Technical Report 1079

# Task-Level Robot Learning

Eric W. Aboaf

MIT Artificial Intelligence Laboratory

*This blank page was inserted to preserve pagination.*

# Task-Level Robot Learning

by

Eric W. Aboaf

B.S. Engineering, University of Pennsylvania (1986)
B.S. Economics, University of Pennsylvania (1986)

An Earlier Version was Submitted to the
Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the
Degree of Master of Science

at the
Massachusetts Institute of Technology

August, 1988

©Massachusetts Institute of Techonology 1988

*This empty page was substituted for a blank page in the original document.*

# Task-Level Robot Learning

by

Eric W. Aboaf

## Abstract

We are investigating how to program robots so that they learn from experience. Our goal is to develop principled methods of learning that can improve a robot's performance of a wide range of dynamic tasks. Our interest is in complex tasks such as throwing, catching, batting, yo-yoing, and juggling. We have developed one method of learning, *task-level learning*, that successfully improves a robot's performance of both a ball-throwing and a juggling task.

With task-level learning, a robot practices a task, monitors its own performance, and uses that experience to adjust its task-level commands. For example, we have programmed a robot to juggle a single ball in three dimensions. The robot practices the juggling task by batting a ball into the air with a large paddle. The robot uses a real-time binary vision system to track the ball and measure its own performance. Task-level learning consists of building a model of the performance errors at the task level during practice. The robot compensates for the performance errors by using that model to refine the task-level commands. When using task-level learning, the number of hits that the robot can execute before the ball is hit out of range dramatically improves.

Task-level learning is a general method of improving a robot's performance of complex dynamic tasks. Task-level learning serves to complement other approaches for improving robot performance such as model calibration. Our investigation is one step in the process of developing a theoretical and experimental foundation for robot learning.

Thesis Supervisor: Christopher G. Atkeson
Title: Assistant Professor, Brain and Cognitive Sciences Department

*This empty page was substituted for a blank page in the original document.*

# Acknowledgements

Many people have helped me in many different ways. Each deserves a great deal of thanks. I would like to thank two in particular:

Chris Atkeson is an adviser who takes you to the circus, buys you toys, and lets you play. He is even willing to let you set the rules of the game. He gets a Big tennis ball that's a bit larger than the ones we juggled.

Steve Drucker is a fellow graduate student who is unparalleled as a coworker and a friend. Without his programming skills or his companionship, the robot would have never hit a single ball. He gets a Big tennis ball that's a bit larger than the ones we juggled.

Many others have provided both technical and moral encouragement. Sundar Narasimhan and David Siegel have developed and supported a real-time multiprocessor system that was indispensible to the juggling experiments. Paul Resnick has listened to my politics and my ambitions. My roommates have made me feel at home. My ex-girlfriends have been a constant source of friendship for me and amusement for others. My parents and siblings have encouraged me during my many years in school.

This thesis describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the A.I. Laboratory's research is provided in part by the Office of Naval Research University Research Initiative Program under Office of Naval Research contract N00014-86-K-0685, and the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85-K-0124. Partial support for the author was provided for by an Office of Naval Research Graduate Fellowship.

*This empty page was substituted for a blank page in the original document.*

# Contents


**1 Introduction** 6

**2 Learning the Ball-Throwing Task** 9
- 2.1 Introduction . . . 9
- 2.2 The Ball Throwing Task . . . 10
- 2.3 The Problem and a Solution . . . 12
  - 2.3.1 What is the Problem? . . . 12
  - 2.3.2 What are the Solutions? . . . 13
- 2.4 The Task Model . . . 13
- 2.5 Task-Level Learning . . . 14
  - 2.5.1 Fixed-Model Learning . . . 15
  - 2.5.2 Refined-Model Learning . . . 19
- 2.6 Discussion . . . 21

**3 Learning the Juggling Task** 23
- 3.1 Introduction . . . 23
- 3.2 The Juggling Task . . . 24
  - 3.2.1 System Description . . . 25
  - 3.2.2 Task Model . . . 30
  - 3.2.3 System Calibration . . . 31
  - 3.2.4 Modeling Errors . . . 32
  - 3.2.5 System Noise . . . 33
- 3.3 Learning the First Hit . . . 34
  - 3.3.1 Convergence Criteria . . . 35
  - 3.3.2 Learning while Suppressing Noise . . . 38
  - 3.3.3 Learning in the Presence of Noise . . . 40
- 3.4 Learning the Second Hit . . . 42
  - 3.4.1 Learning Experiments . . . 43
  - 3.4.2 Juggling Performance After Learning Hits One and Two . 45
- 3.5 Learning the Successive Hits . . . 46
  - 3.5.1 Examining Performance Errors . . . 46
  - 3.5.2 What the Errors Mean . . . 50

3.5.3 Applying State-Based, Task-Level Learning . . . . . 50
3.6 Discussion . . . . . 56

**4 Other Ways to Improve Robot Performance** **58**
4.1 Calibration Approaches . . . . . 58
4.1.1 Component Model Calibration . . . . . 59
4.1.2 Complete System Calibration . . . . . 60
4.2 Non-Calibration Approaches . . . . . 62
4.2.1 Feedback Control . . . . . 62
4.2.2 Iterative Techniques . . . . . 63
4.2.3 Defining the Task in Sensor Space . . . . . 64
4.2.4 Strategy Modifications . . . . . 65
4.3 Calibration and Learning . . . . . 65
4.4 Recent Trajectory Learning Research . . . . . 66

**5 Conclusion and Future Research** **68**
5.1 Conclusion . . . . . 68
5.2 Future Research . . . . . 69

# Chapter 1

# Introduction

We are investigating how to program robots so that they learn from experience. Our premise is that robots can practice a task and use that experience to improve performance. We demonstrate that this premise is valid for two robot tasks—throwing and juggling—and suggest that the performance of other dynamic tasks can be improved by learning from practice.

We base our learning approach on a rather commonplace observation. A person throwing a ball for the first time at a target will often miss. If the ball is thrown too short, the person will aim a little further and throw again. If the ball is thrown too far, the person will aim a little closer. From this simple example, we notice that people tend to vary the *aim* to compensate for the error in performance. We also observe that people are quite willing to practice the task until they finally succeed. We thus pose the question, why not use this approach for a robot?

We develop task-level learning procedures that attempt to mimic the approach people tend to take. The learning procedures formalize the process of correcting for the errors that occur when a robot performs a task. The procedures require that a robot system practices a task, monitors its own performance. and adjusts its commands until the task is performed correctly. A model is used to translate each new task-level command into actuator commands that drive the robot.

The learning procedures are called *task-level* because they directly refine task-level commands, not the low-level actuator commands that drive the robot. In ball-throwing, the learning procedures improve the performance on the task by adjusting where the robot system is *aiming* the ball. The model that transforms this aim into actuator commands is not adjusted. No model calibration is performed during learning. Instead, the aim is adjusted and the model is used to automatically recompute the low-level actuator commands that drive the robot. By judiciously adjusting the robot's task-level aim, the learning procedures compensate for inaccuracies in the model and improve the robot's performance of the task.

Learning at the task level is a promising method of improving a robot's performance of a task. First, less data is necessary to refine the task-level command than to perform extensive model calibration. Instead of making a large number of trial motions to calibrate the system, trial motions that actually attempt the task provide a more concise method of achieving the task goal. Second, learning at the task level reduces the degrees of freedom of the models to be learned. Instead of adjusting all the parameters of the kinematics, dynamics, actuation, and sensing models of a robot, only the task-level commands of the system are modified. In making the task-level adjustments, the learning procedures compensate for the structural modeling errors in the lower level component models of the robot. Ultimately, task-level learning and other types of model calibration can probably be used simultaneously to improve performance.

## Thesis Outline

In Chapter 2 we develop two learning procedures in the context of a throwing task. In this task, the third link of a robot is used to catapult a ball at a target. The robot aims at a target that is a known distance away. A model of ballistics, kinematics, and dynamics is used to calculate the actuator commands that should drive the robot arm to swing forward, lofting the ball onto the target. The robot throws the ball with this sequence of commands. When the ball hits the target area, the robot uses a camera to monitor the exact landing location of the ball. The robot uses this measure of its own performance—how closely the ball landed to the target—to change its aim for the next throw.

After discussing how the robot improved its performance of the throwing task, we analyze and generalize the learning procedures. We first describe the task model that is used to represent the ball-throwing task. We explain how the throwing aim is modified to perform the task. We elaborate on the conditions required for two different task-level learning procedures to converge to the desired performance. We demonstrate that accurate internal models of the robot improve the speed of the learning process. We then extend the learning algorithms to multi-dimensional tasks.

In Chapter 3 we demonstrate the effectiveness of task-level learning on a complex task—juggling. In this task the robot bounces a ball on a paddle. A vision system tracks a tennis ball in the air and estimates the time and location at which it should be hit. Based on this estimate, the robot system computes a sequence of actuator commands that drive the paddle with the proper upward motion. After the paddle hits the ball during the swing, the robot system tracks the motion of the ball, monitors its performance, and prepares for the next hit. For each hit, the robot tries to hit the ball so that it will land at the center of the paddle on the next bounce.

We describe a sequence of task-level learning experiments on the juggling system. A task model is first used to describe the juggling task. With this task model and the learning algorithms, the system learns to perform the first hit in a juggling sequence. The robot system then learns to successfully perform two consecutive hits. The juggling system finally uses a task-level, state-based learning algorithm to successfully hit the ball more than 70 times in a row. At each step, the performance of the juggling system is dramatically improved when task-level learning algorithms are applied.

In Chapter 4 we explain how other researchers have improved the performance of robot systems. We survey a variety of recent descriptions of robot systems, analyzing the approaches researchers took to improve system performance. We discuss calibration approaches which improve the accuracy of the models used to control the robot. We also discuss iterative and feedback control schemes that are similar to the task-level learning procedures that we develop. Finally, we devote a section to identifying the tradeoffs between calibration and learning approaches.

## Thesis Goals

We are pursuing two specific goals with this research. The first is to develop some general learning principles that improve a robot's performance of a wide range of tasks, of which throwing and juggling are examples. A second goal is to explore learning at the task level, a method that is complementary to extensive component model calibration. Both goals are explored by developing and implementing task-level learning on two robot tasks—throwing and juggling.

*This empty page was substituted for a blank page in the original document.*

# Chapter 2

# Learning the Ball-Throwing Task

In this chapter, we present a theoretical and experimental framework for task-level learning. This learning research is developed in the context of a robot ball-throwing task. We begin by describing a ball-throwing task that a robot system performs. We represent the task and system with a task model. Two learning procedures are then developed and applied to improve the performance of the ball-throwing system. We present experimental results, and assess the convergence and performance of each learning procedure. Finally, we generalize the task-level learning procedures so that they can be applied to more complicated tasks.

## 2.1 Introduction

We have demonstrated the process of task-level learning with a ball-throwing robot system that improves its performance with practice (Figure 2.1). Given the location of the target, the system uses a ballistic model, a kinematic model, a trajectory model, and a dynamics model to calculate a sequence of torque commands to drive the robot arm. Using this set of actuator commands and a simple feedback controller, the robot throws the ball at the target. A vision system measures where the ball lands with respect to the target. Based on the error in performance, the system applies a task-level learning procedure to modify its aim—where it is trying to throw the ball. With this new aim and the same models of ballistics, kinematics, trajectories, and dynamics, the system computes a new set of commands to drive the arm. The robot system then throws the ball again with the updated sequence of actuator commands. The robot continues this sequence of performing the task, monitoring its own performance, and refining its aim until the task is successfully performed.

The task-level learning procedure modifies the system's aim based on the error in task performance. The learning procedure is based on our observation that a person adjusts his aim to compensate for errors in performance. If the first throw lands further than the target, the robot system will aim closer. If the first

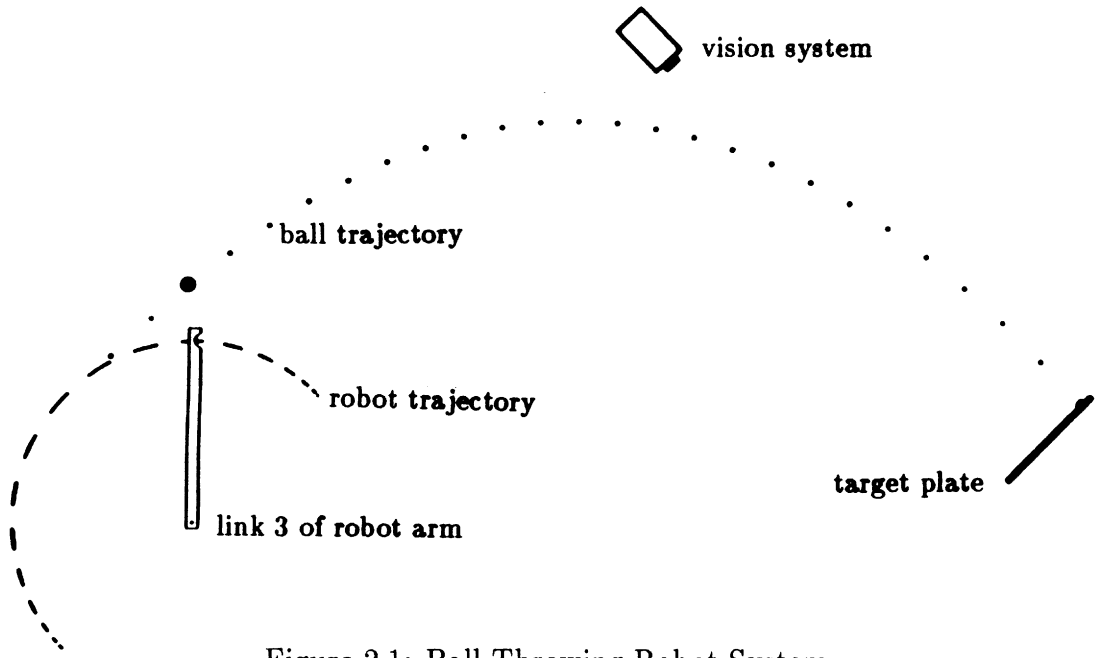


Figure 2.1: Ball-Throwing Robot System

throw lands too close, the robot system will aim further. We have developed two learning algorithms—fixed model learning and refined model learning—that adjust the system's aim to compensate for errors in performance. Both learning procedures improve the performance of the ball-throwing robot system.

## 2.2 The Ball Throwing Task

The task is to throw a ball at a target. Figure 2.1 illustrates the robot system that is configured to accomplish this task. The system includes the MIT Serial Link Direct Drive Arm [An, Atkeson, and Hollerbach 1988], a target plate, and a video camera.

The last link of the robot is used to throw the ball at the target plate. The robot is positioned so that the last link of the arm rotates in a vertical plane. A 0.04 m rubber ball is placed into a 0.035 m diameter hole at the end of the third link of the robot. The robot swings this link through a 180° arc, catapulting the ball to the target. The ball leaves the hole as the robot arm decelerates during the throw. No release mechanism is used, but the release position is assumed to be when the last link is approximately halfway through the trajectory. The height of the ball when it hits the target plate is monitored and improved by learning.

A video camera records where the ball hits the target plate. The impact of the ball on the target plate is sensed by a force sensor. This signal is used to choose

the video frames which are stored for later analysis. After the throw, the location of the ball on the target plate is manually measured from the appropriate video frame. This location measures the robot system's performance of the task.

Much like our conception of the human thrower, the robot system aims at the target. Based on the measured distance between robot and target, an *inverse ballistics model* calculates the desired release velocity of the ball. A simple ballistic model, including only gravity, is used to represent the flight of the ball

$$y_d = v_{ball} \cdot \sin(\theta_{ball}) \cdot t - 1/2 \cdot g \cdot t^2 \quad (2.1)$$

$$x_d = v_{ball} \cdot \cos(\theta_{ball}) \cdot t \quad (2.2)$$

For a given position of the target $(x_d, y_d)$ and release angle of the ball $(\theta_{ball})$, the necessary release velocity is calculated by eliminating the variable $t$ from Equations (2.1) and (2.2). The release angle of the ball ($\theta_{ball} = 45°$) remains fixed throughout the experiment as part of the task strategy.

An *inverse kinematics model* relates the desired release angle and release velocity of the ball to the joint angle and joint velocity of the arm. The model relates the release angle of the ball $(\theta_{ball})$ to the angle of the arm $(\theta_{arm})$ by an offset

$$\theta_{arm} = \theta_{ball} + 90° \quad (2.3)$$

The offset is set to 90° since the ball leaves in a direction perpendicular to the arm. Figure 2.1 shows that the ball will be released at 45° ($\theta_{ball} = 45°$) when the arm is 135° ($\theta_{rel} = 135°$) from the horizontal. The kinematics model also calculates the angular velocity of the arm based on the desired release velocity and the length of the arm.

An *inverse trajectory model* computes the sequence of joint angles necessary to swing the arm. The joint is servoed to a fifth-order polynomial trajectory that moves the arm through a 180° arc, from 225° to 45°. The arm is accelerated from rest until it reaches 135°, and then decelerated to rest at 45°. The desired release angle and desired release velocity are assumed to be midway through the trajectory.

An *inverse dynamics model* and a feedback controller are used to accurately drive the robot arm along the desired trajectory. Feedforward torques are calculated using the acceleration profile of the trajectory and the estimated inertia of the third link. A position–velocity feedback controller insures that the arm closely follows the desired trajectory on each throw. The control law that includes both feedforward and feedback torques is

$$\mathrm{T} = \mathrm{T_{ffwd}} - \mathrm{K_p} \cdot (\theta - \theta_\mathrm{d}) - \mathrm{K_v} \cdot (\dot{\theta} - \dot{\theta}_\mathrm{d}) \quad (2.4)$$

where $K_p$ and $K_v$ are the position and velocity feedback gains.

We can formalize the sequence of component models that describe the ball-throwing system. The desired task performance, which we term the aim, can be

mathematically related to the desired torque commands that are computed by the models

$$\text{torque commands} = \hat{\mathbf{D}}^{-1}(\hat{\mathbf{T}}^{-1}(\hat{\mathbf{K}}^{-1}(\hat{\mathbf{B}}^{-1}(\text{aim})))) \quad (2.5)$$

where $\hat{\mathbf{D}}^{-1}$ represents the inverse dynamics model, $\hat{\mathbf{T}}^{-1}$ the inverse trajectory model, $\hat{\mathbf{K}}^{-1}$ the inverse kinematics model, and $\hat{\mathbf{B}}^{-1}$ the inverse ballistics model. We use a caret ($\hat{}$) to denote a *model*. As described above, the ballistics model computes the necessary release velocity of the ball, the kinematics model then calculates the angular velocity of the arm, the trajectory model computes the necessary time-sequence of joint angles, and the dynamics model calculates the torque commands that drive the joint actuator.

## 2.3 The Problem and a Solution

In this section, we explain why any performance improvement is necessary for the ball-throwing system. We begin by describing the errors that occur when the robot system throws a ball—how far from the target the ball actually lands. We explain that the errors in throwing are due to modeling errors. Finally, we present the rationale for using task-level learning to improve the performance of the ball-throwing robot system.

### 2.3.1 What is the Problem?

When the robot throws the ball using these models, the ball misses the target. Based on the measured distance to the target, 5.75 m, the robot system uses the component models described by Equation (2.5) to calculate the required torque commands. The ball is thrown when the robot is commanded with these joint commands. The ball lands 0.28 m above the target in the target plane. We term the 0.28 m error a *performance error*. Based on the models described above, the robot is unable to successfully throw the ball at the target.

The throw misses the target because our models are inaccurate descriptions of the real system. The ballistic, kinematic, trajectory, and dynamics models only approximate the ball-throwing system. As a result, when the ball is initially thrown, the actual performance differs from the desired performance. Many factors can cause this performance error. These factors include an inaccurate measurement of target location, inaccurate kinematic model, inaccurate vision system, inaccurate feedfoward torques, air resistance, torque saturation, sensor miscalibration, and sensor noise, as well as errors in release angle and release velocity. With all these different factors, it is difficult to improve the models so that the ball-throwing system can reliably hit the target.

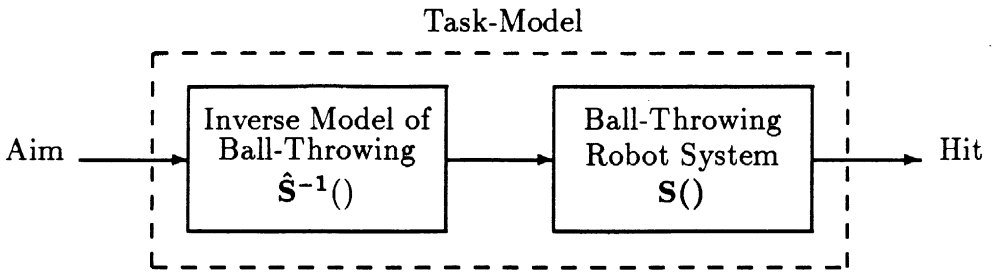


Figure 2.2: Ball-Throwing Task Model

### 2.3.2 What are the Solutions?

Given a system with performance errors, we ask the question: what are the solutions? We suggest that two different approaches can improve the performance of the ball-throwing system: (1) the models of the ball-throwing system can be made more accurate by calibration, or (2) learning algorithms can be applied at the task-level.

Our goal in this research is to explore the second option. Most researchers in robotics address the first approach, and we refer the reader to Chapter 4 for a discussion of robot calibration. We choose to explore an approach that we call *task-level learning*. Instead of calibrating the model of the system, we modify the robot system's task-level command. The task-level command is the target location to which the robot system tries to throw the ball. We begin by formalizing the notion of a *task model* in the next section, and then develop two task-level learning algorithms.

## 2.4 The Task Model

In this section, we formalize the concept of a *task model*. A task model relates the desired performance of a system to the actual performance. In the case of ball-throwing, the task model relates where the system is trying to throw the ball—the aim—to where the ball actually lands—the hit. In the ideal case, when we have a perfect model of the robot system, a task model can be represented by the identity transform, suggesting that the ball will hit wherever the system aims. In general, the task model only approximates the identity transformation.

The task model is composed of two transformations: the inverse model of the system and the system transformation. A block diagram of the task model for the ball-throwing system is shown in Figure 2.2. For ball-throwing, the inverse model of the system is composed of the four component models derived in Section 2.2. This sequence of models transforms an aim—where to hit the ball—to a sequence of actuator commands. The inverse model transformation was described

by Equation 2.5, which we rewrite here

$$\text{commands} = \hat{\mathbf{D}}^{-1}(\hat{\mathbf{T}}^{-1}(\hat{\mathbf{K}}^{-1}(\hat{\mathbf{B}}^{-1}(\text{aim})))) \tag{2.6}$$

This equation, which includes each component model of the ball-throwing system, can be collapsed into

$$\text{commands} = \hat{\mathbf{S}}^{-1}(\text{aim}) \tag{2.7}$$

where $\hat{\mathbf{S}}^{-1}()$ represents the inverse model of the system.

The second part of the task model is the system transformation that describes the ball-throwing robot. This transformation is determined experimentally by commanding the robot actuators and measuring the landing point of the ball. The system transformation relates actuator commands to the landing position of the ball, which we call the hit. Formally

$$\text{hit} = \mathbf{S}(\text{commands}) \tag{2.8}$$

where $\mathbf{S}()$ denotes the ball-throwing system transformation.

Together, the inverse model and the system transformation describe the task. By simply combining Equations (2.7) and (2.8), we obtain a *task model*

$$\text{hit} = \mathbf{S}(\hat{\mathbf{S}}^{-1}(\text{aim})) \tag{2.9}$$

In the ideal case, when the model perfectly describes the system, $\mathbf{S}(\hat{\mathbf{S}}^{-1}())$ reduces to the identity transformation. As discussed above, the ball-throwing models only approximate the robot system, and so the ball-throwing task model only approximates the identity transformation.

## 2.5 Task-Level Learning

In this section, we develop a task-level learning approach that improves the performance of the ball-throwing robot. The basis of task-level learning is to modify the system's task-level command based on errors in task performance. The learning procedure formalizes our interpretation of the human ball thrower who modifies his aim based on how far the ball landed from the target.

We develop two general learning procedures—fixed-model learning and refined-model learning—that improve system performance with practice. Both refine the system's aim based on errors in task performance. Fixed-model learning uses the task model to transform this new aim into new robot commands. Refined-model learning is an interpolation procedure that refines the task model while computing the robot commands.

### 2.5.1 Fixed-Model Learning

In fixed-model learning, the correct aim is estimated based on the performance errors. In ball throwing, the aim is updated after each throw by the amount the ball missed the target. This measured error—whether positive or negative—updates the aim as a running sum

$$\text{aim}_{n+1} = \text{aim}_n - (\text{hit}_n - \text{target}) \tag{2.10}$$

This new aim is transformed through the inverse ball-throwing model to calculate the refined robot commands

$$\text{command}_{n+1} = \hat{\mathbf{S}}^{-1}(\text{aim}_{n+1}) \tag{2.11}$$

A physical interpretation is helpful for understanding Equations (2.10) and (2.11). In the case where the ball falls short, the performance error is negative, raising the aim by that amount. This action corresponds to our intuition that we should aim higher if we are hitting too low. Together, Equations (2.10) and (2.11) provide the basis for fixed-model learning.

Fixed-model learning was applied to the ball throwing task. The target was placed at a horizontal distance of 5.75 m and a height of -0.9 m from the robot. Using the ballistic, kinematic, trajectory, and dynamics models of the robot system, a set of joint commands was calculated using Equation (2.11). ($\text{Aim}_0$ and $\text{hit}_0$ are each defined to be the target.) The robot threw the ball with this set of torque commands, resulting in a hit of 6.03 m in the target plane. Based on the performance error of 0.28 m, the aim was modified to 5.47 m using Equation (2.10). Once again, the models of the robot system described in Equation (2.11) were used to calculate a new sequence of joint commands, and the ball was thrown again. The ball hit at 5.97 m, a new aim was calculated to be 5.25 m, and the model of ball throwing was applied again to calculate joint commands. This iterative learning procedure continued until the robot successfully completed the task. The aim for the successful throw was 4.78 m, almost one meter closer than the target. The open boxes connected by a dashed line in Figure 2.3 show how errors in performance were reduced with practice. The open boxes in Figure 2.4 show how the sequence of aims converged to 4.78 m in the target plane during the learning process. By applying fixed-model learning, the robot system successfully performed the task on the eighth iteration.

It is important to generalize the fixed-model learning procedure so that it can be applied to other tasks. The system task command is the aim, labeled with the vector **aim**, and the task performance is the hit, labeled with the vector **hit**. Generalizing the error correction equation to a multi-dimensional task, we write

$$\mathbf{aim}_{n+1} = \mathbf{aim}_n - (\mathbf{hit}_n - \mathbf{target}) \tag{2.12}$$

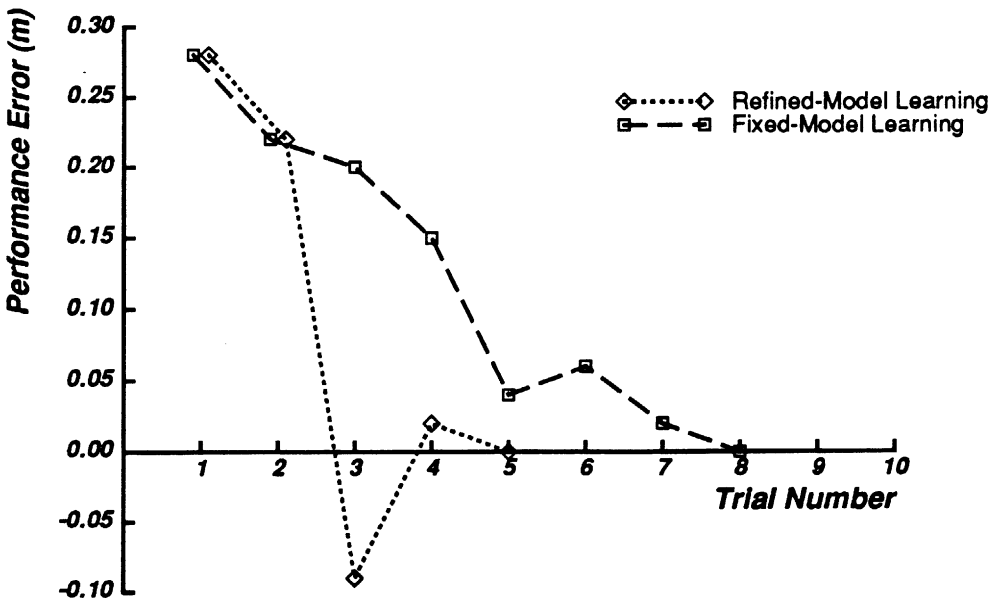


Figure 2.3: Learning Performance

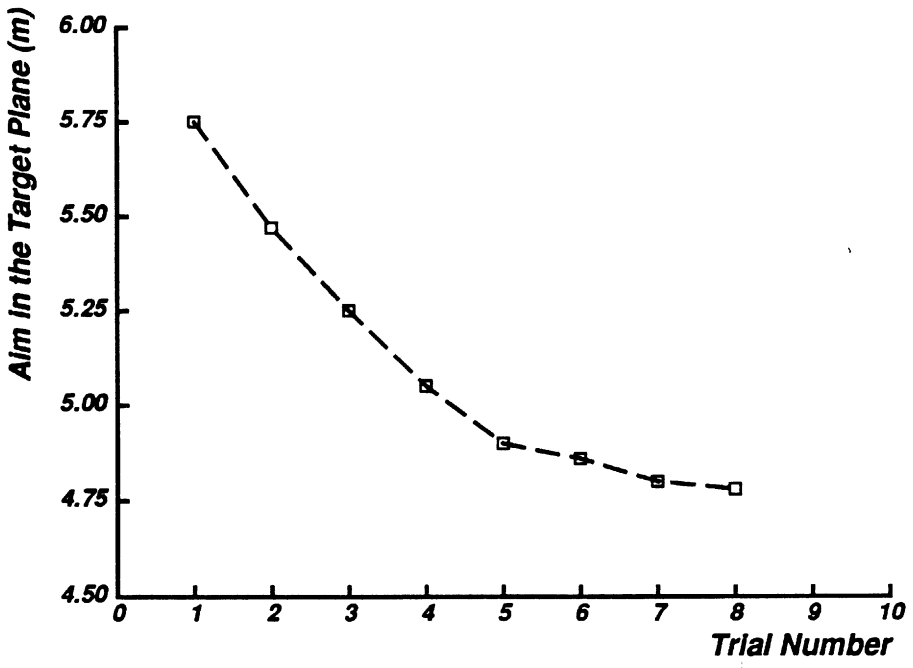


Figure 2.4: Convergence of the Aim

where **target** is the desired system performance. We also extend the task model equation to

$$\mathbf{hit_{n+1}} = \mathbf{S}(\hat{\mathbf{S}}^{-1}(\mathbf{aim_{n+1}})) \tag{2.13}$$

Fixed-model learning has successfully been applied to a multi-dimensional task in the case of trajectory following [Atkeson and McIntyre 1986]. A similar approach has also been developed for the task of kinematic positioning at a visual target [Atkeson et al. 1987].

The convergence of fixed-model learning depends on how accurately the model describes the behavior of the system. The convergence criteria can be derived by using fixed point theory [Wang 1984; Wang and Horowitz 1985]. A learning algorithm can be viewed as a mapping of aims on the $n$th attempt to aims on the next attempt

$$\mathbf{aim_{n+1}} = \mathbf{F}(\mathbf{aim_n}) \tag{2.14}$$

The fixed-model learning algorithm can be put into this form by substituting Equation (2.13) into Equation (2.12). Fixed-model learning modifies the $n$th aim by adding an amount based on the aim transformed by the task model

$$\mathbf{aim_{n+1}} = \mathbf{aim_n} - \mathbf{S}(\hat{\mathbf{S}}^{-1}(\mathbf{aim_n})) + \mathbf{target} \tag{2.15}$$

Note that when the correct hit, $\mathbf{hit}^*$, is achieved by using the correct aim, $\mathbf{aim}^*$, then $\mathbf{hit}^* = \mathbf{S}(\hat{\mathbf{S}}^{-1}(\mathbf{aim}^*))$. In this case, $\mathbf{target} = \mathbf{S}(\hat{\mathbf{S}}^{-1}(\mathbf{aim}^*))$, and Equation (2.15) reduces to the fixed point $\mathbf{aim_{n+1}} = \mathbf{aim_n} = \mathbf{aim}^*$.

We can ask whether this fixed point is stable by analyzing a linearization of Equation (2.15) at the point $(\mathbf{aim}, \mathbf{hit}) = (\mathbf{aim}^*, \mathbf{target})$. We begin by writing an equation for small perturbations around the fixed point. For a perturbation $\delta\mathbf{aim}$ from the fixed point,

$$\mathbf{S}(\hat{\mathbf{S}}^{-1}(\mathbf{aim}^* + \delta\mathbf{aim})) = \mathbf{S}(\hat{\mathbf{S}}^{-1}(\mathbf{aim}^*)) + \mathbf{J}\hat{\mathbf{J}}^{-1}\delta\mathbf{aim} \tag{2.16}$$

where $\mathbf{J}$ is the Jacobian matrix for the system transformation $\mathbf{S}()$, $\hat{\mathbf{J}}$ is the Jacobian matrix for the model $\hat{\mathbf{S}}()$, and $\hat{\mathbf{J}}^{-1}$ is the Jacobian matrix for the inverse model $\hat{\mathbf{S}}^{-1}()$. $\mathbf{J}$, $\mathbf{S}()$, $\hat{\mathbf{J}}^{-1}$, and $\hat{\mathbf{S}}^{-1}()$ are all evaluated at the fixed point. To analyze the fixed point for stability, we consider the case in which the $n$th aim is perturbed from $\mathbf{aim}^*$ by $\delta\mathbf{aim_n}$ so that $\mathbf{aim_n} = \mathbf{aim}^* + \delta\mathbf{aim_n}$. The change in the aim, $\delta\mathbf{aim_{n+1}} = \mathbf{aim_{n+1}} - \mathbf{aim}^*$, can be computed by substituting Equations (2.16) into Equation (2.15), and obtaining

$$\delta\mathbf{aim_{n+1}} = (\mathbf{I} - \mathbf{J}\hat{\mathbf{J}}^{-1})\delta\mathbf{aim_n} \tag{2.17}$$

The matrix $(\mathbf{I} - \mathbf{J}\hat{\mathbf{J}}^{-1})$ provides a necessary condition for convergence of fixed-model learning. When the task model is a *linear* function of the **aim**, the matrix $(\mathbf{I} - \mathbf{J}\hat{\mathbf{J}}^{-1})$ provides global convergence criteria. The error in the aim, $\delta\mathbf{aim}$, will decrease when all the eigenvalues of the matrix $(\mathbf{I} - \mathbf{J}\hat{\mathbf{J}}^{-1})$ are less than one

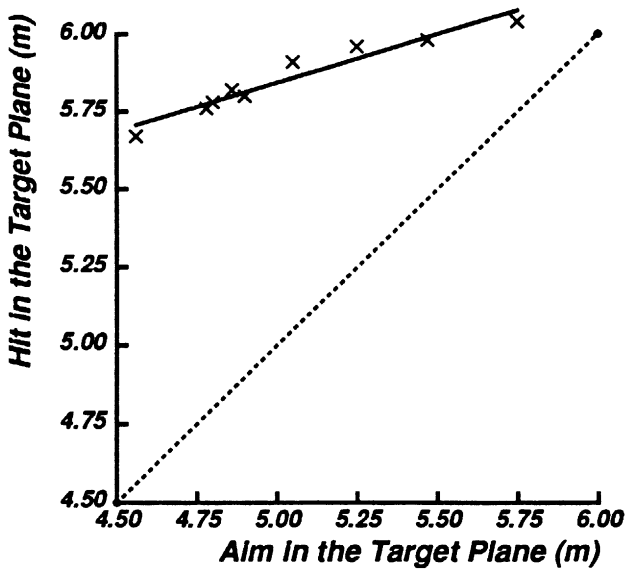


Figure 2.5: Aim/Hit Behavior of the Task Model

in absolute value, with the rate of decrease determined by the magnitude of the eigenvalues. If the magnitudes of all the eigenvalues are less than one, the learning process is stable and performance improves with practice. The magnitude of the eigenvalues of $(\mathbf{I} - \mathbf{J}\hat{\mathbf{J}}^{-1})$ depends on how accurately $\hat{\mathbf{J}}^{-1}$ inverts $\mathbf{J}$, and thus the stability of the learning algorithm depends on how closely the model inverts the controlled system. Thus, better modeling improves the stability and the speed of the learning process.

In the general case where the task model is a *non-linear* function of the **aim**, it is difficult to develop global convergence criteria. The criteria developed for the linear case can be applied locally, however, as a necessary but not sufficient condition for convergence. Thus, all the eigenvalues of the matrix $(\mathbf{I} - \mathbf{J}\hat{\mathbf{J}}^{-1})$ must be less than one in absolute value for learning to converge. If the magnitude of any eigenvalue is greater than one, fixed-model learning will almost certainly degrade performance. The better the model approximates the system, the closer the magnitudes will be to zero, and the more likely learning is to converge.

We applied this local convergence criteria to the ball throwing system. A plot of the aim/hit behavior of the task model is shown in Figure 2.5. The aims and hits are measured along the target plane. The data for the plot was experimentally determined by commanding the robot system with a number of different aims, and recording the landing position of the ball. Note that in the ideal case, when the inverse model perfectly describes the system, the task model reduces to the identity transform, producing the dotted line on the plot. The task model was fit by a linear function: the quantity $(\mathbf{J}\hat{\mathbf{J}}^{-1})$ was estimated to be 0.31. The eigenvalue

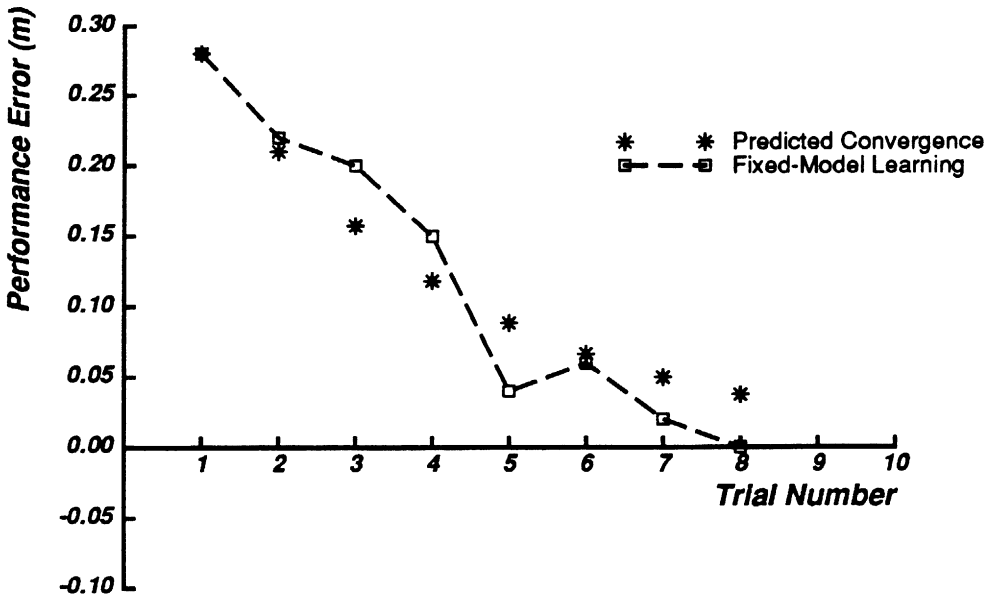


Figure 2.6: Convergence of Fixed-Model Learning

of $(\mathbf{I} - \mathbf{J}\hat{\mathbf{J}}^{-1})$ for this task model was then calculated to be 0.69. This value is less than one, indicating that the ball throws are likely to converge to the target. The open boxes in Figure 2.3 demonstrate that the ball throws did in fact converge.

The *performance* of the learning procedure refers to the *rate* of convergence. The geometric rate of convergence is given by the magnitudes of the eigenvalues of $(\mathbf{I} - \mathbf{J}\hat{\mathbf{J}}^{-1})$. The best performance is achieved when all the eigenvalues are close to zero. In the ball throwing task, for example, the geometric rate of convergence was calculated to be 0.69. The stars in Figure 2.6 illustrate this theoretical rate of convergence. The actual iterations of fixed-model learning, denoted by open boxes, closely approximate this prediction. If the model is made more accurate, the eigenvalues of the matrix $(\mathbf{I} - \mathbf{J}\hat{\mathbf{J}}^{-1})$ approach zero leading to faster convergence, improved learning, and better noise rejection.

### 2.5.2 Refined-Model Learning

Refined-model learning refines the task model as well as the aim during practice. The refined-model approach constructs a local linear model of the system from the last $m + 1$ attempts at the task, given a system of $m$ inputs and $m$ outputs. Thus, this method is an alternative to fixed-model learning only after the $(m + 1)$th iteration. Once in use, refined-model learning sacrifices the original model structure for a simpler local model. This local model is updated after each attempt at the task, leading to a refined system command.

In ball throwing, refined-model learning was applied after two attempts at the task. In order to easily implement refined model learning, a scalar quantity was necessary to characterize the robot command. We chose trajectory duration because the time-length of the throwing motion directly affected the release velocity, which in turn affected the distance the ball was thrown. As in our previous experiments, the first throw with a trajectory duration of 138 ms resulted in a performance error of 0.28 m. The second throw with a 143 ms duration resulted in a performance error of 0.22 m. Refined-model learning linearly extrapolated between these two points, suggesting a throw with a trajectory duration of 160 ms. Given the performance error, $e_n$

$$e_n = \text{hit}_n - \text{target} \tag{2.18}$$

the iteration rule for refined-model learning is

$$\text{command}_{n+1} = \text{command}_n - \frac{e_n}{(e_{n-1} - e_n)/(\text{command}_{n-1} - \text{command}_n)} \tag{2.19}$$

The results of refined-model learning for the ball throwing task are given by the diamonds in Figure 2.3. The desired performance was reached in just five iterations.

Refined-model learning can be generalized to multi-dimensional tasks. We first define the performance error, $\mathbf{e}_n$, to be the difference between the hit and the target on the $n$th iteration

$$\mathbf{e}_n = \mathbf{hit}_n - \mathbf{target} \tag{2.20}$$

We next define $\Delta\mathbf{C}$ and $\Delta\mathbf{E}$ to be $m \times m$ matrices

$$\Delta\mathbf{C} = [\ \mathbf{c}_0 - \mathbf{c}_n \quad \mathbf{c}_1 - \mathbf{c}_n \quad \cdots \quad \mathbf{c}_{n-1} - \mathbf{c}_n\ ] \tag{2.21}$$

$$\Delta\mathbf{E} = [\ \mathbf{e}_0 - \mathbf{e}_n \quad \mathbf{e}_1 - \mathbf{e}_n \quad \cdots \quad \mathbf{e}_{n-1} - \mathbf{e}_n\ ] \tag{2.22}$$

where $\mathbf{c}$ denotes the command. The general refined-model learning equation is then written as

$$\mathbf{c}_{n+1} = \mathbf{c}_n - \Delta\mathbf{C}(\Delta\mathbf{E})^{-1}\mathbf{e}_n \tag{2.23}$$

Refined-model learning in this form is similar to the secant method of finding zeros of functions [Gragg and Stewart 1976]. Improvements for avoiding singularities and for hastening convergence are described in the numerical methods literature and can be readily applied. Our primary interest here is to propose general learning procedures which can later be refined if they appear promising.

Refined-model learning will converge only if the first attempts at the task are sufficiently near the desired performance and if the system function is sufficiently smooth in that neighborhood. Because of these two restrictions, a principled and conservative approach to extrapolation should be taken. In the case of a one-dimensional system, for example, it might be useful to set a limit as to how far to

extrapolate. It might also be wise to interpolate instead of extrapolating as soon as a point on both sides of the desired performance is found [Press et al. 1986].

The *performance* of refined-model learning depends primarily on the command/performance behavior of the system. The performance on the first learning iteration also depends on the accuracy of the internal model. A better model makes the original performance error smaller, making learning faster. In the case of a one-dimensional system, the performance error, $e_n$, will decay superlinearly. This convergence rate is faster than for fixed-model learning, which only converges geometrically [Forsythe, Malcolm, and Moler 1977]. Figure 2.3 demonstrates this performance advantage in the throwing task.

## 2.6 Discussion

We have demonstrated that task-level learning improves a robot's performance of a fairly simple task—ball throwing. After each throw, task-level commands are adjusted to reduce the system's performance errors. Overall, we note four principal contributions.

The most important contribution is to demonstrate that task-level learning can take place by varying the *aim* of the ball-throwing system. The aim is varied by an amount equal to the performance error of the system, much as the person we observed varies his aim based on where the ball landed during the previous throw. The ball-throwing task model is used to interpret this new aim, and to calculate new commands to drive the robot system. We have demonstrated both experimentally and theoretically that task-level learning improves the performance of the ball-throwing system.

A second contribution is to demonstrate that learning can take place at the task-level without extensively calibrating the component models that describe the robot system. Learning improved the task-level performance of the ball-throwing system even though the ballistic, kinematic, trajectory, and dynamic models inaccurately described the system. For example, the desired trajectory of the robot arm was never followed perfectly, but the task was accomplished nonetheless. Figure 2.7 shows the desired and actual velocity trajectory of the final throw. That throw resulted in a perfect hit (zero error) even though the desired and actual trajectories differed. The difference is an indication that learning can proceed at the task level, even though lower level modules do not perform perfectly.

A third contribution is that the task model can be made more accurate while the system is learning a task. Refined-model learning takes this approach, as it builds a local linear model of the actual system to produce the next task command. This simple linear model replaces the task model and improves the speed of task-level learning on the ball-throwing task. Figure 2.3 shows that refined-model learning converged after five learning iterations, while fixed-model learning converged after eight iterations. Another potential approach is to build a local

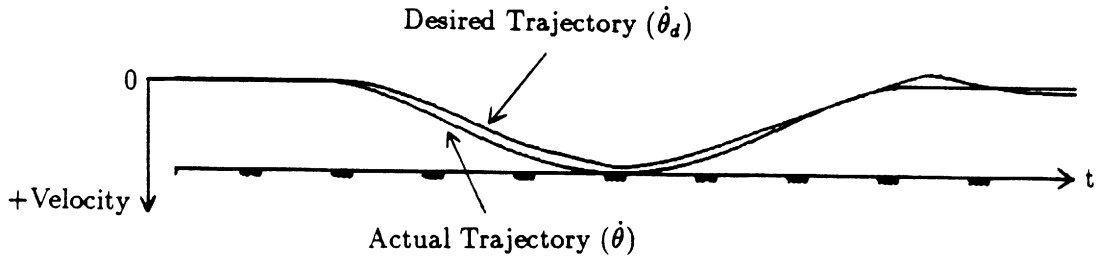


Figure 2.7: Actual and Desired Arm Trajectories for the Final Throw

model *on top* of the task model as the robot practices the task. Such a combined task model could more accurately describe the subtle characteristics of the system, and increase the stability and the speed of task-level learning even further.

A fourth contribution is to formalize an intuitive notion of a task model. The task model is a transformation that relates the desired performance to the actual performance of the task. The task model is composed of two parts: the system transformation and the inverse model of the system. In the ideal case, when the inverse model perfectly describes the system, the task model reduces to the identity transform and the system achieves the desired performance.

# Chapter 3

# Learning the Juggling Task

In this chapter, we discuss how to build a robot system that juggles more consistently with practice. We explain that the juggling task provides a rich domain in which to test our task-level learning algorithms. We describe the characteristics of the juggling task and a robot system that achieves this task. We explain how to apply the learning algorithms described in Chapter 2 to improve the performance of a juggling robot. Finally, we provide experimental results of a robot system that improves its juggling performance with practice.

## 3.1 Introduction

In the juggling task, the robot bounces a platform tennis ball on a paddle. The task is to repeatedly hit the ball with the paddle, bouncing it up into the air much as a person can do with a tennis racquet and ball. The robot system monitors the trajectory of the ball in the air. Based on an estimated trajectory, the robot calculates how to hit the ball upwards and back to the center of the paddle. The robot should be able to perform the task indefinitely.

The first major reason for choosing the juggling task is its multi-dimensional nature. The task goal is three dimensional: (1) to hit the ball to a specified height, (2) for the ball to land at a position $x$ on the paddle, and (3) for the ball to land at a position $y$ on the paddle. This multi-dimensional goal is in contrast to the one-dimensional goal in the robot throwing experiments. Juggling gives us the opportunity to test a multi-dimensional form of the task-level learning algorithms developed at the end of Chapter 2.

A second reason to study the juggling task is the complexity of the model that describes the robot juggling system. The juggling model is made up of many component models, including vision, forward ballistics, inverse ballistics, restitution, kinematics, trajectory following, and dynamics. Ball throwing required three models, but juggling requires as many as eight component models. The sheer number of models provides an excellent test for the model-based, task-level

learning algorithms that we have developed.

A third reason for working on the juggling task is to raise the issue of generalization. The learning algorithm developed in the throwing task applies directly to the problem of learning a single aim or hitting a ball from a single location. However, in the juggling task the ball falls to many *different* locations on the paddle. We are thus forced to consider two learning schemes. When arbitrarily generalizing, corrections in the aim that are learned when a ball falls to one paddle location are used when the ball falls to all other paddle locations. When selectively generalizing, the learning can be indexed according to a state-space, so that when the ball falls to similar locations the learning is generalized and when the ball falls to different locations no generalization occurs.

A fourth reason to focus on the juggling task is to examine the issue of training. In order to improve the system's juggling performance, the robot is trained in a sequence of three subtasks. The robot juggling system learns to perform the first hit, the second hit, and then the successive hits. Only after learning these subtasks, can the robot perform the juggling task. The need for this sequence of subtasks suggests that the training *process* is important in learning a complex task.

## 3.2 The Juggling Task

In this section, we describe the characteristics of the robot juggling system. The component vision and robot systems are described, as well as the models that are used to plan each hit in a juggling sequence. From these models, we explain how to build a task model of juggling. Model calibration procedures are outlined to underscore the need for accurate models. The repeatability of the robot juggling system is also analyzed in order to provide a more accurate indication of the system's capabilities.

The task is to bounce a ball at the end of the robot paddle. The task begins when a ball is dropped from the ceiling and falls towards the robot paddle. The task involves monitoring the flight of the ball, and estimating its downward trajectory. Based on the estimated landing location of the ball, a desired upward trajectory is calculated that returns the ball to the center of the paddle after the hit. Using these upward and downward trajectories, the robot system computes the necessary velocity at which to hit the ball. The robot must then calculate a paddle trajectory that will hit the ball with the correct velocity and the correct angle at the correct time. After the robot moves the paddle through this desired trajectory and hits the ball upwards, the sequence of monitoring the ball, calculating a response, and hitting the ball is repeated.

The task on the *first* hit is simplified by using an estimate of the trajectory of the ball gleaned from previous experimental data. This estimate is accurate because the ball is always dropped by a solenoid from the same location above

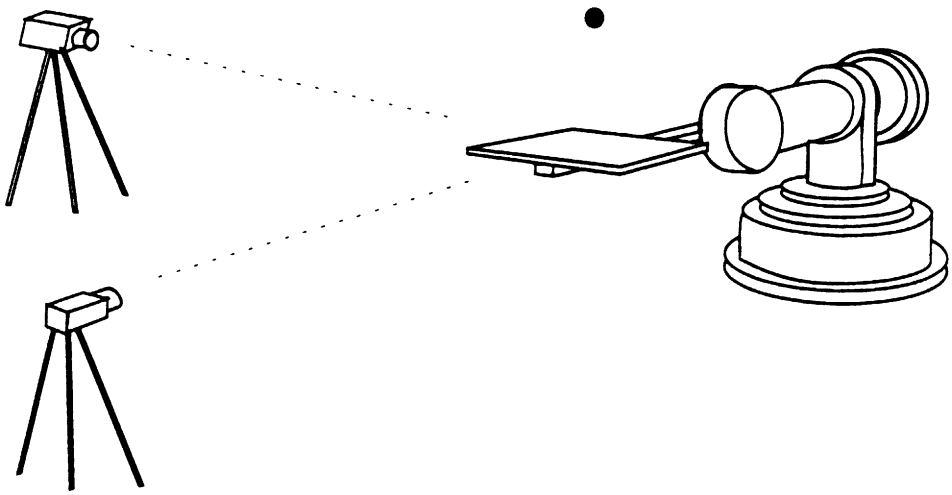

Figure 3.1: Juggling Robot/Vision System

the paddle. With this estimate, the vision system does not need to monitor the downward flight of the ball for the first hit of a juggling sequence. This simplification is necessary because there is not enough data to make an accurate estimate of the ball's trajectory before the robot is required to move. The vision system needs to track the ball moving both upwards and downwards to make an accurate estimate of the trajectory. Since the downward trajectories of the first ball is repeatable, using one prototype of the first downward trajectory allows us to apply the model-based learning algorithms with full generality on the first hit. All successive hits can be performed by monitoring the ball in real-time, computing a response, and executing a hit trajectory.

### 3.2.1 System Description

A paddle is attached to the Direct Drive Arm to hit the ball (Figure 3.1). The paddle is 0.46 m on a side, and its center is mounted 0.175 m from the joint two axis and 0.464 m from the joint three axis. The paddle is made of 0.02 m premium plywood which is mounted on five 0.025 m wide hollow aluminum studs that are laid across a 0.10 m wide hollow aluminum beam.

The task begins when a platform tennis ball is released by a solenoid from a distance of 1.5 m above the paddle. A vision system monitors the trajectory of the ball at 30 Hertz. Two video cameras (Sanyo VDC-3860) equipped with a half millisecond electronic shutter are mounted 5.0 m from the robot (Figure 3.1).

One camera is positioned directly in front of the robot paddle. A second camera is mounted to the right of the robot paddle, perpendicular to the first camera. Each camera has a field of view of 0.7 by 1.0 m, centered about the point $(0.0, 0.0, 1.0)$ in paddle coordinates. To simplify the vision problem, the platform tennis balls appears white on a background of black. The robot is also never in the field of view of the cameras.

The analog signal from each camera is processed by a Datacube vision system with the aid of a Sun host computer. The RS-170 signal from each camera is digitized independently using Datacube hardware. A threshold is applied to the digitized output and the $x, y$ pixel locations of each bright pixel is written into memory. The Sun computer is used to average the $x, y$ locations of the pixel values. This information provides the $x, y$ centroid of the tennis ball in each camera frame. With the Datacube system operating at frame rate, camera-space centroids from both cameras can be extracted every 33 ms.

Camera-space centroid information $(x_1, y_1, x_2, y_2)$ is sent over serial line to a 68020 microprocessor, running under the Condor real-time operating system [Narasimhan and Siegal 1987]. This microprocessor immediately associates a time to the centroid data to produce a full data point of the form $(x_1, y_1, x_2, y_2, t)$. The processor keeps track of the position of the ball using a simple finite-state machine. When a ball falls below the field of view of the cameras, the processor resets its centroid buffer. Once the ball reappears in the field of view, centroids are saved starting at the beginning of the centroid buffer. The processor also communicates directly with three other microprocessors that run high-level juggling code and control the robot arm.

Camera-space centroids are transformed to paddle coordinates by means of the *vision model.* The transformation assumes a simple orthographic projection model of image formation. Since the cameras are mounted perpendicular to one another, the $x, y, z, t$ centroid of a tennis ball in paddle coordinates is easily inferred from a full data point, $(x_1, y_1, x_2, y_2, t)$. A simple camera system calibration procedure determines the parameters that describe vision transformation. The calibration is discussed in Section 3.2.3.

Based on the centroids of the ball in flight generated by the real-time vision system, a *forward ballistics model* estimates the landing point of the ball on the paddle. The model performs a parabolic least-squares fit of the height vs. time ($z$ vs. $t$) centroid data. With the resulting parabolic trajectory, the model determines the landing time of the ball. The ball is assumed to land when it crosses plane $z = 0.0$, the height of the paddle. (This assumption is not completely accurate, and will be handled in the discussion of the *angle-time offset model.*) The forward ballistics model also does a linear least-squares fit of the $x$ vs. $t$ and the $y$ vs. $t$ centroid data. Based on the landing time of the ball, the $x$ and $y$ position on impact is estimated. From these three fits, the velocity of the ball at impact is also computed. The least-squares fits in the forward ballistics model are always

made with at least six centroids to improve the reliability of the fit. A new fit is made each time another centroid is acquired. For a ball that travels to about 1.0 m in height, between 11 and 12 centroids are generally available before the robot arm must begin to move. The last centroid is acquired when the ball is at a height of 0.70 m and falling towards the paddle.

Based on the task goal and the estimated landing position of the ball, an *inverse ballistics model* computes the desired outgoing trajectory that the ball should follow. This trajectory assumes a simple ballistic model to represent the flight of the ball

$$x_f = x_i + \dot{x}_i \cdot t \tag{3.1}$$
$$y_f = y_i + \dot{y}_i \cdot t \tag{3.2}$$
$$z_f = z_i + \dot{z}_i \cdot t - 1/2 \cdot g \cdot t^2 \tag{3.3}$$

The outgoing trajectory is fully determined by specifying the current landing position of the ball and the task goal. The estimate of the current landing position is given by the forward ballistics model. The task goal is a three-dimensional vector that describes the height, $z$, that the ball should reach, and the $x, y$ landing point on the next bounce. At the start of a juggling sequence, the task aim is to hit the ball to a height of 1.0 m and to have it land at the center of the paddle, $x, y = (0.0, 0.0)$. This three-dimensional task aim vector is varied during the learning experiments to improve the performance of the juggling robot system.

Once estimates of the incoming and desired outgoing trajectories of the ball are made using the forward and inverse ballistics models, a *restitution model* predicts the angle and velocity with which to hit the ball. The model assumes perfect angular restitution of the ball, predicting that the angle of incidence and reflectance of the ball with respect to the paddle are equal. The incident and reflectant angles are decomposed into rotations about the $x$ and $y$ axes of the paddle. The angles at which to hit the ball is computed by averaging the incident and reflectant component angles

$$\phi_x = (\phi_x^i + \phi_x^r)/2 \tag{3.4}$$
$$\phi_y = (\phi_y^i + \phi_y^r)/2 \tag{3.5}$$

where the angles are measured to the vertical. The restitution model also predicts the velocity with which to hit the ball. A simplified model [Beer and Johnston 1977] assumes that the relative velocity of ball and paddle before and after the hit is proportional

$$v_{paddle}^r - v_{ball}^r = e \cdot (v_{ball}^i - v_{paddle}^i) \tag{3.6}$$

The proportionality factor, $e$, is termed the coefficient of restitution. The coefficient of restitution is determined experimentally by dropping the ball onto the stationary paddle, and measuring the height of the bounce relative to the initial height.

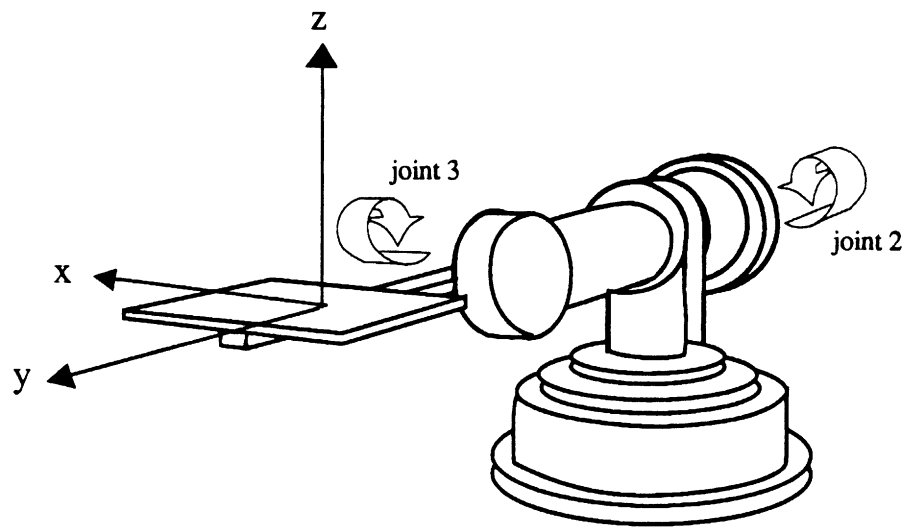


Figure 3.2: Juggling Robot

A *kinematics model* relates the desired angle and velocity of the paddle to the joint angles and velocities of the robot arm. Joints two and three of the MIT Serial Link Direct Drive Arm [An, Atkeson, and Hollerbach 1988] are used to hit the ball (Figure 3.2). Joint one is held with a brake in a fixed position during the juggling experiments. Joint two provides rotation about the $y$ axis, allowing the robot to hit the ball in the $x$ direction. Joint three provides rotation about an axis parallel to the $x$ axis and serves two functions. The angular position of joint three is used to hit the ball in the $y$ direction, returning the ball towards the center. The velocity of joint three is used to impart an upward velocity to the ball. The kinematics model relates the hit angles to robot joint angles by using a simple offset. This offset is determined experimentally as described in the calibration subsection. The kinematics model calculates the angular velocity of joint three based on the desired hit velocity and an estimate of the hit location.

An *angle-time offset model* is necessary to correct the discrepancy between the forward ballistics and kinematics models. The forward ballistics model calculates the time at which the ball crosses the horizontal plane $z = 0.0$, the height of the level paddle. The kinematics model computes the angle *away* from the horizontal at which to hit the ball. An inconsistency occurs because the timing is estimated independently of the robot kinematics and because the kinematics of the manipulator couples movement in the third joint with a change in paddle position. The angle-time offset model takes this kinematic-timing behavior into account. Based on the angle at which the ball will be hit, the model estimates the height at which

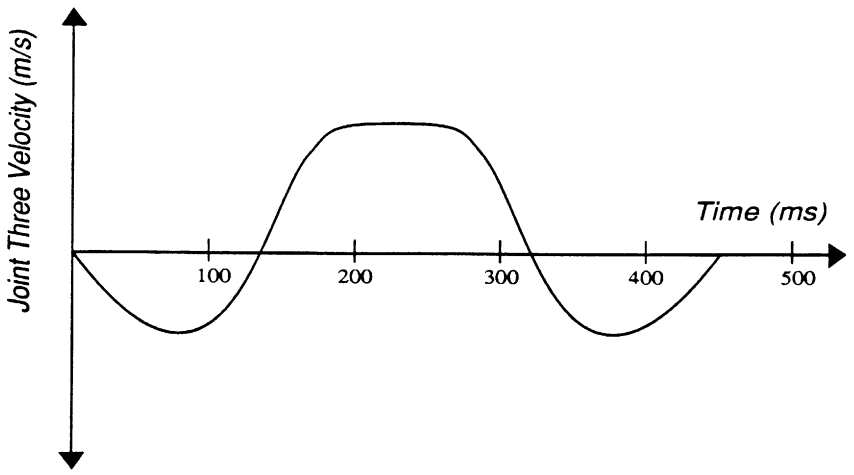


Figure 3.3: A Typical Juggling Trajectory (Joint Three)

contact with the ball will be made. The model uses the forward ballistics data to estimate the velocity of the ball in this region. A time offset is calculated that represents the travel time of the ball between the actual hit position and the level paddle position. This offset is added to the estimated time at which to hit the ball.

A *trajectory model* computes the joint trajectories that bring the paddle to the desired angle and velocity for each hit. The model includes three phases—speed-up, constant velocity, and slow-down—for the trajectories of joints two and three (Figure 3.3). In the speed-up trajectory, which lasts 200 ms, both joints start at the level position. Joint two executes a fifth-order polynomial starting from $\theta_2^{level}$ to $\theta_2^{hit}$. Joint three follows a fifth-order polynomial from $\theta_3^{level}$ to $(\theta_3^{hit} - 0.025 \cdot \dot{\theta}_3^{hit})$, ending with a velocity of $\dot{\theta}_3^{hit}$. In the constant velocity phase, which lasts 50 ms, joint two is servoed to the hit angle, $\theta_2^{hit}$. Joint three moves at constant velocity $\dot{\theta}_3^{hit}$, crossing the angle $\theta_3^{hit}$ precisely mid-way through the trajectory. In the slow-down phase, which lasts 200 ms, joint two executes another fifth-order polynomial trajectory starting from $\theta_2^{hit}$ back to $\theta_2^{level}$. Joint three executes a fifth-order polynomial trajectory starting from $(\theta_3^{hit} + 0.025 \cdot \dot{\theta}_3^{hit})$ with initial velocity $\dot{\theta}_3^{hit}$, and ending at the paddle level position, $\theta_3^{level}$. We chose this particular sequence of trajectories as a way to increase the probability that the paddle would hit the ball with the desired angle and velocity.

A *dynamics model* and feedback controller are used to accurately drive the robot joints along the desired trajectory. The dynamics model compensates for the effects of gravity on joints two and three, as well as for the inertia of joint three. The inertia of joint two is not taken into account because the joint motion is small. The resulting feedforward torques are calculated based on the acceleration profiles of the hit trajectories. A position–velocity feedback controller insures that the arm closely follows the desired trajectory on each hit. The control law that

includes both feedforward and feedback torques is

$$\mathbf{T} = \mathbf{T}_{\mathbf{ffwd}} - \mathbf{K_p} \cdot (\theta - \theta_\mathbf{d}) - \mathbf{K_v} \cdot (\dot{\theta} - \dot{\theta}_\mathbf{d}) \tag{3.7}$$

where $\mathbf{K_p}$ and $\mathbf{K_v}$ are the position and velocity feedback gains.

### 3.2.2 Task Model

With the component models developed in the last subsection, we can build a *task model* for each juggling hit. The task model relates the desired performance (**aim**) of the juggling system to the actual performance (**hit**)

$$\mathbf{hit} = \mathbf{S}(\hat{\mathbf{S}}^{-1}(\mathbf{aim})) \tag{3.8}$$

where $\mathbf{S}(\hat{\mathbf{S}}^{-1}())$ represents the task model of the juggling system. It is important to note that the task model involves *both* the juggling system and inverse juggling model transformations.

The **aim** and **hit** are specific to the robot juggling task that we described in the previous section. In our juggling task, the **aim** determines where we intend to hit the ball, and the **hit** represents where the ball actually lands. The **aim** is a three-dimensional quantity, with $x$, $y$, and $z$ components. The $x$ and $y$ components represent where the ball should land relative to the paddle center, and the $z$ component describes how high the ball should go. Similarly, the $x, y, z$ components of **hit** describe where the ball actually landed and how high it actually went. The **aim** and **hit** can be expressed as column vectors

$$\mathbf{aim} = [\mathrm{aim}_x,\ \mathrm{aim}_y,\ \mathrm{aim}_z]^T \tag{3.9}$$

$$\mathbf{hit} = [\mathrm{hit}_x,\ \mathrm{hit}_y,\ \mathrm{hit}_z]^T \tag{3.10}$$

where the symbol $^T$ denotes transpose.

The model and system transformations in the task model are also specific to our juggling system. First, we decompose the inverse juggling system model, $\hat{\mathbf{S}}^{-1}()$, into the component models described in the previous subsection. The synthesis of these component models forms the juggling model

$$\hat{\mathbf{S}}^{-1}() = \hat{\mathbf{D}}^{-1}(\hat{\mathbf{T}}^{-1}(\hat{\mathbf{A}}^{-1}(\hat{\mathbf{K}}^{-1}(\hat{\mathbf{R}}^{-1}(\hat{\mathbf{B}}^{-1}(\hat{\mathbf{F}}^{-1}(\hat{\mathbf{V}}^{-1}()))))))) \tag{3.11}$$

where $\hat{\mathbf{D}}^{-1}$ describes the dynamics model, $\hat{\mathbf{T}}^{-1}$ the trajectory model, $\hat{\mathbf{A}}^{-1}$ the angle-time offset model, $\hat{\mathbf{K}}^{-1}$ the kinematics model, $\hat{\mathbf{R}}^{-1}$ the restitution model, $\hat{\mathbf{B}}^{-1}$ the ballistics model, $\hat{\mathbf{F}}^{-1}$ the forward ballistics model, and $\hat{\mathbf{V}}^{-1}$ the vision model. Second, we include the system transformation for the juggling robot, $\mathbf{S}()$, in the task model equation and obtain

$$\mathbf{hit} = \mathbf{S}(\hat{\mathbf{D}}^{-1}(\hat{\mathbf{T}}^{-1}(\hat{\mathbf{A}}^{-1}(\hat{\mathbf{K}}^{-1}(\hat{\mathbf{R}}^{-1}(\hat{\mathbf{B}}^{-1}(\hat{\mathbf{F}}^{-1}(\hat{\mathbf{V}}^{-1}(\mathbf{aim}))))))))) \tag{3.12}$$

This task model describes the actions of the robot juggling system for each hit.

In the ideal case, Equation 3.12 reduces to the identity transformation because the model accurately describes, or inverts, the system. Unfortunately, the juggling system, like many other robot systems, is difficult to model accurately. In reality, the model *approximates* the system, and Equation 3.12 only approximates the identity transformation. Just how accurately the task model resembles the identity transformation is discussed in Section 3.3.1.

### 3.2.3 System Calibration

To accurately describe the robot juggling system, we need to calibrate the parameters of our structural models. The calibration of these models is essential for two reasons. First, accurate models provide information on how to command the system to achieve a particular result. With accurate models of the component systems, we can control the juggling system more precisely. Second, accurate models improve the performance of our task-level learning algorithms. With accurate models, learning becomes more effective in correcting errors and therefore faster and more stable.

The seven component models described above were calibrated before any juggling experiments were performed. Two of these models—kinematics and vision—are particularly sensitive to error. These models are calibrated at the start of each juggling session. Other models must be calibrated just once in order to achieve good system performance over time.

The first model to be calibrated is the *kinematics model.* We calculate the joint offsets that are necessary to bring the paddle to the horizontal position. This calibration is performed by moving the paddle to a horizontal position and reading the joint resolvers.

Once the position of the level paddle is found, the *vision model* is calibrated. This transformation is obtained by calibrating the cameras directly to the paddle coordinate frame. In the calibration procedure, a black pole, with a tennis ball mounted at the top, is attached to the paddle. The distance between the center of the paddle and the tennis ball is accurately measured. The robot moves this calibration pole to ten points within the field of view of each camera. Data is collected that includes centroids of the ball in camera coordinates and estimates of the ball location based on the robot kinematics model. The robot is then fitted with another calibration pole of a different length. Again, the robot moves this pole to ten points within the field of view of each camera while centroids and position estimates are calculated. Two calibration poles are necessary to adequately sweep out a large part of the camera space.

The vision model is defined by the following equations that transform from camera pixel coordinates to paddle coordinates

$$x_{world} = \mathrm{m}_x \cdot y_{camera\,1} + \mathrm{b}_x \tag{3.13}$$

$$y_{world} = \mathrm{m}_y \cdot y_{camera\,2} + \mathrm{b}_y \tag{3.14}$$

$$z_{world} = \mathrm{m}_z \cdot x_{camera\,2} + \mathrm{b}_z \tag{3.15}$$

A least-squares procedure is run on the 20 data points to determine the values of $m_x$, $m_y$, $m_z$, $b_x$, $b_y$, and $b_z$. Each camera is rotated 90° about its optical axis so that the $x$ pixel coordinates axis is vertical. Also, the $x$ value of camera one, which is redundant with the $x$ value of camera two, is never used.

As a fine tuning mechanism, the *vision model* is further calibrated with the use of a plumb line. A plumb line is hung from the ceiling to fall exactly above the paddle center. A ball is attached to the line 0.50 m above the level paddle. The cameras are then used to calculate the centroid of the ball using the previously calibrated vision model. Any differences between the calculated centroid and the point (0.0, 0.0, 0.5) are noted. The constant offsets in the vision model are adjusted by these differences to reduce any error.

### 3.2.4 Modeling Errors

In this subsection, we describe the sources of modeling error in the robot juggling system. We point out the inaccuracies in some of the models that describe the robot system. Later, we will describe how task-level learning algorithms will improve the performance of the juggling robot without adjusting these inaccurate models. Now, our goal is to understand the capabilities of the juggling system, as well as our ability to control it. Although the issue of modeling error could be analyzed in great detail, we only provide an indication of the situation based on our six month long experience with the system.

Several models that describe the juggling robot are inaccurate structural and parametric representations of the system. The vision model makes several assumptions that are not valid in practice. First, the model assumes a simple orthographic projection model of image formation. The model should also include the effects of perspective projection. Second, the vision model assumes a linear transformation between camera coordinates and world coordinates. In practice, we should include nonlinearities caused by lens and imaging array distortion. These two assumptions cause position errors in the estimated centroids of up to 0.02 m based on the trajectory of the ball.

The ballistic models also add errors to the juggling system. First, the models propagates the position errors of the vision model. The computed trajectories are based on inaccurate estimates of ball location, causing inaccurate estimates of the landing position of the ball. These errors are not uniform throughout the camera field of view, but instead are related to the path of the ball. Second, the ballistics models rely on a simple representation of flight that is based solely on gravity. The effects of air resistance are not accounted for. Third, the ballistics models propagate any timing errors in the vision model. These timing errors can come from transmission delays on the serial line. When a juggling hit is based

on inaccurate timing information, the ball will be hit inaccurately. Timing errors as small as 5 ms can cause 0.15 m inaccuracies in the landing position of a ball. In each case, the ballistics models are required to estimate the landing time and location of the ball 275 ms before the ball will be hit. Extrapolating forward in time magnifies any modeling errors.

The restitution model is a fairly inaccurate representation of the contact between the paddle and the tennis ball. Unfortunately, the mechanics of a paddle hitting a ball are difficult to model. The restitution model used in the juggling system assumes a smooth and frictionless paddle and ball. The ball and paddle obviously do not fit this description. Based on this assumption, the model assumes perfect angular restitution. Furthermore, the model assumes a perfectly flat and uniformly stiff paddle. Our paddle is more rigid along the $y$ axes, and more flexible at the edges. The paddle is also slightly concave. Finally, the model does not account for the effects of a spinning ball.

The angle-time offset model is based on an approximation of the ball's velocity. The model assumes that the ball travels with constant velocity near the hitting plane. In fact the ball is accelerating in that region. The constant velocity approximation causes timing errors on the order of 1 ms. An iterative scheme could be implemented to better describe the coupling between kinematics, restitution angle, and forward ballistics.

The dynamics model only partially describes the trajectory-following behavior of the robot arm. The model assumes no friction or damping, and no mechanical coupling between the joints. A feedback controller is included in the system to reduce the effects of these assumptions. Experiments reveal that angular error is less than 1° and the velocity error is less than 0.5 rad/s near the point of contact with the ball. The errors vary with the trajectory, but tend to be repeatable for a given trajectory.

### 3.2.5 System Noise

In this subsection, we discuss the issue of noise—how repeatably the juggling system can hit a ball to a particular location. The repeatability problems arise from physical realities that we do not or are not able to model accurately.

As a measure of system repeatability, we performed several experiments. First, we repeatedly dropped a tennis ball from a solenoid mechanism onto a stationary paddle. Second, we repeatedly hit a tennis ball dropped onto a moving paddle. Third, we analyzed the ability of the system to hit the tennis ball on the second hit.

In our first experiments, we measured the position to which a tennis ball bounced after hitting the stationary paddle. The robot was turned off and the paddle was clamped to a horizontal position. The ball was released from a solenoid 1.5 m above the paddle. Based on our vision system, we concluded that the ball

repeatedly followed the same trajectory downward. We analyzed in detail the repeatability of the position to which the ball bounced. The standard deviation of the landing position in the $x$ and $y$ directions was $\sigma_x = 0.05$ m and $\sigma_y = 0.06$ m. The standard deviation of the maximum height attained by the ball was $\sigma_z = 0.01$ m. Relative to repeatability of the vision and forward ballistics model, these standard deviations were significant.

We performed similar experiments with the paddle moving. The robot was programmed to hit the ball straight upward. Fifteen trials were performed in which the paddle was commanded with the same trajectory. We were interested in the landing position of the ball on the bounce following the hit. The standard deviation of the landing position was $\sigma_x = 0.07$ m and $\sigma_y = 0.07$ m. The standard deviation of the maximum height attained by the ball was again $\sigma_z = 0.01$ m.

Finally, we analyzed the landing position of the ball after two hits. Once again, fifteen trials were performed. In this case, the standard deviation of the errors were larger than in the previous experiments. The standard deviations were $\sigma_x = 0.10$ m, $\sigma_y = 0.09$ m, and $\sigma_z = 0.03$ m. These standard deviations can be considered significant compared to the paddle size which is 0.46 m on a side.

Our goal in performing these experiments was to assess the repeatability of the contact between paddle and ball. Unfortunately, we do not understand the phenomenon of contact very well. Our intuition suggests that a part of the repeatability problem is caused by the platform tennis balls. The balls have a single seam that results from the molding process. This seam, and any non-uniformities in the rubber compound of the ball, could cause unrepeatable bounces. We chose the platform tennis balls over standard tennis balls because the standard tennis balls exhibited even larger repeatability errors. Racquet balls or hand balls have a smoother surface and might provide more consistent bounces.

These experiments demonstrate the level of noise in the juggling system. This noise prevents the robot from performing the juggling task without visual feedback for more than three or four consecutive hits. The juggling task cannot be performed "open loop." As a result, the visual feedback and the eight component models that describe the juggling robot are indispensable for hitting the ball many times in a row.

## 3.3 Learning the First Hit

In this section, we describe how to apply task-level learning algorithms to improve the robot system's performance of the first hit. We analyze the theoretical convergence properties of the fixed-model learning algorithms as they apply to the juggling task. We describe experiments in which the learning algorithms are successfully applied to the first hit. Finally, we provide an indication of the effect of system noise on the learning algorithms.

Learning the first hit is a good starting point for applying our task-level learning algorithms to the juggling task. First, we are using the complete set of models described in the previous section. Testing the task-level learning algorithms on a system with eight component models provides a good indication of the applicability of the learning techniques to complex tasks. Second, the first hit provides a straightforward way to test the success of learning a multi-dimensional task. As opposed to the throwing experiments that have only target height as the goal, the first juggling hit has three separate goals. The goal is: (1) to hit the ball to the height of 1.0 m, (2) for the ball to land at a position $x$ on the paddle, and (3) for the ball to land at a position $y$ on the paddle. Finally, the effect of noise on the task-level learning algorithms can be easily analyzed. Learning algorithms can be applied in the presence of noise, or while averaging out the effects of noise. With these experiments, we can get an indication of whether too much noise exists in the juggling system for learning to be successfully applied.

Learning the first hit is also a good staring point because the first hit task is simpler than the entire juggling task. The first hit is simpler because the ball always follows the same downward trajectory. The ball is dropped from the ceiling by a solenoid release mechanism and falls to the same location on the paddle, with the same velocity, and with the same flight time. Learning the entire juggling task will be more difficult because of the need for generalization. After the first hit, the ball will invariably follow *different* downward trajectories and land at different locations on the paddle. The learning that takes place at one location must then be *generalized* to another location. Analyzing the first hit allows us to temporarily ignore the issue of generalization, and instead to concentrate on understanding task-level learning.

### 3.3.1 Convergence Criteria

Before performing any juggling experiments, we examine whether fixed-model learning will theoretically converge for the juggling task. The predicted convergence is given by

$$\delta\mathbf{aim}_{n+1} = (\mathbf{I} - \mathbf{J}\hat{\mathbf{J}}^{-1})\delta\mathbf{aim}_n \tag{3.16}$$

which was derived as Equation 2.17 in Chapter 2. Fixed-model learning will converge when all the eigenvalues of the matrix $(\mathbf{I} - \mathbf{J}\hat{\mathbf{J}}^{-1})$ are less than one in absolute value. The magnitudes of the eigenvalues of $(\mathbf{I} - \mathbf{J}\hat{\mathbf{J}}^{-1})$ depend on how accurately $\hat{\mathbf{J}}^{-1}$ inverts $\mathbf{J}$, or how accurately the system is modeled. The better the model approximates the system, the closer the magnitudes will be to zero, and the more likely learning is to converge.

The eigenvalues of $(\mathbf{I} - \mathbf{J}\hat{\mathbf{J}}^{-1})$ can be determined experimentally by performing some simple experiments. The experiments involve analyzing the validity of the *task model* near the desired task performance. In other words, the experiments involve analyzing whether the juggling model accurately describes the robot system.

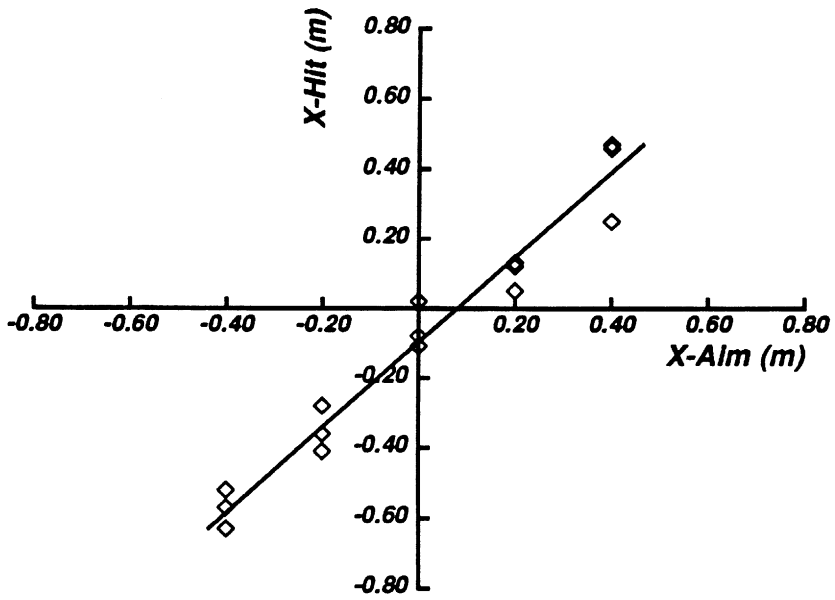


Figure 3.4: Task Model: X Aim/Hit

To analyze the juggling task model,

$$\mathbf{hit} = \mathbf{S}(\hat{\mathbf{S}}^{-1}(\mathbf{aim})) \tag{3.17}$$

we perform experiments that vary the desired performance, **aim**, and monitor the resulting actual performance, **hit**. Since our goal is to hit the ball to $x, y, z = (0.0, 0.0, 1.0)$, the **aim** is varied in this neighborhood. The inverse model of the system, $\hat{\mathbf{S}}^{-1}()$, is used to compute the necessary torque commands based on the prescribed **aim**. The commands are used to drive the juggling system, effectively passing robot commands through the **S()** transformation. After varying the task **aim** on a number of trials, we can plot the resulting **hit** versus **aim**. Since the juggling task is multi-dimensional, we begin by analyzing **aim/hit** behavior along the $x$, $y$, and $z$ axes. The graphs associated with the juggling task model are shown in Figures 3.4 , 3.5, and 3.6. Three hits were performed at each **aim** to show the noise in the system.

The matrix $(\mathbf{I} - \mathbf{J}\hat{\mathbf{J}}^{-1})$ is determined directly from this data. We simply analyze the *change* in hit as a result of the *change* in aim, since we can approximate Equation 2.13 or 3.17 by

$$\delta\mathbf{hit} = (\mathbf{J}\hat{\mathbf{J}}^{-1})\delta\mathbf{aim} \tag{3.18}$$

The diagonal terms of the matrix $(\mathbf{J}\hat{\mathbf{J}}^{-1})$ can be readily gleaned from Figures 3.4, 3.5, and 3.6 by simply measuring each slope. The off-diagonal terms are obtained by analyzing the change in one component of the hit with respect to a change in a

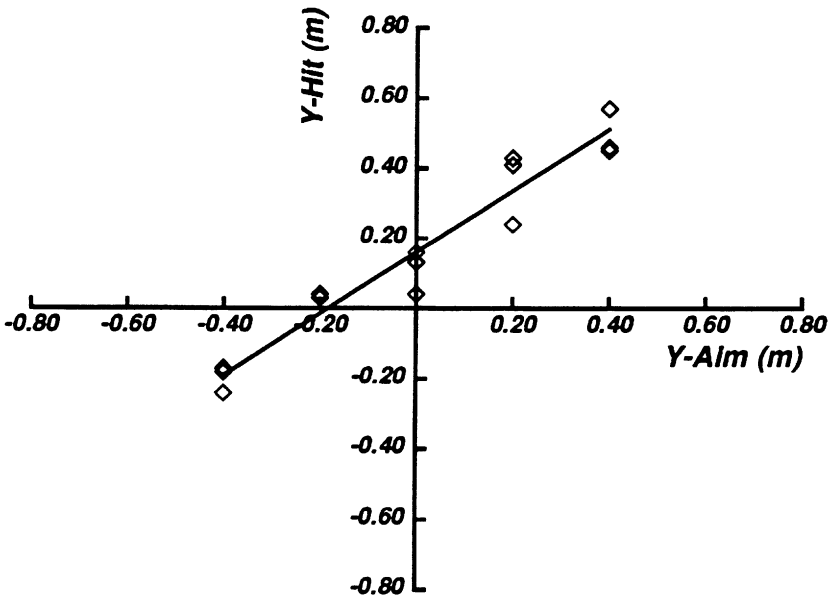


Figure 3.5: Task Model: Y Aim/Hit

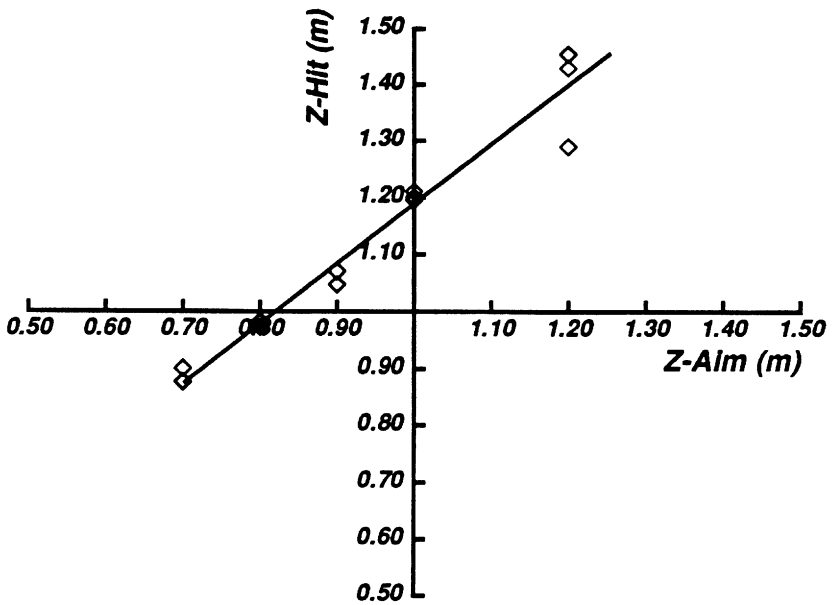


Figure 3.6: Task Model: Z Aim/Hit

*different* component of the aim. We have analyzed data for the off-diagonal terms and find small, noisy correlations. For our purposes, the matrix $(\mathbf{J}\hat{\mathbf{J}}^{-1})$ reduces to

$$\begin{bmatrix} 1.17 & -0.19 & 0.12 \\ 0.05 & 0.85 & -0.01 \\ 0.08 & -0.11 & 1.12 \end{bmatrix} \tag{3.19}$$

From this matrix, we can readily compute the eigenvalues of the convergence matrix $(\mathbf{I} - \mathbf{J}\hat{\mathbf{J}}^{-1})$ and obtain $\lambda_{1,2,3} = 0.22, -0.13, 0.04$. These eigenvalues are all less than one in absolute value, suggesting that fixed-model learning will converge for juggling hits in this neighborhood. Our analysis of convergence is only valid near the **aim** $= (0.0, 0.0, 1.0)$, however. To analyze convergence for a hit to the height of 2.0 m, we would have to do similar experiments to the ones described above in which the **aim** is varied near $(0.0, 0.0, 2.0)$. The slopes of the graphs could again be computed, the matrix elements identified, and the eigenvalues calculated. Similarly, the convergence criteria we derived experimentally is only valid for hitting the ball from the point $(0.0, 0.0, 0.0)$ on the paddle. If the ball is dropped to another point on the paddle, a new set of data must be examined to assess convergence in that neighborhood.

We conclude that learning will converge for this particular hit, but may or may not converge for hits that are significantly different. Since the ball is always dropped to the same position at the start of each juggling sequence, the learning algorithms should converge for the first hit. From the point of view of accurate modeling, we can conclude that our model describes the juggling system "well enough" for the first hit. Similarly, we can conclude that the juggling model has been calibrated "sufficiently" for the first hit.

### 3.3.2 Learning while Suppressing Noise

In our first experiments, we apply the fixed-model learning algorithms to the first juggling hit while suppressing the effect of system noise. Noise is suppressed by performing each hit five times and averaging the results. This averaging procedure reduces the effect of noise, allowing us to focus on the success of the task-level learning algorithms applied to this multi-dimensional task.

The first hit was successfully performed after four learning iterations. The target was to hit the ball to the point $(0.0, 0.0, 1.0)$. The learning sequence began with the juggling system aiming at $(0.0, 0.0, 1.0)$. Five trials were performed with this aim, and the resulting average hit was $(-0.26, 0.46, 1.21)$. The juggling robot consistently hit the ball forward, to the left, and too high. Based on the average hit, a new aim of $(0.26, -0.46, 0.79)$ was calculated using Equation 2.12. Once again, five trials with this aim were performed, and the results averaged to $(0.15, -0.23, 1.12)$. A new aim of $(0.11, -0.23, 0.67)$ was calculated, and five more hits were performed. The averaged results were $(-0.02, -0.07, 1.00)$. We

| Aim | Hit | Error |
|---|---|---|
| $(0.00, 0.00, 1.00)$ | $(-0.26, 0.46, 1.21)$ | $(-0.26, 0.46, 0.21)$ |
| $(0.26, -0.46, 0.79)$ | $(0.15, -0.23, 1.12)$ | $(0.15, -0.23, 0.12)$ |
| $(0.11, -0.23, 0.67)$ | $(-0.02, -0.07, 1.00)$ | $(-0.02, -0.07, 0.00)$ |
| $(0.13, -0.16, 0.67)$ | $(0.00, 0.01, 0.99)$ | $(0.00, 0.01, -0.01)$ |

Figure 3.7: Learning the First Aim (Noise Suppressed)

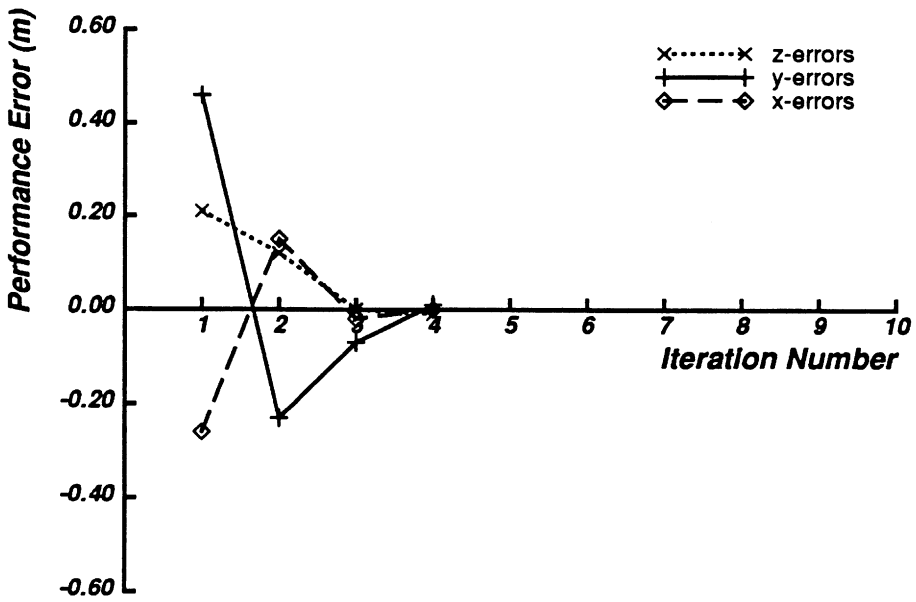


Figure 3.8: Learning the First Hit (Noise Suppressed)

applied fixed-model learning one more time with an aim of $(0.13, -0.16, 0.67)$, and the actual hits averaged out to $(0.00, 0.01, 0.99)$. At this point, the task was considered complete since the error in each hit component was less than one standard deviation from the target. The measured standard deviations in the task components were $\sigma_x = 0.07$ m, $\sigma_y = 0.07$ m, and $\sigma_z = 0.01$ m. Figure 3.7 shows the sequence of aims that was necessary to achieve the task, and Figure 3.8 shows the corresponding sequence of component $x, y, z$ errors.

The *rate* of convergence is different than estimated from our convergence data. Our data from Section 3.3.1 predicts a geometric rate of convergence of 0.22, suggesting that the errors should reduced by approximately 78% on each iteration. The actual geometric convergence rate more closely approximates 50% on each

iteration. Several reasons may explain this discrepancy. First, the convergence criteria derived using fixed point theory is only valid for linear modeling errors. The error between the inverse model, $\hat{\mathbf{S}}^{-1}()$, and the system transformation, $\mathbf{S}()$, is probably non-linear. A denser plot of the **aim/hit** behavior of the juggling task model would probably show the non-linear relationship. Second, the coupling terms of the matrix $(\mathbf{I} - \mathbf{J}\hat{\mathbf{J}}^{-1})$ will tend to slow convergence. While correlations for these coupling terms in our data were small, some coupling does exist. Although fixed-model learning effectively decouples the command corrections by transforming the new aim through the inverse system model, the decoupling is only as accurate as the model. As a result, changes in one component of the aim will affect *another* component of the hit.

The aim that was finally used to perform the first juggling hit successfully was $(0.13, -0.16, 0.67)$. The robot juggling system was aiming forward, to the left, and low in order to hit the ball to the desired target location. The difference between this aim vector and the target vector $(0.0, 0.0, 1.0)$ is an indication of the inaccuracy of the juggling model. However, with this modeling accuracy task-level learning successfully converges after four iterations.

### 3.3.3 Learning in the Presence of Noise

Additional experiments were performed to assess the success of fixed-model learning in the presence of noise. The experiments were designed to provide only an indication of the effect of noise, since a full assessment requires numerous trials and a detailed statistical analysis that are beyond the scope of this thesis. The goal in performing a limited number of experiments is to determine whether learning is possible with the amount of noise present in the juggling system. A secondary goal is to discover the possible ways to reduce the effect of noise on learning.

In contrast to the previous learning experiments of the first hit, the data was not averaged and no trials were repeated. Fixed-model learning was applied directly after each trial to provide an indication of the effect of noise on convergence. Three learning sequences are presented in Figures 3.9, 3.10, and 3.11. The plots show the component errors in the task goal versus the trial number. Learning converged in four trials, eight trials, and five trials, respectively. In each case, learning took more iterations (but fewer trials) to converge than in the previous experiments when noise was suppressed by averaging. The criteria for task completion was the same as in the previous experiments.

Our criteria for task convergence no longer guarantees that the learned aim will consistently lead to a successful hit. Noise in the system will allow a number of *different* aims to produce the same target hit. For example, the final aims for the three learning sequences in the presence of noise were $(0.11, -0.10, 0.71)$, $(0.10, -0.18, 0.69)$, and $(0.09, -0.19, 0.69)$. These aim vectors are not equal, and differ from the aim vector of $(0.13, -0.16, 0.67)$ learned during the trial averaging

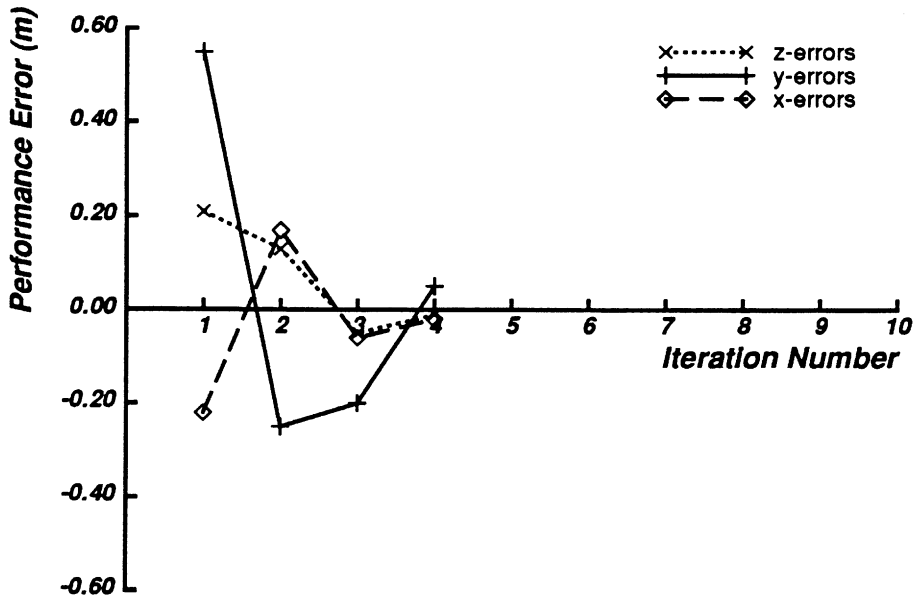


Figure 3.9: First Hit: Learning with Noise—Sequence 1

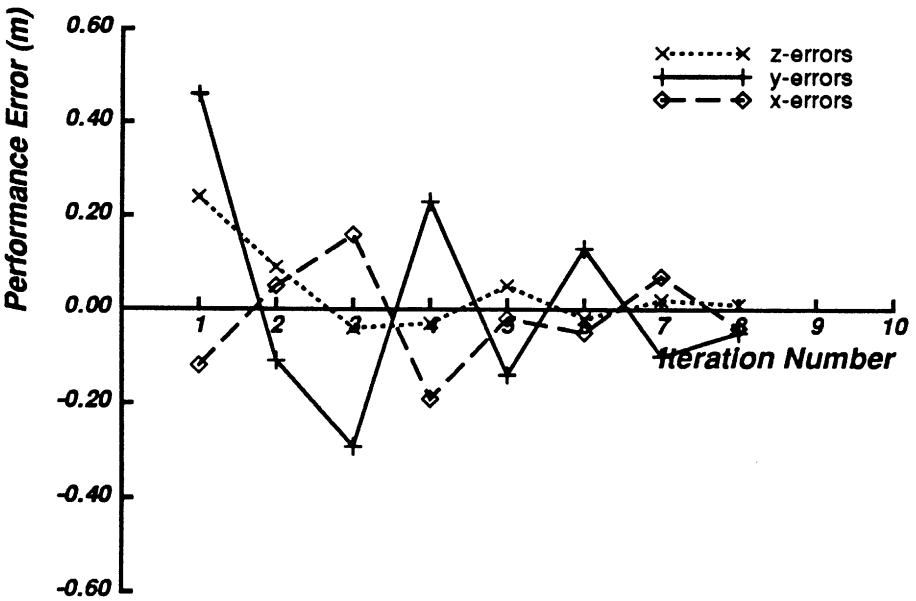


Figure 3.10: First Hit: Learning with Noise—Sequence 2

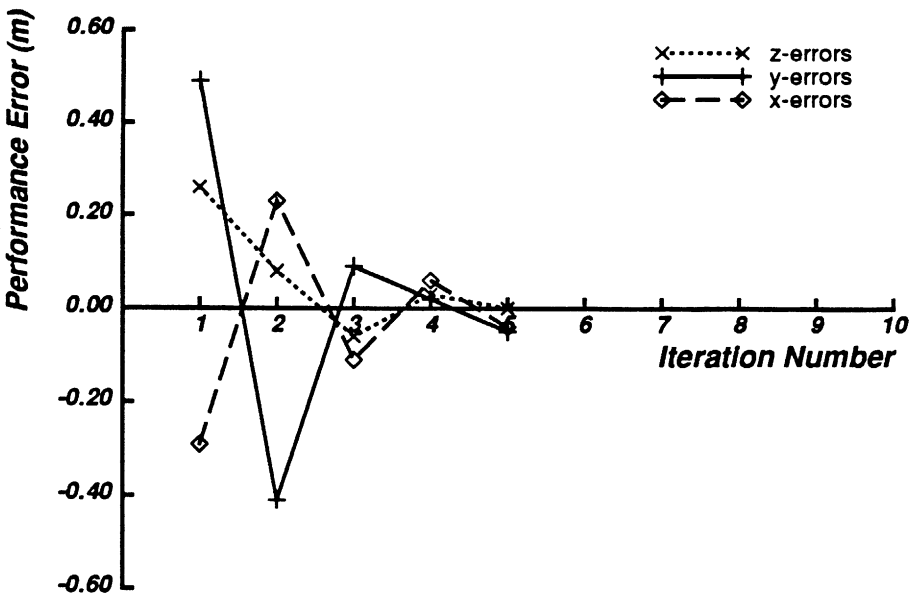


Figure 3.11: First Hit: Learning with Noise—Sequence 3

sequence. The difference in the aims is within the repeatability of the system. Some trial averaging or noise suppression seems necessary for the learned aim to converged to the desired one.

Noise in the system promotes oscillatory behavior in the learning sequence. As a result, task errors tend to be reduced more slowly in these three learning sequences. Applying learning to a system with noise sometimes increased and sometimes decreased the errors in performance. For example, system noise made the learning both converge and diverge between the second and third hits in the learning sequence described by Figure 3.10. Error in the $x$ and $y$ components increased, while error in the $z$ task component was reduced. Learning continued to oscillate after the third hit with the amplitude in the error decaying. The tentative conclusion to draw is that noise reduces the rate of convergence, increases the oscillatory nature of the learning algorithm, and prevents learning from converging beyond the noise level of the system.

## 3.4 Learning the Second Hit

In this section, we describe how the robot system learns the second hit with task-level learning algorithms. We begin with a discussion of why the robot needs to learn anything more than the first hit. We describe a training sequence that the robot effectively undergoes to perform the first and second hits. We present

experimental results of the robot juggler learning to perform the second hit. We finally describe the performance of the robot juggling system after learning the first and second hit.

Before running the juggling system past the first hit, we have to decide what aim to apply to the successive hits. First, the target aim can be applied directly. This approach assumes that the juggling model accurately describes the robot system. Second, the learned aim vector for the first hit can be applied to all the remaining hits. In this case, learning from the first hit is *generalized* to the second and successive hits.

With either of these approaches, the robot juggler is unsuccessful at hitting the ball more than several times. In the first approach, when the target aim vector of $(0.0, 0.0, 1.0)$ is applied after the first hit, we are assuming that the system is perfectly calibrated. Our data in Figures 3.4, 3.5, and 3.6 dispute this assumption, suggesting that the juggling model only approximates the system. Experiments using this approach result in the ball drifting off the paddle after several hits. In the second approach, when the aim vector of the first learned hit is applied, the robot is just as unsuccessful at hitting the ball more than several times. In this case, the assumption made is that learning from the first hit generalizes to the second hit. Unfortunately, the state, or operating point, of the task model differs for each hit and generalizing is not that simple. In the first hit, the ball is dropped from 1.5 m and reaches the paddle at approximately 1.5 m/s. For the second hit, the ball peaks at a height of 1.0 m, and reaches the paddle with a velocity of 1.0 m/s. The accuracy of the juggling model is *different* in each case. As a result, learned compensations in the aim for the 1.5 m drop are very different from learned compensations in the aim on the 1.0 m bounce.

Because neither of these approaches proved successful, we applied task level learning to the second hit. The second hit is learned much as the first hit was learned.

In choosing to improve the performance of the first hit and then the second hit, we are effectively "training" the robot. The juggling task is being decomposed into two simpler subtasks. Each juggling subtask is learned sequentially in order to improve performance of the entire task. In subsequent sections, a third subtask is learned before the robot system is capable of performing the juggling task.

### 3.4.1 Learning Experiments

Hitting the ball the second time is much like hitting the ball the first time. The ball should peak at approximately $z = 1.0$ m, and fall to the center of the paddle which is denoted by $x, y = (0.0, 0.0)$. The ball usually follows this trajectory, except for deviations caused by noise in the system. Deviations from the desired trajectory are monitored using visual feedback. Based on this data and the juggling model, the robot system computes the appropriate paddle trajectory that will hit the ball

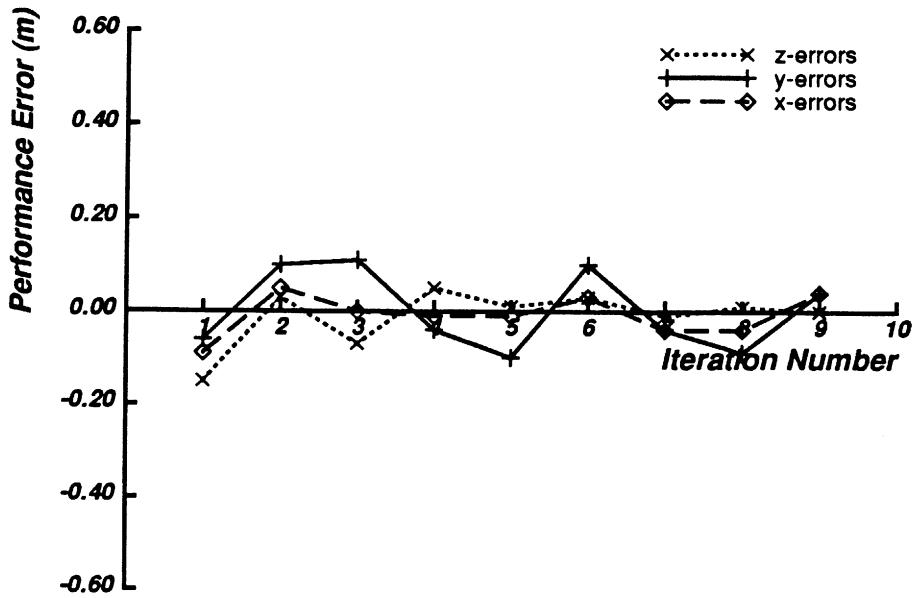


Figure 3.12: Learning the Second Hit

to the target. Once again the target is $x, y, z = (0.0, 0.0, 1.0)$.

We applied task-level learning to improve the robot's performance of the second hit. The robot was programmed to perform the first hit with the task aim that was learned in our previous set of experiments. This task aim was initially used on the second hit, also. This initialization was effectively a method of generalizing estimates of the inaccuracies in the task model, and applying them from the first hit to the second. As we explained above and as the data will show, this generalization was by no means a perfect one.

Task-level learning was initially applied in the presence of noise. However, when the robot was within one standard deviation of the hit, three trials were performed and averaged before making another learning iteration. With this method, we hoped to rapidly converge to the area around the desired aim. Once near the desired aim, three trials performed at each learned aim. This averaging procedure suppressed the effect of noise on the learning algorithm and made us more certain that the correct aim had been found.

Task-level learning successfully converged after the ninth iteration. A graph of the performance errors at each learning iteration is shown in Figure 3.12. Within seven iterations in the presence of noise, learning had converged to within one standard deviation ($\sigma_x = .07$, $\sigma_y = .07$, $\sigma_z = .01$). Three trials were averaged at each successive learning iteration. With this noise suppression approach, learning again converged to within one standard deviation of the error on the ninth iteration. The task aim that finally resulted in the successful hit was $(0.06, -0.12, 0.84)$.

### 3.4.2 Juggling Performance After Learning Hits One and Two

In this subsection, we ask the question: How well can the system juggle after learning the first and second hits? The aim necessary to generate a successful first hit was found in Section 3.3. The aim necessary to successfully perform the second hit was determined in Section 3.4. In the following experiments, the learned aim from the second hit is generalized to all successive hits, and the system was commanded to juggle until the ball strayed from the paddle.

We generalize the learned aim for the second hit to all successive hits. The aim learned for the second hit of $(0.06, -0.12, 0.84)$ is used as the aim for all successive hits. We command the successive hits with this aim, reasoning that the second hit is similar to the successive ones. This assumption is valid if the ball always returns to the center of the paddle. If the ball lands far from the center of the paddle, the hits are sufficiently different that the generalization may not be accurate. Nonetheless, this generalization is better than commanding the target aim $(0.0, 0.0, 1.0)$ which was shown to produce less than three or four hits in Section 3.4.

We performed juggling experiments using a learned aim for the first hit, a learned aim for the second hit, and the learned aim for the second hit generalized to successive hits. The robot juggling system performed 20 juggling sequences in this configuration. The robot averaged eight hits, with a low of three hits and a high of 23 hits. The performance of the robot could be characterized as erratic.

A visual analysis of the robot failures indicates that the ball oscillates back and forth in the $y$ direction on successive bounces, eventually landing out of reach. The robot tends to overcompensate along this direction. When the ball is hit forward of the center, the robot overreacts by hitting the ball too far back. If the ball is hit behind the center, the robot overcompensates, hitting the ball too far forward. These oscillations eventually cause the ball to stray from the paddle area.

The tentative conclusion we reach is that the command learned for the second hit *cannot* be arbitrarily generalized to all other hits. In other words, the learned aim for the second hit of $(0.06, -0.12, 0.84)$ should not be used as the aim for all successive hits. In particular, the aim learned for a ball falling to the paddle center differs from the command necessary to compensate for a ball that lands far forward of the center. The inaccuracy in the task model that was successfully learned on the second hit differs from the inaccuracy in the task model that exists for balls landing forward or backward of the paddle center. The aim must be learned based on the *location* the ball lands on the paddle.

## 3.5 Learning the Successive Hits

In this section, we apply task-level learning past the first and second hits of a juggling sequence. We first analyze the performance errors of the juggling system in order to understand why the robot only averages eight hits. We then describe how to apply task-level learning in order to further improve the juggling performance of the robot system. In doing so, we address the issues of generalization and task state in the context of the juggling system. Finally, we describe the performance of the juggling robot, which eventually averages 25 hits after task-level learning is applied.

### 3.5.1 Examining Performance Errors

We begin by examining the performance errors that occur during juggling. We analyze data from the 20 juggling trials which were described in Section 3.4.2. Our goal is to understand whether the characteristics of each hit indicate what performance errors are likely to occur.

We define three performance errors that correspond to the performance goals of the system. The performance errors are $x$ errors, $y$ errors, and $z$ errors. Each error is the measured distance between the target vector $(0.0, 0.0, 1.0)$ and the hit vector. The error is always measured to the target vector of $(0.0, 0.0, 1.0)$ because the juggling task specifies that the ball should always be hit to the height of 1.0 m and to the center of the paddle.

The performance errors of the juggling robot can be understood based on the characteristics of each hit. In order to interpret the error for each of the 160 juggling hits, we examine each performance error versus the $x$ and $y$ paddle location *from which* the ball was hit. The performance errors could also be analyzed with respect to the $x$, $y$, and $z$ velocity of the ball at the location from which it was hit. However, we chose to simplify our analysis from the start, using the state variables corresponding only to location of the ball, not velocity. In addition, a statistical analysis suggests that performance errors correlate most strongly to the $x$ and $y$ paddle location from which a ball is hit.

Since the performance data is intrinsically three-dimensional, we have used two methods to present the data. First, we display three error grids in Figure 3.13 that correspond to the the $x$, $y$, and $z$ performance errors of the juggling system. The paddle is divided up into a $4 \times 4$ grid, with a grid resolution at 0.12 m. Each grid is labeled with the $x$ and $y$ axes that correspond to the position from which the ball was hit. The grid axes are the same as the $x, y$ axes of the robot's paddle. The performance errors of all the hits that are made from one grid section are averaged together. The average error is displayed in the appropriate section in Figure 3.13. The grid is thus composed of average performance errors based on paddle hit location. The number of hits that occurred in each section is displayed

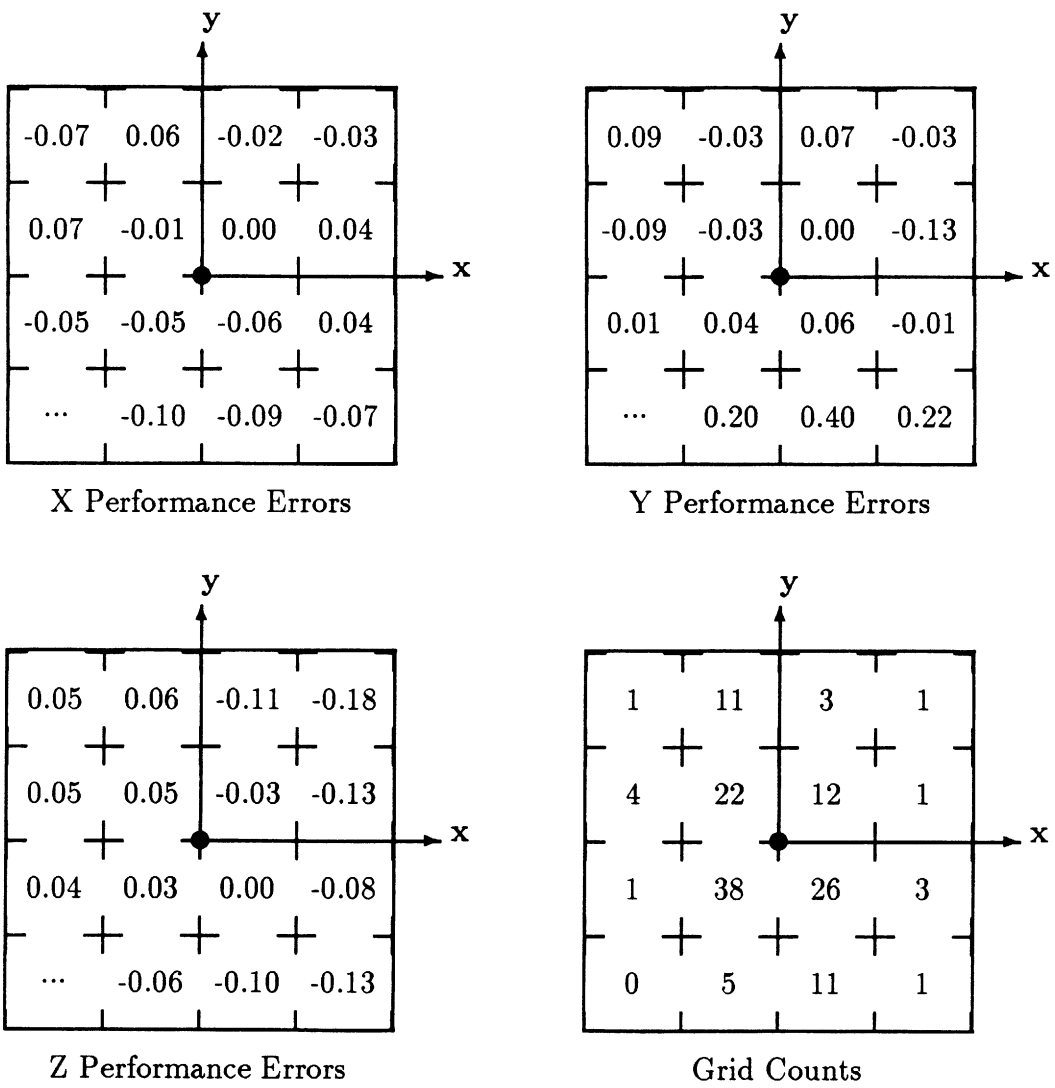


Figure 3.13: Performance Errors (m)

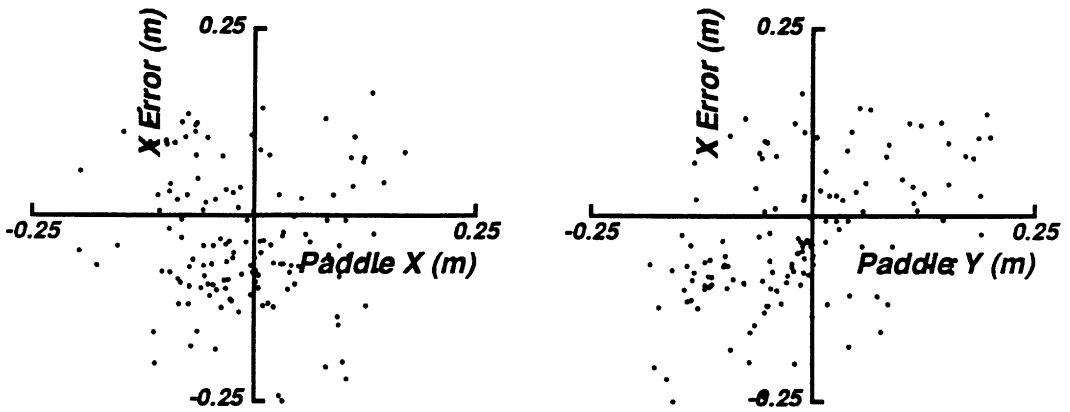


Figure 3.14: X Performance Errors

in the bottom-right grid of Figure 3.13.

Each grid provides an indication of the kind of errors that occur in the juggling system. The $x$ grid suggests that the ball is hit too far to the left (a negative error), except when it is hit from the center of the right edge. The $y$ grid suggests that the ball is consistently hit too far forward when it is hit from the lower edge of the paddle. The $z$ grid indicates that the ball is consistently hit too low when hit from the left side and too high when hit from the right side of the paddle. The ellipsis in a grid box indicate that the ball was never hit from that region of the paddle.

Another way to understand the performance errors is to graph each error as a function of the paddle location from which the ball was hit. The $x$, $y$, and $z$ performance errors are each plotted against the $x$ and $y$ paddle location from which the the ball was hit. The plots are shown in Figures 3.14, 3.15, and 3.16. The plots of $y$ performance error show a parabolic correlation between the $y$ paddle location and $y$ errors. The two plots of $z$ performance errors show a correlation between both $x$ and $y$ paddle location and the $z$ errors.

The data from both the plots and grids offer insight into the performance of the jugging robot. First, the data has somewhat of a predictive nature. Based on the $x$ and $y$ location from which a ball is hit, the data predicts the performance error that is likely to result. Still, some performance errors do not correlate at all with paddle location. For instance, the $x$ paddle location correlates very poorly to $x$ performance errors. Second, the data coincides with some intuitive observations of the juggling robot. For example, the large positive errors in $y$ performance that occur when a ball is hit from the lower edge of the paddle suggests that the robot overcompensates on these hits. This conclusion is similar to the qualitative observation made earlier that balls tend to oscillate front-to-back until they are out of reach. Third, the robot system is clearly noisy. Balls that are hit from the same location are often hit to different points. Correlations do exist between

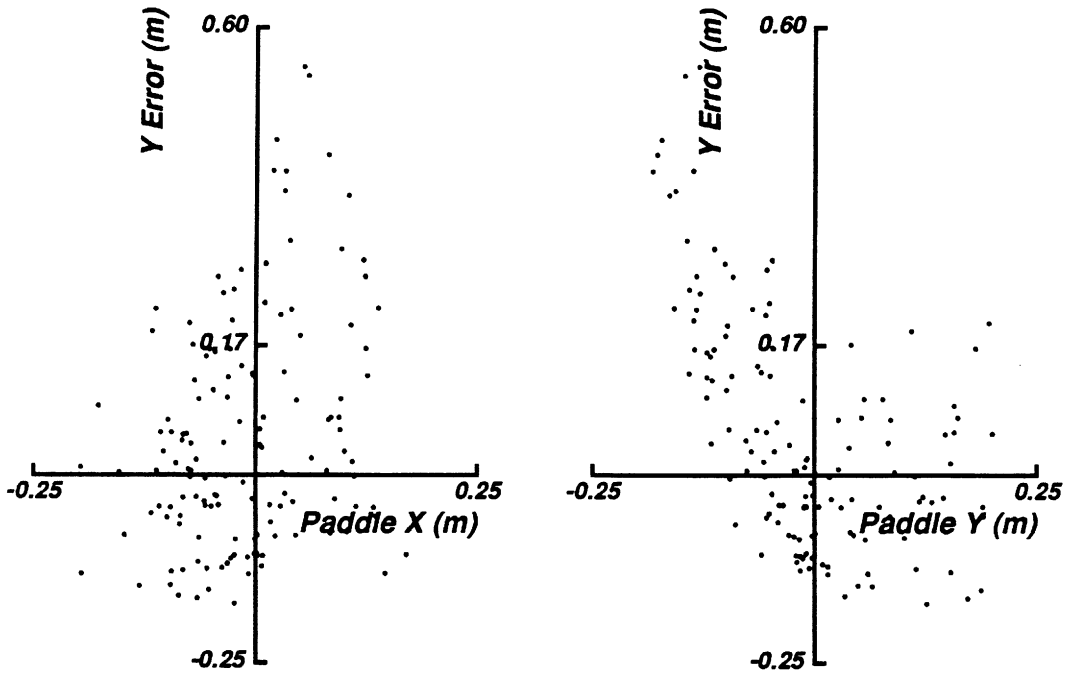


Figure 3.15: Y Performance Errors

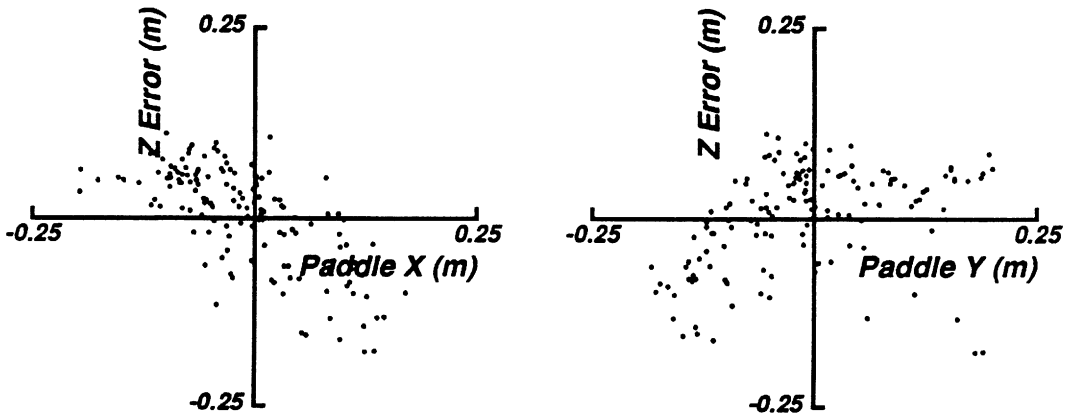


Figure 3.16: Z Performance Errors

paddle location and performance errors, but only in the presence of significant noise.

### 3.5.2 What the Errors Mean

In this subsection, we try to understand what the performance errors mean in the context of task-level learning. In other words, we try to answer the question: what can the juggling robot learn?

The data shows the performance errors that occur when a ball is hit from a particular location on the paddle. The data is similar to the errors that were corrected when the first and second hits were learned. In those cases, an $x$, $y$, and $z$ performance error occurred when the robot used the target aim of $(0.0, 0.0, 1.0)$ to hit the ball. That performance error was dependent on the ball being hit from the $x, y = (0.0, 0.0)$ location on the paddle. Our new data provides an indication of the performance error of the system for a ball hit from *any other* location on the paddle.

The performance error at each paddle location can be used to correct the aim for a hit from that particular paddle location. When the first and second hit were learned, the performance error was used to modify the aim. Now that we have data for hits from different paddle locations, we can correct the aim for many different hits. The aim for a particular hit can be corrected with Equation 2.10 by using the performance error that occurred when a ball was previously hit from that location. To include this notion of state, Equation 2.10 can be rewritten

$$\mathbf{aim_{n+1}}(x, y) = \mathbf{aim_n}(x, y) - (\mathbf{hit_n}(x, y) - \mathbf{target}) \tag{3.20}$$

The script variables $x$ and $y$ denote the state of the system—the location on the paddle from which a ball is hit. Thus, a correction to the aim is calculated based on the state of the juggling system.

The notion that the aim is dependent on the state of the system—where the ball will be hit from—deserves some explanation. The implication is that the error in the task model is not uniform throughout the workspace of the juggling robot. Instead, the error explicitly depends on where the ball is be hit from—the $x$ and $y$ state of the system.

### 3.5.3 Applying State-Based, Task-Level Learning

In this subsection, we describe how to implement task-level learning on the juggling system. We begin by describing a table-based method of task-level learning that is based on the error grids described above. We then describe two function-based methods in which functions are fit to the performance error data. Finally, we directly address the question of how to generalize past experience when applying task-level learning.

Our first experiments involved a table-based method of task-level learning. We began by only trying to reduce the $y$ performance error of the juggling system. Our motivation was that the ball usually oscillated front-to-back until it was out of reach. To reduce the $y$ errors, the $y$ performance error grid was used. Based on the position from where the ball was to be hit, the $y$ error grid shown in Figure 3.13 was used to adjust the aim. The adjustments in aim were made using Equation 3.20. Since only the $y$ aim was modified, Equation 3.20 was decomposed into component equations

$$\begin{aligned} \text{aim}^{\text{x}}_{\text{n+1}} &= \text{aim}^{\text{x}}_{\text{n}} && (3.21) \\ \text{aim}^{\text{y}}_{\text{n+1}}(x,y) &= \text{aim}^{\text{y}}_{\text{n}} - (\text{hit-grid}^{\text{y}}_{\text{n}}(x,y) - \text{target}^{\text{y}}) && (3.22) \\ \text{aim}^{\text{z}}_{\text{n+1}} &= \text{aim}^{\text{z}}_{\text{n}} && (3.23) \end{aligned}$$

Note that the x, y, z superscripts denote the component of the aim. The script $x$ and $y$ denote two state variables of the system—the $x, y$ location from which the ball is hit. Only the aim in the y direction, $\text{aim}^{\text{y}}$, was adjusted in these experiments. Ten juggling sequences were performed with the robot using this correction to the aim. Unfortunately, the average performance of the robot did not improve. On average, eight hits were performed both before and after the learning was applied.

The table-based method was unsuccessful for two reasons. First, the coarse resolution of the table masks the character of the inaccuracies in the task model. Averaging the performance errors in each 0.12 m square grid box obscures the variations in performance error. One way avoid this problem is to increase the resolution of the grid, but that implies doing many more juggling trials to record enough observations. As the data indicate, even at this coarse resolution, the lower-left grid box contains no observations. Second, system noise is an issue because the observations that make up the grid are not uniformly distributed. As Figure 3.13 shows, the data is concentrated along the two center columns of the grid. Several of the other averages in the grid are in fact based on only one observation. Any noise in those observations would seriously affect the learning process.

Our second set of experiments involved fitting a planar *function* to the performance error observations. Again, we initially worked with the $y$ performance errors. A plane was fit to the errors that occurred when a ball was hit from location $x, y$ on the paddle. (A plane was fit to the raw data shown in Figure 3.15, not the cumulative averages of the $y$ grid in Figure 3.13.) For these learning experiments, Equation 3.20 became

$$\begin{aligned} \text{aim}^{\text{x}}_{\text{n+1}} &= \text{aim}^{\text{x}}_{\text{n}} && (3.24) \\ \text{aim}^{\text{y}}_{\text{n+1}}(x,y) &= \text{aim}^{\text{y}}_{\text{n}} - (\text{hit-function}^{\text{y}}_{\text{n}}(x,y) - \text{target}^{\text{y}}) && (3.25) \\ \text{aim}^{\text{z}}_{\text{n+1}} &= \text{aim}^{\text{z}}_{\text{n}} && (3.26) \end{aligned}$$

Based on the planar fit to the data, the term hit-function$_n^y(x,y)$ was set to

$$\text{hit-function}_n^y(x,y) = 0.062 + 0.243 \cdot x + 1.042 \cdot y \tag{3.27}$$

Note the large dependence of the $y$ performance error on the $y$ location from which the ball was hit. We performed 15 juggling trials with this aim correction equation. The average number of hits rose from eight to ten.

With this encouraging result, we fit a second plane to the new performance errors that occurred during these juggling trials. We effectively decided to iterate using planar functions that describe the performance errors. Since the new performance errors occurred with the new aims, the new planar function was added or *superimposed* onto the first one. As a result, Equation 3.26 was reused, but Equation 3.27 became

$$\begin{aligned}\text{hit-function}_n^y(x,y) &= 0.062 + 0.243 \cdot x + 1.042 \cdot y \\ &\quad + 0.046 + 0.002 \cdot x - 0.228 \cdot y\end{aligned} \tag{3.28}$$

Together, these planes describe the performance error of the juggling robot. With these refined corrections, ten juggling sequences were performed. Unfortunately, the average performance *fell* from eight hits to six. This second state-based learning iteration degraded system performance.

The problem with this function-based approach was in the *planar functions* chosen to describe the state-dependent performance errors. The problem can be examined from several viewpoints. First, planar functions did not accurately describe the performance errors from a qualitative standpoint. The data in Figure 3.15 suggests a parabolic relation between $y$ performance error and the $y$ location from which the ball was hit. Second, from a statistical perspective, planar fits did not accurately describe the data. A planar fit to the data has a fitting coefficient of 32%, while a second order polynomial fit has a fitting coefficient of approximately 60%. Third, an observation of the robot's performance suggests that the second hit consistently propelled the ball too far back. The ball was hit backwards because of an inaccurate correction to the aim. For the second hit, the ball generally falls near the center of the paddle ($x, y = 0.0, 0.0$), and hit-function$_n^y(x,y)$ reduces to 0.108 m. This constant is applied as a correction to the aim, making the robot aim 0.108 m further back. This correction directly contradicts the successful learning experiments for the second hit in which balls are hit from ($x, y = 0.0, 0.0$). In those experiments, an aim was successfully learned, suggesting that the constant term in the hit-function should be close to 0.0 m. We thus conclude that planar fits obscure the second order nature of the performance error.

Our third set of experiments involved a function-based approach that fits second order polynomials to the performance error. Our motivation is both qualitative and statistical. From a qualitative point of view, the data shown in Figure 3.15 exhibits a parabolic nature. Statistically, a second order polynomial fit to

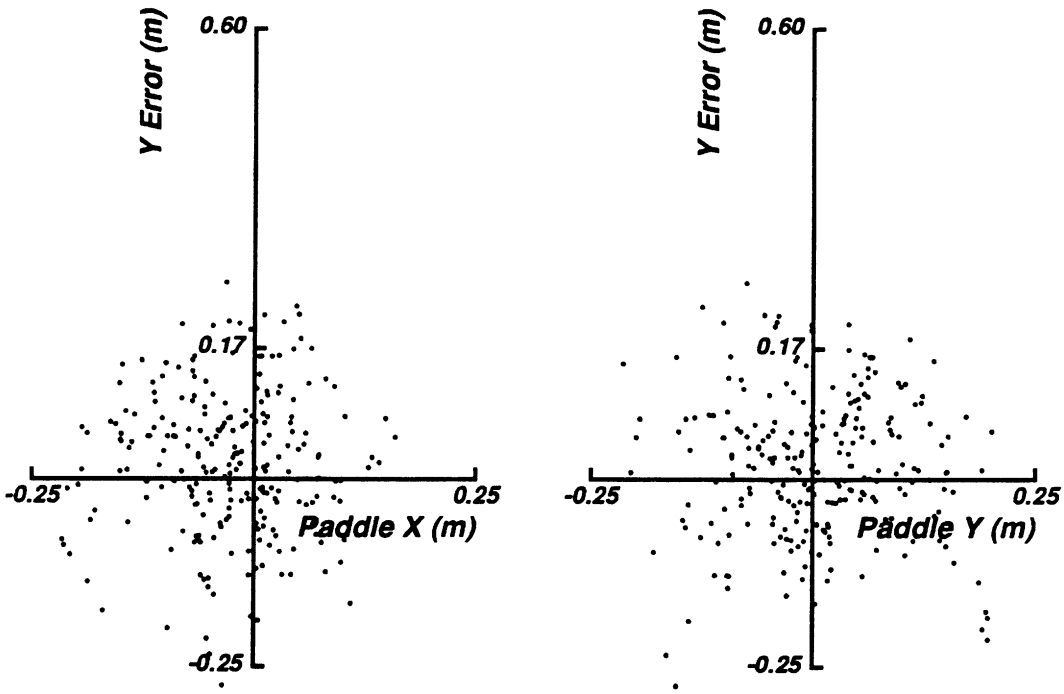


Figure 3.17: Y Performance Errors After Learning

the performance data raises the fitting coefficient from 32% to 60%. Once again, we concentrated on only learning the $y$ performance errors. Equation 3.26 served as the basis for learning, and Equation 3.27 became

$$\begin{aligned} \text{hit-function}_{\text{n}}^{\text{y}}(x,y) &= -0.028 + 0.188 \cdot x - 1.020 \cdot y \\ &\quad - 1.812 \cdot x \cdot y + 0.037 \cdot x^2 + 7.517 \cdot y^2 \end{aligned} \tag{3.29}$$

Based on this function-based task-level learning approach, the robot successfully hit the ball an average of 25 times over ten trials. The low was 12 hits and the high of 44 hits (a software limit at the time) was reached three times.

The successful juggling trials can be analyzed from several perspectives. First, the new errors in performance can be plotted against the location from which the ball was hit. Figure 3.17 shows the $y$ performance errors that resulted once learning was applied. These two graphs can be compared directly to the plots in Figure 3.15 that describe the juggling performance errors *before* learning is applied. Second, a polynomial can be fit to the new data to identify any remaining structure. A fit was performed, but the fitting coefficient of 3% indicates that the location from which the ball was hit no longer explains the performance errors. Furthermore, the standard deviation of the $y$ performance errors is 0.10 m, suggesting that the errors have been nearly reduced to the level of noise in the system ($\sigma_y = 0.07$ m).

In our final set of experiments, state-based task-level learning was applied to

eliminate all task errors—along $x$, $y$, and $z$ axes. These experiments are similar to those in which the $y$ performance error was fit by a second order polynomial. Now, the $x$ and $z$ errors are also fit by second order polynomials, and each function is simultaneously used to improve juggling performance. The equations that correct for the task errors are rewritten as

$$\begin{aligned}
\text{aim}^{x}_{n+1}(x,y) &= \text{aim}^{x}_{n} - (\text{hit-function}^{x}_{n}(x,y) - \text{target}^{x}) && (3.30)\\
\text{aim}^{y}_{n+1}(x,y) &= \text{aim}^{y}_{n} - (\text{hit-function}^{y}_{n}(x,y) - \text{target}^{y}) && (3.31)\\
\text{aim}^{z}_{n+1}(x,y) &= \text{aim}^{z}_{n} - (\text{hit-function}^{z}_{n}(x,y) - \text{target}^{z}) && (3.32)
\end{aligned}$$

The corresponding functions are obtained by fitting data to the $x$, $y$, and $z$ performance errors

$$\begin{aligned}
\text{hit-function}^{x}_{n}(x,y) &= -0.029 + 0.037 \cdot x - 0.426 \cdot y \\
&\quad - 0.772 \cdot x \cdot y + 1.385 \cdot x^2 - 0.603 \cdot y^2 && (3.33)\\
\text{hit-function}^{y}_{n}(x,y) &= -0.028 + 0.188 \cdot x - 1.020 \cdot y \\
&\quad - 1.812 \cdot x \cdot y + 0.037 \cdot x^2 + 7.517 \cdot y^2 && (3.34)\\
\text{hit-function}^{z}_{n}(x,y) &= 0.036 - 0.534 \cdot x + 0.212 \cdot y \\
&\quad - 1.648 \cdot x \cdot y - 2.344 \cdot x^2 - 2.886 \cdot y^2 && (3.35)
\end{aligned}$$

The fitting coefficients for these functions were 24%, 60%, and 76%, respectively. The small fitting coefficient for errors in the $x$ direction indicate that the location from which the ball was hit is not a good predictor. The experimental results were fairly good. The jugging robot averaged 21 hits, with a low of 7 and a high (just once) of 44. From a qualitative standpoint, the robot seemed a bit more erratic. The erratic behavior and the smaller average number of hits can partially be explained by the use of a polynomial to correct the $x$ aim. Corrections based on such a poor fit could have caused unstable performance.

The successful results of these experiments can be analyzed both graphically and statistically. The performance errors that occurred can be graphed as a function of the location from which the ball was hit. The errors that occurred after the task-level learning was applied are shown in Figures 3.18, 3.19, and 3.20. These graphs can be compared directly to those of Figures 3.14, 3.15, and 3.16. To check whether any second order structure remained in the data, second order polynomials were fit to each performance error. Fitting coefficients for the polynomial were all under 25%. The implication is that the position from which the ball was hit no longer predicts performance error. In addition, the standard deviations of the hit errors are $\sigma_x = 0.07$ m, $\sigma_y = 0.10$ m, and $\sigma_z = 0.03$ m, close to the noise level of the system.

Each of the three approaches described above—table-based, planar-function-based, and parabolic-function-based—are efforts at *generalizing* task-level information. In each case, the method provides a correction to the aim for a ball that

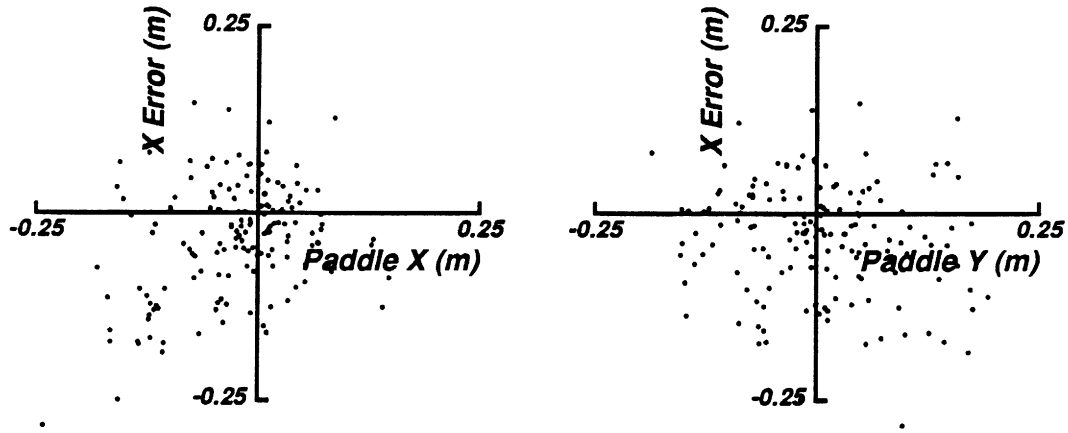


Figure 3.18: X Performance Errors

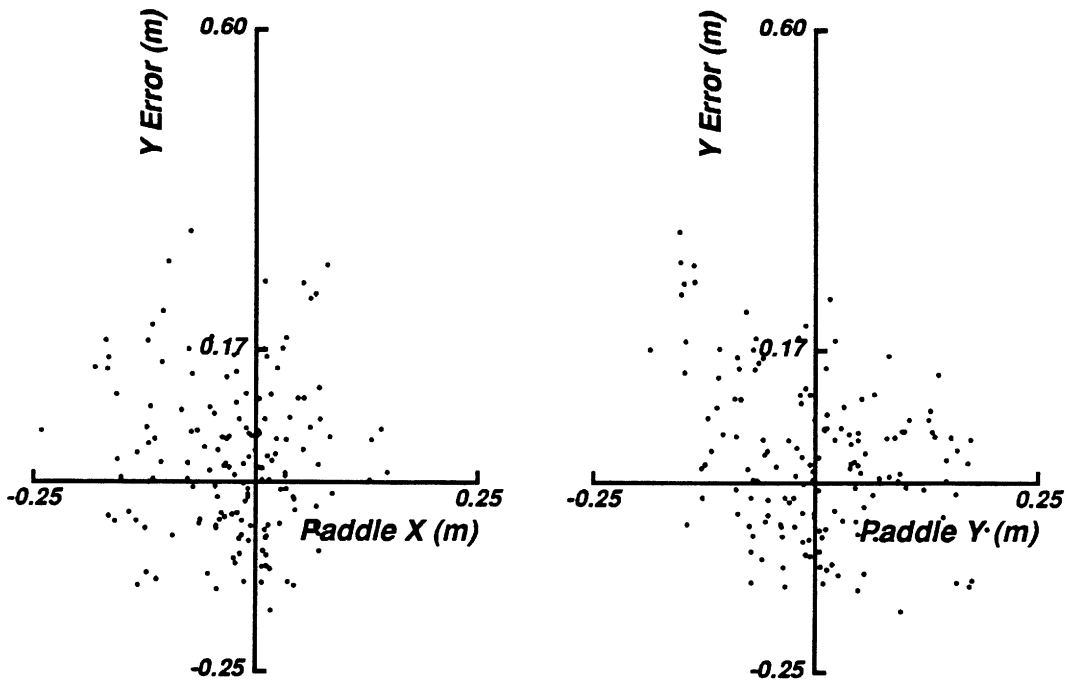


Figure 3.19: Y Performance Errors

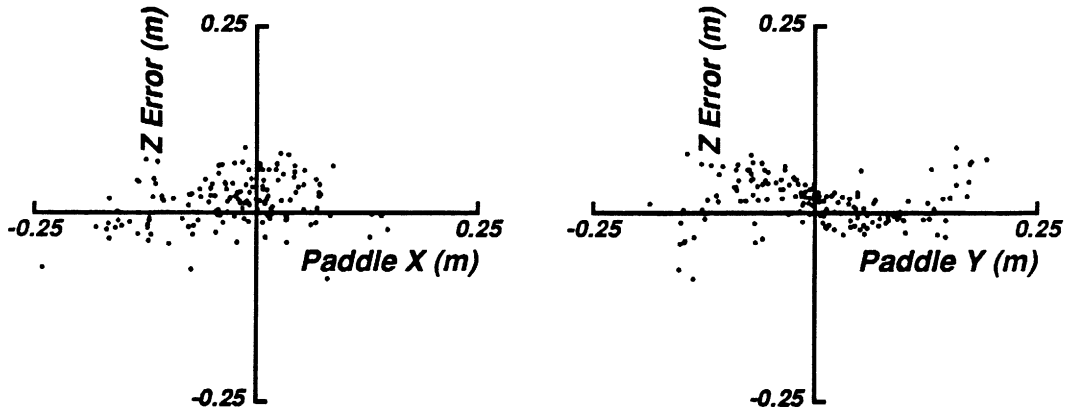


Figure 3.20: Z Performance Errors

is hit anywhere on the paddle. The basis for this correction is a series of hits, or experiences, that occurred at discrete points on the paddle. These 160 experiences are generalized to a hit from any location on the paddle. In the table-based method, the observations were generalized by dividing the paddle into a grid, and averaging all observations in each section. In the function-based methods, a planar or polynomial function is fit to the data points. In both methods, the 160 experiences are generalized to corrections in the $x$, $y$, and $z$ aim.

These three approaches provide a convenient way of dealing with noise during the generalization process. In the table-based method, observations are averaged, reducing the effect of noise in the system. In the function-based methods, a least-squares procedure is used to find a surface that best describes the observations. In both approaches, the effect of noise is largely reduced, allowing task-level learning to improve system performance.

## 3.6 Discussion

We have demonstrated that an extended form of the task-level learning algorithms successfully improves the performance of a juggling robot. The juggling system practiced the task, monitored its own performance, and adjusted its aim to better perform the task. Overall, we note three major contributions.

The first contribution of this chapter is that the performance of a complex juggling system can be improved with task-level learning. A task model of juggling formed the basis for describing and improving the robot system's performance. The task-level algorithms developed in Chapter 2 were extended so that they could be applied to a complex, multi-dimensional juggling task. The task-level learning algorithms were further extended to take into account some of the state variables of the juggling system. This extended form of the task-level learning algorithms

improved the performance of the juggling system from an average of 8 hits to an average of 25 consecutive hits.

The second contribution is that generalizing past experience is fundamental to improving the performance of a robot system. Generalization is useful because past experience provides an indication of future robot performance. In the case of juggling, performance errors are generalized on the basis of the location from which the ball is hit. Many experiences are necessary because errors in the task model are distributed non-uniformly over the robot's workspace. A robot system can learn from past experience, using task-level learning to correct for the performance errors that are likely to occur. For the juggling system, generalizing past experience was instrumental in improving performance from an average of 8 hits to an average of 25 consecutive hits.

The third contribution is to identify the need for a sequence of training steps that improve the performance of a complex task. In this case, the training steps correspond to three subtasks: the first juggling hit, the second juggling hit, and the successive juggling hits. The performance of each subtask in the training sequence can be improved by task-level learning. So far, the form of the training sequence is chosen by the researchers based on an intuitive understanding of the juggling task. The *process* of choosing a particular set of training steps is an avenue for further research. Our conclusion is that a training sequence is important for the juggling task, and may be necessary for other complex tasks.

Finally, we want to answer the question: what has the juggling robot learned? The robot system has learned what aim to use in order to hit a ball back to the center of the paddle. In fact, the robot has learned a number of *different* aims that depend on the location from which the ball will be hit. The variation in these aims is based on how accurately the task model describes the juggling system for different hits.

*This empty page was substituted for a blank page in the original document.*

# Chapter 4

# Other Ways to Improve Robot Performance

In this chapter, we discuss other approaches that researchers have used to get robots to perform a variety of tasks. In order to do so, we survey several robot systems that have been built in recent years. We focus on the approach that each research group has chosen to improve system performance.

We broadly classify the surveyed research into two categories: calibration and non-calibration research. First, we discuss calibration research which we further divide into two phases. Component model calibration emphasizes the precise identification of the parameters of the component kinematics, dynamics, and visual models of a robot system. System calibration involves aligning the component models of a robot system in order to control the entire system. Second, we discuss approaches that researchers have taken when calibration has *not* lead to successful task performance. These non-calibration approaches include feedback control and other iterative schemes.

To put task-level learning in perspective, we include a section that briefly compares calibration and learning.

To provide some background, we briefly survey recent work in trajectory learning that proposes practice as a means of improving performance.

## 4.1 Calibration Approaches

One way to increase robot functionality is to control the robot using better models. A good deal of research in robotics has thus been directed towards accurately modeling the component modules of robot systems—the kinematics, dynamics, actuation, and sensing components. These models are calibrated to correctly predict the response of the component systems of the robot. Once these component models are calibrated, they must be aligned with respect to one another. This process, which we call system calibration, is an integral part of building a robot

system. After calibrating both the component models and the system as a whole, the robot system can often be commanded to perform a wider range of tasks.

In this section, we examine calibration research for some clues on how the performance of robot systems can be improved. The literature is extensive in this field, and we do not survey all recent work. Instead, we try to examine some representative robot systems [Andersson 1988; Beni, Hackwood, and Trimmer 1984; Clocksin et al. 1985; Gershon and Porat 1988; Ikeuchi et al. 1986; Inoue and Inaba 1984; King et al. 1988; Liebes et al. 1988; Lozano-Perez et al. 1987; Luo, Mullen, and Wessell 1988; Roth and O'Hara 1987; Skaar, Brockman, and Hanson 1987; Taylor, Hollis, and Lavin 1985; Whitney 1987] in order to understand the issues relevant to robot learning. The first subsection is devoted towards identifying the basic issues in calibrating the component models of a robot. The second subsection serves to identify the calibration issues that arise when a complete robot system is assembled.

### 4.1.1 Component Model Calibration

Much recent work has concentrated on accurately identifying and calibrating the dynamic, kinematic, and sensing models of robots. We briefly touch upon research in the areas of robot dynamics and kinematics in order to point out some of the important issues involved. For a more detailed investigation of the field, we refer the reader to a survey of the literature [Hollerbach 1988].

The first issue in model calibration is to choose an accurate *structural model* of the system. A rich and accurate model is necessary for each component of the robot system. For example, in the area of robot kinematics, the Denavit-Hartenberg representation of a serial link manipulator is often chosen [Denavit and Hartenberg 1955]. This representation makes certain assumptions about the structure of the system. The drive train is assumed to be free from backlash, compliance, and gear transmission error. In addition, nearly-parallel neighboring joint axes make the kinematic parameters extremely sensitive to measurement error. When the parameters of this structural model are accurately estimated, a robot can be controlled to accuracies of approximately 0.5-1.0 mm [Hayati and Roston 1986].

To improve the positioning accuracy of a robot, the *structure* of the kinematic model needs to be extended. These extensions routinely attempt to accurately model the features that the original Denavit-Hartenberg structure neglects—nearly parallel joint axes, joint backlash and compliance, and gear transmission error. When the robot kinematic model is extended to include these effects, a robot can be controlled to accuracies of up to 0.2-0.3 mm [Chen and Chao 1986; Whitney, Lozinski, and Rourke 1986].

The second major issue in model calibration is estimating the parameters of the model. This parameter estimation procedure tends to vary based on the particular

model of the component system. The essential feature is to operate the system over the workspace of interest, and to find parameters of the model that maximize the model's predictive accuracy. For example, in robot dynamics calibration, parameters corresponding to masses and inertias of the robot links are estimated based on the Newton-Euler model of rigid body dynamics. Parameter estimation begins by moving the manipulator through its workspace. Based on estimates of arm acceleration and corresponding motor torques, the model parameters are estimated [Atkeson, An, and Hollerbach 1986; Mayeda, Osuka, and Kangawa 1984; Mukerjee 1984; Neuman and Khosla 1985; Olsen and Bekey 1985].

These two major issues must be faced when a component model of a robot system is calibrated. A model structure must be chosen and then the model parameters must be estimated. For each model, a different structure is identified that accurately describes the system and the resulting parameters are determined. Models of cameras, lasers, and other sensors are also calibrated in this fashion.

### 4.1.2 Complete System Calibration

Once the component models of a system are calibrated, they must be combined to describe the entire robot system. During this system calibration phase, several major issues arise. First, the robot system can be calibrated using one of two approaches. Either the coordinate frames of component models are aligned or one component system is used to calibrate another. Second, each component model is as important as the component system's effect on total system performance. As a result, different component systems are modeled to different accuracies. Third, models are often calibrated in the vicinity of the task to eliminate the need for more precise models. In this fashion the model is "tuned" to a particular area of the workspace. Fourth, the final test of the effectiveness of system calibration is whether the robot can be commanded to successfully perform the task.

The coordinate frames of component models are often aligned with respect to one another. Alignment allows information from one model to be properly used by another. This system calibration approach was necessary in several robot systems. In the Handey robot system [Lozano-Perez et al. 1987], the range sensor world coordinate frame, the solid modeler world frame, and the robot kinematics world frame needed to be aligned. A significant amount of effort was put into calibrating these component models before the system could function properly. In a parts acquisition robot system [Roth and O'Hara 1987], the Cartesian frame of the range sensor needed to be aligned to the Cartesian frame of the robot tool. This calibration step was necessary because the sensor was mounted directly on the robot. A similar model alignment situation arose in an arc welding robot system [Clocksin et al. 1985] where a laser range sensor was mounted directly on the tool of the robot.

Some researchers have avoided this coordinate frame alignment problem by

using one component system to calibrate another. In a system developed by Inoue and Inaba [1984], the stereo vision model is calibrated by moving the robot tip in the field of view of the cameras. Calibration of the stereo pair is based on estimates of robot position obtained from the robot kinematic model. Researchers have recently developed elaborate calibration procedures for this situation, terming the problem hand-eye calibration [Tsai and Lenz 1987; Shiu and Ahmad 1987]. The essential feature is to use the kinematic model to provide Cartesian coordinates to the sensor system. In a different context, the stereo vision model of a mobile robot [Brooks, Flynn, and Marill 1987] is calibrated relative to the forward motion vision system. Forward velocity, the output from the motion vision model, is used as the calibration input to the stereo vision system. In each of these cases, calibration of the component model is only as accurate as that of the component model used to provide calibration information. For example, in the hand-eye calibration schemes, vision calibration is only as accurate as the robot kinematic model.

A second major calibration issue is the relative accuracy of each component model in the robot system. Researchers calibrate a component model relative to the component system's importance in performing the task. In the case of a ping-pong playing robot system [Andersson 1988], significant effort was placed in calibrating the vision system while other component systems were only modeled in a rudimentary way. Two separate pairs of stereo cameras were used to improve the accuracy of the stereo ranging model and camera timing characteristics were carefully modeled. Emphasis was placed on accurately modeling component systems that were critical to performing the ping-pong task. In a printed-circuit board assembly system [Liebes et al. 1988], researchers were forced to improve both kinematic and camera component models. A table memory was used to model the transformation between Cartesian space and joint space because the standard Denavit-Hartenberg model was too inaccurate for the task. Another table memory was used to model the effects of image distortion caused by the lens and imager of an inexpensive camera. Only after both component systems were modeled more accurately could the robot perform the assembly task. In a parts acquisition system [Roth and O'Hara 1987], the accuracy of a laser range sensor model prevented the robot from acquiring the part successfully. Researchers implemented a table memory to accurately model the effect of sensor nonlinearities. In each of these robot systems, the accuracy of a particular component model was improved before the entire system could function properly.

A third major issue is that system calibration is usually performed in the vicinity of the task. While structural models theoretically describe the system over its entire operating range, in practice they do not. As a result, models are calibrated near the area in which the system will perform the task. For vision models, calibration fixtures are often placed near the area in which the robot will operate [Tsai and Lenz 1987]. Some researchers have exploited the concept of calibrating a robot in the area where the task is to be performed. In a real-time ball-catching

system [Skaar, Brockman, and Hanson 1987], the model describing the relation between camera coordinates and the robot arm is estimated continuously. As the ball gets closer to the robot's cup, new estimates of model parameters are weighted more heavily. Other researchers have implemented a similar approach in the area of hand-eye coordination [Liang, Lee, and Hackwood 1988], terming the procedure dynamic self-calibration.

The fourth major issue is that calibration is complete when a robot system can successfully achieve the task. At this point, the component models describe the robot system "accurately enough" and no learning is necessary to improve the performance of the task. With this thought in mind, no further calibration is necessary in a variety of recent systems appearing in the literature, including a vision-based grasping system [Ikeuchi et al. 1986], a parts acquisition system [Roth and O'Hara 1987], a manipulation system [Lozano-Perez et al. 1987], and a circuit-board assembly system [Liebes et al. 1988]. Other systems that have been described in the literature are ripe for additional calibration or learning schemes if excellent performance is required. These systems, which do not always successfully perform the desired task, include a conveyor tracking system with an 87% success rate [Luo, Mullen, and Wessell 1988], a ball-catching system which succeeds 80% of the time [Skaar, Brockman, and Hanson 1987], and a robot ping-pong system [Andersson 1988].

## 4.2 Non-Calibration Approaches

In this section, we discuss methods that researchers have taken when their *calibrated* robot systems were unable to perform the desired task. We describe some feedback control techniques as well as iterative approaches that have been used by researchers to improve system performance. We also examine why tasks should be defined in sensor coordinates, explaining how this approach improves system accuracy.

### 4.2.1 Feedback Control

Feedback control is one technique for improving the performance of a robot system. In feedback control, commands to the robot are modified based on errors in performance. The most common robot application implemented in terms of feedback control is the task of visually servoing a robot to a desired position. Weiss [1984] formally analyzes and classifies feedback control approaches to the visual servoing task. The simplest approach, termed "static look and move" involves commanding the robot in world coordinates, visually estimating the task error in world coordinates, and updating the world coordinate command to the robot. The approach is termed "static" because each step is performed sequentially.

A second feedback control approach, termed "dynamic look and move" by Weiss, is similar to the static approach, except that each step is processed in parallel. The "dynamic look and move" approach has been used by several different robot researchers to improve system performance. In a robotic sewing system [Gershon and Porat 1988], two dynamic servo controllers are used to maintain cloth tension and to produce constant seam width. For each servo, a sensor measures the task error, and a controller with a simple model of the task updates the commands to the robot. In an arc welding system [Clocksin et al. 1985], the feedback controller compares sensor measurements of task error with previously defined sensor readings. In the event that the current sensor measurements differ from the previously "taught" measurements, commands to the robot are modified to reduce welding errors.

In both uses of feedback control, the issue of model-based command modifications is addressed. First, task errors are transformed to robot command modifications using models that relate errors to commands. In each robot system, the model is a very simple one, involving only a feedback control gain. In the case of robotic sewing, a gain transforms sensed errors in seam width to angular commands to the robot. In the arc welding robot, the transformation from task errors to robot Cartesian commands is also based on a gain. Weiss suggests that for best system performance, an accurate model of the robot system should relate task errors to command modifications.

Neither of these robot controllers addresses the issue of performing a task. First, no framework in which to model and successfully perform a the robot task is described. In fact, only simplied models involving one-dimensional feedback gains are used to correct task errors. In addition, the researchers are selectively choosing which command variables to modify and which errors to sense, based on experience and intuition. Second, no notion of a *sequence of steps* is involved in correcting task errors. The feedback is always state dependent, and ignores any need for a sequence of robot commands that will eliminate the task error. In a more straightforward approach to accomplishing a task, Whitney [1987] outlines a process that links low-level robot commands to task performance based on the system model. He applies this process to a robotic grinding system. After each grinding pass, the system measures grinding errors and issues commands to the robot based on a model of the grinding task.

### 4.2.2 Iterative Techniques

Several researchers have implemented iterative techniques that reduce errors in task performance. We separate these techniques from feedback control because they use a more detailed model of the task. These techniques involve measuring the error in sensor coordinates, transforming this error into world coordinates, and commanding a robot motion that eliminates the error.

These iterative schemes are applied to the task of visually-guiding a robot to a desired position. In an assembly robot [King et al. 1988], a one-step technique was used to improve robot performance beyond the 1.5 mm accuracy achieved during calibration. The robot is first commanded to move a screw above a hole, while a vision system estimates the resulting Cartesian error. The robot system adds this error to the commanded robot position, improving the positioning performance of the system and successfully inserting the screw into the hole. In the robot system designed for high-precision inspection [Beni, Hackwood, and Trimmer 1984], a similar iterative scheme is applied. Based on the Cartesian positioning error estimated by one camera, an offset is added to the Cartesian robot command. After the robot is commanded to move, a second, high-precision camera estimates the new Cartesian error. Based on this error, a second offset is added to the robot command. This two-step scheme improves robot precision from 2.0 mm to 0.05 mm. Both these approaches are model-based in the sense that the inverse model of the system is used to transform errors in performance into command corrections.

Other researchers have implemented similar techniques that iterate until a desired level of precision is achieved. In a robot system designed for precise manipulation [Taylor, Hollis, and Lavin 1985], parts misalignment is reduced to less than 0.01 mm. The iterative steps include sensing the error in camera space, transforming the error into Cartesian coordinates, and adding an offset to the robot command. The iterative algorithm sequences through these three steps until the error is reduced to 0.01 mm. With this iterative approach, the positioning error of the robot system can be reduced to the relative precision of the sensor. This iterative scheme can be considered multi-dimensional since errors in the $x$, $y$, and $\theta$ directions are transformed into robot command corrections. The command correction step is also model-based since the error in camera coordinates is transformed to joint space based on models of the sensor and robot.

These iterative techniques resemble our task-level learning algorithms. The techniques use a model of the visually-guided robot to transform sensor coordinates to Cartesian coordinates. For the task of visually-guiding a robot, task coordinates coincide with Cartesian coordinates. Based on the Cartesian error in performance, the Cartesian commands to the robot are modified. We have developed and formalized similar ideas, and are applying them to complex, multi-dimensional, dynamic tasks.

### 4.2.3 Defining the Task in Sensor Space

When iterative, feedback, or learning techniques are used, the task is usually defined in sensor coordinates. Defining the task in the sensor frame allows the task to be performed to the resolution of the sensor [King et al. 1988; Taylor, Hollis, and Lavin 1985]. As the sensor resolution is increased, the task can be performed

more accurately until the actuator resolution is exceeded [Beni, Hackwood, and Trimmer 1984].

System performance can be improved when a task is defined directly in sensor coordinates. The sensor can then be used to measure task performance. If, instead, a task is defined in a coordinate frame that is related to the sensor by a model, the task will only be performed to the accuracy of the model. The error in the robot system will be equal to the calibration error between the model and the actual coordinate frame transformation. Better system performance can be achieved when a sensor directly defines (or measures) task performance.

Several researchers have defined tasks in sensor coordinates when their robot systems used sensory information. Defining the task in this manner prevented the inaccuracies of one model—the sensor model—from degrading system performance. Luo, Mullen, and Wessell [1988] defined a conveyor tracking and part interception task in camera coordinates. Skaar, Brockman, and Hanson [1987] defined a ball-catching task directly in camera space. Inoue and Inaba [1984] defined a rope-into-ring task and a knot-tying task directly in camera space. Clocksin et al. [1985] described how an arc welding robot system was "taught" a sequence of correct sensor readings by running the system on a prototype fixture.

### 4.2.4 Strategy Modifications

Another method of improving system performance is to modify the *strategy* that the system uses. While a full discussion of task strategies is beyond the scope of this thesis, we can appeal to intuitive notions of strategies to describe how some systems successfully perform tasks. In a parts acquisition system [Roth and O'Hara 1987], grasp locations are chosen based on how reliably they can be described by the sensor. Grasp point locations that are sensitive to sensor error are discarded. The strategy of *which* grasp points to use is modified in order to successfully perform the "parts acquisition" task. In a robot ping-pong player [Andersson 1988], strategy modification is an integral portion of the system. The plan for how to hit and where to hit the ball is modified by expert "tuners." Each tuner, operating in its domain of expertise, estimates the potential success of the planned ping-pong hit. If the system estimates that the hit will be unsuccessful, the "tuner" modifies the hitting strategy.

## 4.3 Calibration and Learning

In this section, we compare learning and calibration as methods of improving system performance. We begin by analyzing the advantages of accurate modeling and the difficulties encountered. We discuss the question: how well should a system be modeled and calibrated? We suggest that calibration and learning are complementary approaches to improving task performance.

Improving the performance of a system by accurate modeling and calibration has several advantages. When the structure of the model is chosen correctly and the parameters are estimated correctly, the model is valid for any inputs and outputs. The experience gained in selected trials generalizes to the entire range of operation. As a result, learning does not need to take place every time the model is used. Additionally, a structured model provides a compact method of representing the input/output behavior of a system. Data need not be stored for every potential scenario, but the model can instead be evaluated when necessary.

Attempts at accurate modeling and extensive calibration have several shortcomings. First, the number and range of robot motions necessary to fully estimate the model parameters is often large. Making trial motions that actually attempt the task may be a more efficient method of reaching the task goal. Second, no matter how well the parameters are estimated, the models are often based on structural assumptions that are not valid in practice. The Newton-Euler model of dynamics, for example, is based on rigid body dynamics which typically is a good but not perfect description of robots. Compensating for the structural assumptions of the model requires a great deal of data, time, and ingenuity. Even after compensation some structural modeling errors will probably remain.

Since accurate modeling and calibration are sometimes difficult, we ask the question: how well should a system be modeled? A tradeoff exists between the time and energy spent in accurate modeling and the desire for better system performance. One answer is that a system should be modeled well enough for the task to be achieved. This answer is often chosen when robot systems are calibrated until the task can be successfully performed—parts can be grasped, seams can be welded, objects can be tracked, and ping-pong can be played. Another answer is to model the system accurately enough for learning or iterative schemes to be applied successfully. This approach embraces the view that the robot should practice a task and learn from experience.

Finally, we want to suggest that learning and calibration are complementary approaches to increasing system performance. With accurate modeling, a robot system can successfully perform a larger number of tasks. Accurate modeling also increases the likelihood that learning will converge and increases the speed of the convergence. With learning, the same tasks can be performed with less accurate models of the robot system. Learning is often easier to implement than extensive model calibration procedures. The conclusion we reach is that both accurate modeling and learning can be used to improve robot performance.

## 4.4 Recent Trajectory Learning Research

In this section, we briefly discuss trajectory learning research. This research focused on learning one component model of a robot system—the dynamics model. We analyze past work in this area as a base from which to examine task-level

learning.

Robot learning research has focused on the trajectory following subtask [Arimoto et al. 1985; Casalino and Gambardella 1986; Craig 1984; Furuta and Yamakita 1986; Hara, Omata, and Nakano 1985; Harokopos 1986; Mita and Kato 1985; Morita 1986, Togai and Yamano 1986; Uchiyama 1978; Wang 1984; Wang and Horowitz 1985]. Robots are made to follow a particular trajectory more accurately as they repeat the movement. Feedforward torque commands for repetitive movements are refined on the basis of previous movement errors. This research has focused primarily on linear learning operators which often ignore the underlying model of the system. The work has also emphasized the stability and not the performance of the proposed algorithms.

We have begun to explore the advantages of using the inverse model as the learning operator and how to apply learning algorithms at the task level. Atkeson and McIntyre [1986] explored fixed-model learning for the trajectory following subtask. The research shows that using the inverse Newton-Euler model as the learning operator reduces most of the movement errors. The same learning algorithms have been applied to the task of positioning a robot at a visual target [Atkeson et al. 1987]. The theoretical convergence criteria and performance for the learning algorithm was then derived. These learning procedures can now be extended to dynamic, complex, multi-dimensional tasks—throwing and juggling.

# Chapter 5

# Conclusion and Future Research

In this chapter, we draw a number of conclusions from our work in task-level learning and suggest several avenues for future research. The conclusions and suggestions are based on our experience in developing and implementing task-level learning on throwing and juggling tasks.

## 5.1 Conclusion

*Task-Level Learning Works.* Task-level learning can successfully improve a robot's performance of complex, multi-dimensional dynamic tasks. The learning algorithm is based on a simple notion of how a person throwing a ball corrects for errors in performance. The task-level aim of the system is modified based on errors in task performance. Learning at the task-level improved a robot's performance of both a ball-throwing task and a complex juggling task. Without doubt, task-level learning could successfully improve a robot's performance of a number of other complex dynamic tasks.

*Learning Can Occur at the Task Level.* Learning can be applied at the task level to improve the performance of robot systems. Less data is necessary to refine task-level commands than the many low-level commands that drive the robot's component systems. Learning at the task level also reduces the degrees of freedom of the models to be learned. Ultimately, learning at the task-level can be used with learning at other levels to simultaneously improve performance.

*Accurate Models Speed Up Learning.* Accurate models improve the initial performance of a system and speed up the performance improvement with practice. The initial performance of the ball-throwing and juggling tasks was possible only with accurate models of the robot systems. Likewise, the performance improvement with practice was based on using accurate models of the systems to transform errors into command corrections. Learning does not obviate the need for accurate models of the task. Instead, learning and accurate modeling are complementary methods of improving task performance.

*Difficulties in Generalizing State-Dependent Errors.* One problem with task-level learning is the need to generalize task-level errors based on the state of the system. In the juggling task, different corrections to the aim were necessary depending on the paddle location from which a ball would be hit. The difficulty lies in identifying which state variables correlate with the task-level errors. In complex systems, the number of state variables to examine is large. However, once the important state variables are identified, task-level learning can proceed using an extended version of the original learning algorithms.

*The Aim is Learned.* Task-level learning improves the performance on a task by adjusting the system's aim. The system learns the task-level goal that will generate the desired performance of the task. In ball throwing, the system learns where to aim the ball so that the ball will land on the target. In juggling, the system learns a number of different aims to use in order to hit the ball back to the center of the paddle. In each case, the system is learning corrections to the aim to compensate for inaccuracies in the component models that describe the task.

## 5.2 Future Research

*Generalization.* Generalizing experience between similar tasks is an important component of learning. The difficulty lies in identifying *what* to generalize, *when* to generalize, and *how much* to generalize. Several related juggling tasks could be performed with the juggling system to try to answer these questions. The original task could be modified to hit the ball to a higher location. The task could entail hitting the ball to an $x, y$ location away from the center of the paddle. The task could involve hitting a different ball with a different coefficient of restitution. In each case, the experience gained in the original juggling task can be used as a guide to improving the performance of the modified task.

*Decomposing the Task into Training Steps.* The concepts of *whether* and *how* to decompose a task into training steps are important ones to investigate further. In Chapter 3, the juggling task was decomposed into three subtasks based on the intuition of the researchers. Instead, a principled method is necessary. Many different complex tasks could first be analyzed to understand the potential need to decompose a task into subtasks, or training steps. Such an analysis might also suggest how to choose the particular set of simpler subtasks.

*Effects of Noise.* A detailed statistical analysis of the effects of noise on the task-level learning algorithms is necessary. It is important to understand exactly how the repeatability of the system affects the speed and stability of the learning process. With such an understanding, the learning algorithms can be modified to perform successfully in the presence of noise.

*Task Strategies.* Adjusting the task strategy is a separate level of learning that needs to be investigated further. Up to now, the strategy used to accomplish a task has been fixed. For example, in ball throwing, the task strategy involved

adjusting the release velocity to throw the ball closer or further. The system never used the strategy of modifying the release angle to affect the distance the ball is thrown. We need to understand how to identify, characterize, and choose from among the different task strategies that can be used to perform a task.

# References

**Aboaf, E.W., C.G. Atkeson and D.J. Reinkensmeyer**, "Task-Level Robot Learning", IEEE Conf. on Robotics and Automation, (Philadelphia, PA, April 24-29, 1988).

**An, C.H., C.G. Atkeson, and J.M. Hollerbach**, *Model-Based Control of a Robot Manipulator* (MIT Press, Cambridge, MA, 1988).

**Anderson, R.L**, *A Robot Ping-Pong Player: Experiment in Real-Time Intelligent Control* (MIT Press, Cambridge, MA, 1988).

**Arimoto, S., S. Kawamura, F. Miyazaki, and S. Tamaki**, "Learning Control Theory for Dynamical Systems", Proc. 24th Conf. on Decision and Control, (Fort Lauderdale, Florida, Dec. 11-13, 1985).

**Atkeson, C.G., E.W. Aboaf, J. McIntyre, and D.J. Reinkensmeyer**, "Model-Based Robot Learning", Fourth Intl. Symposium on Robotics Research, (Santa Cruz, CA, August 9-14, 1987).

**Atkeson, C.G., C. An, and J.M. Hollerbach**, "Estimation of Inertial Parameters of Manipulator Loads and Links", *International Journal of Robotics Research* **5** (3): (1986) pp. 101-119.

**Atkeson, C. G. and J. McIntyre**, "Robot Trajectory Learning Through Practice", IEEE Conf. on Robotics and Automation, (San Francisco, CA, April 7-10, 1986).

**Beer, F.P. and E.R. Johnston, Jr.**, *Vector Mechanics for Engineers* (McGraw-Hill Book Company, New York, NY, 1977).

**Beni, G., S. Hackwood, and W.S. Trimmer**, "High-Precision Robot System for Inspection and Testing of Electronic Devices", IEEE Conf. on Robotics and Automation, (Atlanta, GA, March 13-15, 1984).

**Brooks, R.A., A.M. Flynn, and T. Marill**, "Self-Calibration of Motion and Stereo Vision for Mobile Robots", Fourth Intl. Symposium on Robotics Research, (Santa Cruz, CA, August 9-14, 1987).

**Casalino, G. and L. Gambardella**, "Learning of Movements in Robotic

Manipulators", Proc. IEEE Conf. on Robotics and Automation, (San Francisco, CA, April 7-10, 1986).

**Chen, J. and L.M. Chao**, "Positioning Error Analysis for Robot Manipulators with All Rotary Joints", Proc. IEEE Conf. on Robotics and Automation, (San Francisco, CA, April 7-10, 1986).

**Clocksin, W.F., J.S.E. Bromley, P.G. Davey, A.R. Vidler, C.G. Morgan**, "An Implementation of Model-Based Visual Feedback for Robot Arc Welding of Thin Sheet Steel", *International Journal of Robotics Research* **4** (1): (1985) pp. 13-26.

**Craig, J. J.**, "Adaptive Control of Manipulators Through Repeated Trials", Proc. American Control Conference, (San Diego, June 6-8, 1984).

**Denavit, J. and R.S. Hartenberg**, "A Kinematic Notation of Lower Pair Mechanisms Based on Matrices", *J. Applied Mechanics* **4** : (1955) pp. 215-221.

**Forsythe, G.E., M.A. Malcolm, and C.B. Moler**, *Computer Methods for Mathematical Computations* (Prentice Hall, Englewood Cliffs, NJ, 1988).

**Furuta, K. and M. Yamakita**, "Iterative Generation of Optimal Input of a Manipulator", Proc. IEEE Conf. on Robotics and Automation, (San Francisco, CA, April 7-10, 1986).

**Gershon, D. and I. Porat**, "Vision Servo Control of a Robotic Sewing System", IEEE Conf. on Robotics and Automation, (Philadelphia, PA, April 24-29, 1988).

**Gragg, W.B. and G.W. Stewart**, "A Stable Variant to the Secant Method for Solving Nonlinear Equations", *SIAM J. Numerical Analysis* **13** (6): (1976) pp. 889-903.

**Hara, S., T. Omata, and M. Nakano**, "Synthesis of Repetitive Control Systems and its Application", Proc. 24th Conf. on Decision and Control, (Fort Lauderdale, Florida, Dec. 11-13, 1985).

**Harokopos, E. G.**, "Optimal Learning Control of Mechanical Manipulators in Repetitive Motions", Proc. IEEE Conf. on Robotics and Automation, (San Francisco, CA, April 7-10, 1986).

**Hayati, S.A. and G.P. Roston**, "Inverse Kinematic Solution for Near-Simple Robots and Its Application to Robot Calibration", *Recent Trends in Robotics: Modeling, Control, and Education* (Elsevier Science Publ. Co., edited by M. Jamshidi, J.Y.S. Luh, and M. Shahinpoor, 1986).

**Hollerbach, J.M.**, "A Survey of Kinematic Calibration", *Robotics Review* (MIT Press, edited by O. Khatib, J.J. Craig, T. Lozano-Perez, Cambridge, MA, 1988).

**Ikeuchi, K., H.K. Nishihara, B.K.P Horn, P. Sobalvarro, S. Nagata**, "Determining Grasp Configurations using Photometric Stereo and the Prism Binocular Stereo System", *International Journal of Robotics Research* **5** (1): (1986) pp. 46-65.

**Inoue, H. and M. Inaba**, "Hand-Eye Coordination in Rope Handling", First Intl. Symposium on Robotics Research, (Bretton Woods, NH, August 25 - September 2, 1983).

**King, F.G, G.V. Puskorius, F. Yuan, R.C. Meier, V. Jeyabalan, and L.A. Feldkamp**, "Vision Guided Robots for Automated Assembly", IEEE Conf. on Robotics and Automation, (Philadelphia, PA, April 24-29, 1988).

**Liang, P., J.F. Lee, and S. Hackwood**, "A General Framework for Robot Hand-Eye Coordiantion", IEEE Conf. on Robotics and Automation, (Philadelphia, PA, April 24-29, 1988).

**Liebes, S., W. Gong, A. Gray, H. Martins, B. Modesitt, and R. Tella**, "FLAIR—Robotic Printed Circuit Board Assembly Workcell", IEEE Conf. on Robotics and Automation, (Philadelphia, PA, April 24-29, 1988).

**Lozano-Perez, T., J.L. Jones, E. Mazer, P.A. O'Donnell, W.E.L Grimson, P. Tournassoud, and A. Lanusse**, "Handey: A Robot System that Recognizes, Plans, and Manipulates", IEEE Conf. on Robotics and Automation, (Raleigh, NC, March 31 - April 3, 1987).

**Lou, R.C., R.E. Mullen, and D.E. Wessell**, "An Adaptive Robotic Tracking System Using Optical Flow", IEEE Conf. on Robotics and Automation, (Philadelphia, PA, April 24-29, 1988).

**Mayeda, H., K. Osuka, and A. Kangawa**, "A new identification method for serial manipulator arms", Preprints IFAC 9th World Congress, (Budapest, July 2-6, 1984).

**Mita, T., and E. Kato**, "Iterative Control and its Application to Motion Control of Robot Arm – A Direct Approach to Servo-Problems", Proc. 24th Conf. on Decision and Control, (Fort Lauderdale, Florida, Dec. 11-13, 1985).

**Morita, A.**, "A Study of Learning Controllers For Robot Manipulators With Sparse Data", M.S. thesis, Mechanical Engineering, Massachusetts Institute of Technology (February 27, 1986).

**Mukerjee, A.**, "Adaptation in biological sensory-motor systems: A model for robotic control", Proc., SPIE Conf. on Intelligent Robots and Computer Vision, SPIE Vol. 521, (Cambridge, November, 1984).

**Narasimhan, S. and D.M. Siegal**, "The Condor Programmer's Manual: Version II", Artificial Intelligence Lab Working Paper 297, Massachusetts Institute of Technology, (July, 1987).

**Neuman, C.P. and P.K. Khosla**, "Identification of robot dynamics: an application of recursive estimation", Proc. 4th Yale Workshop on Applications of Adaptive Systems Theory, (New Haven, May 29-31, 1985).

**Olsen, H.B. and G.A. Bekey**, "Identification of parameters in models of robots with rotary joints", Proc. IEEE Conf. Robotics and Automation, (St. Louis, Mar. 25-28, 1985).

**Press, W.H., B.P. Flannery, S.A. Teukosky, and W.T. Vetterling,** *Numerical Recipes* (Cambridge University Press, Cambridge, England, 1986).

**Roth, G. and D. O'Hara**, "A Holdsite Method for Parts Acquisition Using a Laser Rangefinder Mounted on a Robot Wrist", IEEE Conf. on Robotics and Automation, (Raleigh, NC, March 31 - April 3, 1987).

**Shiu, Y.C. and S. Ahmad**, "Finding the Mounting Position of a Sensor by Solving a Homogeneous Transform Equation of the Form AX=XB", IEEE Conf. on Robotics and Automation, (Raleigh, NC, March 31 - April 3, 1987).

**Skaar, S.B., W.H. Brockman, and R. Hanson**, "Camera-Space Manipulation", *International Journal of Robotics Research* **6** (4): (1987) pp. 20-32.

**Taylor, R.H., R.L. Hollis, M.A. Lavin**, "Precise Manipulation with Endpoint Sensing", Second Intl. Symposium on Robotics Research, (Kyoto, Japan, August 20-23, 1984).

**Togai, M. and O. Yamano**, "Learning Control and Its Optimality: Analysis and Is Application to Controlling Industrial Robots", Proc. IEEE Conf. on Robotics and Automation, (San Francisco, CA, April 7-10, 1984).

**Tsai, R.Y. and R. Lenz**, "A New Technique for Fully Autonomous and Efficient 3D Robotics Hand/Eye Calibration", Fourth Intl. Symposium on Robotics Research, (Santa Cruz, CA, August 9-14, 1987).

**Uchiyama, M.**, "Formation of High-Speed Motion Pattern of a Mechanical Arm by Trial", *Trans. of Society of Instrument and Control Engineers (Japan)* **19** (5): (1978) pp. 706-712.

**Wang, S.H.**, "Computed Reference Error Adjustment Technique (CREATE) For The Control of Robot Manipulators", 22nd Annual Allerton Conf. on Communication, Control, and Computing, (October, 1984).

**Wang, S.H. and I. Horowitz**, "CREATE - A New Adaptive Technique", Proc. of the Nineteenth Annual Conf. on Information Sciences and Systems, (March, 1985).

**Weiss, L.E.**, "Dynamic visual Servo control of Robots: An Adaptive Image-Based Approach", Ph.D. thesis, Department of Electrical and Computer

Engineering, Carnegie-Mellon University, CMU-RI-TR-84-16 (April, 1984).

**Whitney, D.E.,** "Elements of an Intelligent Robot Grinding System", Fourth Intl. Symposium on Robotics Research, (Santa Cruz, CA, August 9-14, 1987).

**Whitney, D.E., C.A. Lozinski, and J.M. Rourke,** "Industrial Robot Forward Calibration Method and Results", *J. Dynamics Systems, Meas., Control* **108** : (1986) pp. 1-8.

**CS-TR Scanning Project**
**Document Control Form**

Date : 8/17/95

**Report #** AI-TR-1079

Each of the following should be identified by a checkmark:
Originating Department:

☒ Artificial Intellegence Laboratory (AI)
☐ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

## Document Information

Number of pages: 80 (88-IMAGES)
Not to include DOD forms, printer instructions, etc... original pages only.

Originals are:
☒ Single-sided or
☐ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☒ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other:_____

Check each if included with document:

☒ DOD Form (2) ☐ Funding Agent Form ☒ Cover Page
☒ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages (by page number): BLANKS FOLLOW PAGES #'ED 1, 2, 3, 8, 57

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description : Page Number:

IMAGE MAP: (1-6) PAGE #1, UN#BLANK, 2, UN#BLANK, 3, UN#BLANK
(7-80) PAGES #'ED 4-8, UN#BLANK, 9-57, UN#BLANK, 58-75
(81-85) SCANCONTROL, COVER, SPINE, DOD(2)
(86-88) TRGT'S (3)

Scanning Agent Signoff:

Date Received: 8/17/95 Date Scanned: 8/21/95 Date Returned: 8/24/95

Scanning Agent Signature: Michael N. Cook

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

| REPORT DOCUMENTATION PAGE | | READ INSTRUCTIONS BEFORE COMPLETING FORM |
|---|---|---|
| 1. REPORT NUMBER<br>AI-TR 1079 | 2. GOVT ACCESSION NO. | 3. RECIPIENT'S CATALOG NUMBER<br>AD-A209940 |
| 4. TITLE (and Subtitle)<br>Task-Level Robot Learning | | 5. TYPE OF REPORT & PERIOD COVERED<br>memorandum |
| | | 6. PERFORMING ORG. REPORT NUMBER |
| 7. AUTHOR(s)<br>Eric Aboaf | | 8. CONTRACT OR GRANT NUMBER(s)<br>N00014-86-K-0685<br>N00014-85-K-0124 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS<br>Artificial Intelligence Laboratory<br>545 Technology Square<br>Cambridge, MA 02139 | | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS |
| 11. CONTROLLING OFFICE NAME AND ADDRESS<br>Advanced Research Projects Agency<br>1400 Wilson Blvd.<br>Arlington, VA 22209 | | 12. REPORT DATE<br>August 1988<br>13. NUMBER OF PAGES |
| 14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)<br>Office of Naval Research<br>Information Systems<br>Arlington, VA 22217 | | 15. SECURITY CLASS. (of this report)<br>UNCLASSIFIED<br>15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT (of this Report)

Distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

tasks
robotics
learning

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

We are investigating how to program robots so that they learn from experience. Our goal is to develop principled methods of learning that can improve a robot's performance of a wide range of dynamic tasks. Our interest is in complex tasks such as throwing, catching, batting, yo-yoing, and juggling. We have developed one method of learning, *task-level learning*, that successfully improves a robot's performance of both a ball-throwing and a juggling task.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(block 20 con't.)

With task-level learning, a robot practices a task, monitors its own performance, and uses that experience to adjust its task-level commands. For example, we have programmed a robot to juggle a single ball in three dimensions. The robot practices the juggling task by batting a ball into the air with a large paddle. The robot uses a real-time binary vision system to track the ball and measure its own performance. Task-level learning consists of building a model of the performance errors at the task level during practice. The robot compensates for the performance errors by using that model to refine the task-level commands. When using task-level learning, the number of hits that the robot can execute before the ball is hit out of range dramatically improves.

Task-level learning is a general method of improving a robot's performance of complex dynamic tasks. Task-level learning serves to complement other approaches for improving robot performance such as model calibration. Our investigation is one step in the process of developing a theoretical and experimental foundation for robot learning.

# Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

**Scanned**

**Date:** 8/21/1995

**M.I.T. Libraries**
**Document Services**