawrence, P.D. 1993. “Application of Force Feedback to Heavy Duty Hydraulic Machines”, in *Proceedings of the International Conference on Robotics and Automation.*

(Pedersen, 98) Pedersen, J. 1998. “Robust Communication for High Bandwidth Real-Time Systems”, Carnegie Mellon Technical Report CMU-RI-TR-98-13.

(Press et. al., 88) Press, W.H., Vetterling, W.T., Teukolsky, S.A., and Flannery, B.P. 1988. “Numerical Recipes in C: The Art of Scientific Computing”, Cambridge University Press.

(Rocke, 94) Rocke, D. 1994. “Control system for automatically controlling a work implement of an earthmoving machine to capture material”, U.S. Patent 5528843.

(Rocke, 95) Rocke, D. 1995. “Automatic excavation control system and method”, U.S. Patent 5446980.

(Rowe and Stentz, 97) Rowe, P., and Stentz, A. 1997. “Parameterized Scripts for Motion Planning”, in *Proceedings of the International Conference on Intelligent Robots and Systems*.

(Sakai and Cho, 88) Sakai, T., and Cho, K. 1988. “Operation System for Hydraulic Excavator for Deep Trench Works”, in *Proceedings of the 5th International Symposium on Robotics in Construction*.

(Salcudean et, al., 97) Salcudean, S.E., Tafazoli, S., Lawrence, P.D., and Chau, I. 1997. “Impedance Control of a Teleoperated Mini Excavator”, in *Proceedings of the International Conference on Advanced Robotics*.

(Sameshima and Tozawa, 92) Sameshima, M., and Tozawa, S. 1992. “Development of Auto Digging Controller for Construction Machine by Fuzzy Logic Control”, in *Proceedings of the Conference of Japanese Society of Mechanical Engineers*.

(Santamaria et. al., 98) Santamaria, J.C., Sutton, R.S., and Ram, A. 1998. “Experiments with Reinforcement Learning in Problems with Continuous State and Action Spaces”, *Adaptive Behavior*, 6(2), pp. 163-218.

(Schaal and Atkeson, 94) Schaal, S. and Atkeson, C.G. 1994. “Robot Juggling: Implementation of Memory-Based Learning”, *Control Systems Magazine*, 14(1), pp. 57-71.

(Schneider, 93) Schneider, J.G. 1993. “High Dimension Action Spaces in Robot Skill Learning”, Technical Report, Computer Science Department, University of Rochester.

(Schneider, 95) Schneider, J.G. 1995. “Robot Skill Learning Through Intelligent Experimentation”, PhD Thesis. Department of Computer Science. University of Rochester.

(Seward et. al., 92) Seward, D., Bradley, D., Mann. J., and Goodwin, M. 1992. “Controlling and Intelligent Excavator for Autonomous Digging in Difficult Ground”, in *Proceedings of the International Symposium on Automation and Construction*.

(Seward et. al., 96) Seward, D., Margrave, F., Sommerville, I., and Morrey, R., 1996. “LUCIE the Robot Excavator - Design for System Safety”, in *Proceedings of the International Conference on Robotics and Automation.*

(Shi et. al., 95) Shi, X., Wang, F., and Lever, P. 1995. “Task and Behavior Formulations for Robotic Rock Excavation”, in *Proceedings of the International Symposium on Intelligent Control*.

(Shi et. al., 96) Shi, X., Lever, P., and Wang, F. 1996. “Experimental Robotic Excavation with Fuzzy Logic and Neural Networks”, in *Proceedings of the International Conference on Robotics and Automation.*

(Singh, 95) Singh, S. 1995. “Learning to Predict Resistive Forces During Robotic Excavation”, in *Proceedings of the International Conference on Robotics and Automation*.

(Singh, 97) Singh, S. 1997. “The State of the Art in Automation of Earthmoving”, *ASCE Journal of Aerospace Engineering* Vol. 10, No. 4.

(Singh and Cannon, 98) Singh, S., and Cannon, H. 1998. “Multi-Resolution Planning for Earth-moving”, in *Proceedings of International Conference on Robotics and Automation*.

(Song and Koivo, 95) Song, B., and Koivo, A.J. 1995. “Neural Adaptive Control of Excavators”, in *Proceedings of the International Conference on Intelligent Robots and Systems*.

(Stentz et. al., 98) Stentz, A., Bares, J., Singh, S., and Rowe, P. 1998. “A Robotic Excavator for Autonomous Truck Loading”, in *Proceedings of the International Conference on Intelligent Robots and Systems*.

(Sutton, 88) Sutton, R.S. 1988. “Learning to Predict by the Method of Temporal Differences”, *Machine Learning Vol. 3, No. 1*, pp. 9-44.

(Thrun, 98) Thrun, S.B. 1998. “The Role of Exploration in Learning Control”, in *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches*.