Algorithms for Pinball Simulation, Ball Tracking and Learning Flipper Control

by

Michael Patrick Johnson

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Bachelor of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1993

Author .....
Department of Electrical Engineering and Computer Science
May 17, 1993

Certified by .....
Christopher G. Atkeson
Associate Professor of Brain and Cognitive Sciences
Thesis Supervisor

Accepted by .....
Leonard A. Gould
Chairman, Department Committee on Undergraduate Theses

Algorithms for Pinball Simulation, Ball Tracking and Learning Flipper Control

by

Michael Patrick Johnson

Submitted to the Department of Electrical Engineering and Computer Science
on May 17, 1993, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Computer Science and Engineering

Abstract

In this thesis, I describe the creation of a real-time pinball simulation, trajectory prediction system and real-time computer pinball player. The simulation is object-oriented and was implemented in C and the X Window System. The tracker predicts the ball trajectory using only sequential (x,y) ball observations in order to work with the output of a computer vision system. The computer player is optimized using two different random search techniques — stochastic hill-climbing and Hoeffding Relay Race — over the parameter space of a class of piecewise linear controllers. All systems were designed to be easily interfaced to a real pinball board in the future.

Thesis Supervisor: Christopher G. Atkeson
Title: Associate Professor of Brain and Cognitive Sciences

Algorithms for Pinball Simulation, Ball Tracking and Learning Flipper Control

by

Michael Patrick Johnson

Submitted to the Department of Electrical Engineering and Computer Science
on May 17, 1993, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Computer Science and Engineering

Abstract

In this thesis, I describe the creation of a real-time pinball simulation, trajectory prediction system and real-time computer pinball player. The simulation is object-oriented and was implemented in C and the X Window System. The tracker predicts the ball trajectory using only sequential (x,y) ball observations in order to work with the output of a computer vision system. The computer player is optimized using two different random search techniques — stochastic hill-climbing and Hoeffding Relay Race — over the parameter space of a class of piecewise linear controllers. All systems were designed to be easily interfaced to a real pinball board in the future.

Thesis Supervisor: Christopher G. Atkeson
Title: Associate Professor of Brain and Cognitive Sciences

Contents

List of Figures

List of Tables

Acknowledgements

This thesis would not exist if not for the support and expert guidance of Andrew W. Moore, a post-doctoral fellow working with Chris Atkeson at the AI Lab. I have worked with Andrew for almost a year now on various projects; I have not only learned incredible amounts of information, but I was allowed to take part in a very interesting and quite promising area of machine learning research which I hope to continue in graduate school. Hopefully some of his wisdom and acute intuition has also rubbed off on me. I wish him luck and prosperity in his recent faculty appointment to Carnegie Mellon University's computer science department.

I would also like to thank Chris Atkeson, my thesis advisor, who allowed me to do the things this year that I have wanted to do throughout my undergraduate years at MIT but have been unable to. When he had one of my classes play with a pile of devil sticks and come up with strategies to have a robot learn to do it, it was clear where I wanted to work. Chris taught me that you can do serious research, but still have fun doing it. (And I thought *I* had a lot of gizmos and toys!)

Chapter 1

Introduction

> *'Where shall I begin, please your Majesty?' he asked. 'Begin at the beginning,' the King said, gravely, 'and go on till you come to the end: then stop.'*
>
> — Lewis Caroll

There are those who play pinball and those who do not. Those who do not tell those of us who do that it is merely a random process, perhaps Poisson in the waiting time for the ball to be lost. One can do just as well flipping all the time. Hence, they say, people only think they get better because their personal high score gets better over time and that this is due to the variance of the process — play long enough and you'll land on the tails of the distribution every so often. Being a zealous pinball player myself, I disagree (although on some bad days it is easy to feel like it *is* just a glorified roulette wheel). I believe it is possible for a human to learn to play better and to acquire the ability to accurately hit whatever part of the table they wish from most ball states.

It is also a very interesting control problem due to the stochastic element and limited available actuation. We have only have few opportunities to control the ball and two competing goals — get a good score and don't lose the ball while doing it. This may seem simple, but there is a latent complexity that reveals itself on closer scrutiny. Very simple actions — which flipper to flip and when — lead to many dif-

ferent outputs. In addition, small differences in the action can lead to very different outputs. It is this kind of sensitivity, coupled with the simplicity and limited amount of actuation, that make this a difficult and interesting control problem. Classical linear control algorithms can not handle such problems, as they are in general nonlinear and discontinuous. Research is flourishing in these kinds of control problems that span the border between classical AI planning and classical control.

The goal of this thesis, then, is to find out if a computer could learn to play a simple version of pinball as well as a human player, gauged by the mean score per ball. It would be interesting if the computer player seemed to learn techniques such as catching the ball in the return lane in order to gain better control over it and passing the ball between flippers, things that experienced pinball players learn to do (or more often are told how to do by better players than themselves).

1.1 Background and Related Research

1.1.1 Human Player Evolution

A human learning pinball tends to go through four stages — the spastic flipper, the able flipper, the accurate flipper and the expert flipper. The spastic flipper merely flips both flippers when the ball is in range and hopes for the best. This strategy does not tend to get good scores over time. Next he starts being more selective in his choice of flipping, only flipping when he has to and becoming more accurate, but still unaware of many of the techniques — he is the able flipper. In the next stage, he starts to learn how to catch the ball with the flipper so that he can plan his shots and have more control over where the ball goes. He also may learn how to pass the ball between flippers and how to nudge the table when he is close to losing the ball. Much of his game is catching the ball and firing at known spots; he can hit them a good percentage of the time — this is the accurate flipper. The expert flipper has played enough to know many techniques. He can usually hit the ball from any input configuration to any place on the board quite accurately. He avoids dangerous shots

in favor of safe ones. These are the people who keep getting replays and get ten games for one quarter.

Our goal, then, is to try and reach the able player. We would like a player that at least appears intelligent and manages to get scores comparable to that of the average human player. We present and discuss human player data gathered on our implementation of a pinball simulation in Chapter 2

1.1.2 A Real Pinball Machine

Grzegorz Rogalski, an undergraduate at MIT who also worked with Chris Atkeson in the AI Lab, investigated the problem of pinball control on a home-made, simplified pinball machine [Rogalski, 1992]. Part of his work involved designing and building a board that the ball could successfully be tracked on with a vision system (i.e. painted flat black) and allowing the computer to actuate the flippers. His preliminary pinball player simply flipped when the ball was within range. He mentions that it is reminiscent of the strategy employed by early human players — hit it when you can and pray for the best. He was planning on doing trajectory prediction, but ran out of time. The trajectory prediction scheme I describe in Chapter 4 should work well with his vision system.

My research has kept this real-world machine in mind; all of the algorithms invented were created with the idea that they should work on this real machine as well as with a simulation. Indeed, once efficient learning algorithms are developed and tested in simulation, they should be connected to this real machine by another student in the future.

1.1.3 Global System Overview

Figure 1-1 shows the overall design of the system. Notice that the first block can be either an interface to a real machine such as Rogalski's or the simulation that we use.

We choose to use a simulation for this preliminary research to eliminate the many

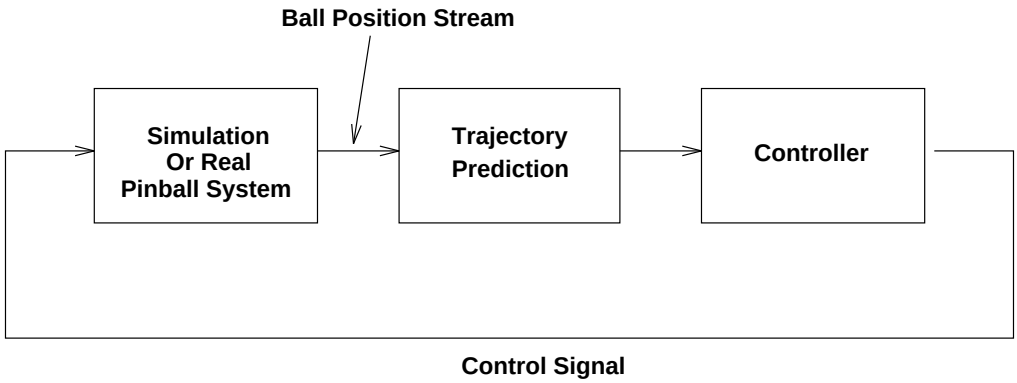

Figure 1-1: System Overview

problems that arise with using a real board: malfunction, dealing with real-time I/O, visual outliers and the many required sensors that all must work perfectly. In addition, a simulation allows us to sample human players easily and allows us to run several searches in parallel on different computers. Once the algorithms work well in simulation, the problems with interfacing to the real board will be worth the trouble.

1.1.4 Philosophy of the Machine Player

Our method for finding a good computer player is a directed random search over the parameter space of the controller framework specified in Chapter 5. This type of "brute-force" learning technique considers the "intelligence" of the controller to be emergent; that is, we do not try to find a good controller by using a lot of domain knowledge, thinking of good strategies to employ and then implementing the "optimal" strategy by hand. Rather, we merely provide a robust framework with some number of parameters and then use a search technique to optimize the parameters with respect to some goal, in our case high scores. As we sample the system more and more over different parameter settings, we can discover which ones are "good" and which ones are "bad." The system will appear to become more "intelligent" over time as the search finds better controllers.

This type of technique has proven useful in control of stochastic Markov systems, where optimal solutions are found with various dynamic programming algorithms

that try to learn the state transition table from samples of the system [Moore and Atkeson, 1992].

Stefan Schaal [Schaal, 1993] has also been investigating this type of directed random search technique for control of a juggling robot. For example, he is using a continuous genetic algorithm search to have a robot bounce a ball indefinitely on a tennis racket.

1.2 Overview

This thesis consists of three main parts: development of a pinball simulation, trajectory prediction and representation, and learning algorithms for the computer player. These sections are broken up as follows:

Chapter 2 contains a brief discussion of the actual simulated pinball table used in this research and presents human player results on it to be used as a benchmark for computer performance.

Chapter 3 discusses the issues in the development of a pinball simulation. It is made as a stand-alone section and can be skipped over without loss of continuity.

Chapter 4 discusses the prediction of ball trajectories and formulating a good representation for a controller to use.

Chapter 5 describes a controller architecture and framework for a computer player.

Chapter 6 deals with several optimization techniques applied to the controller framework in order to produce a good computer player.

Chapter 7 discusses conclusions from the results obtained and offers directions for future areas of research in the problem.

Chapter 2

Description of the Problem and Human Results

While they were content to peck cautiously at the ball, he never spared himself in his efforts to do it a violent injury.

— P. G. Wodehouse, 1881

This chapter describes the structure of the pinball table that the controller will have to play and presents human player results for the same. It also gives an explicit description and analysis of the problem and discusses important simplifications that were made to make it tractable.

2.1 Analysis of the Problem

What exactly is our goal? What would we like the pinball player to learn? We would like a high score, clearly, as we mentioned earlier. In a real pinball machine, however, this is difficult to learn how to do, since the board changes state, with jackpot modes and things that must be done in sequence at certain times. This kind of high-level control is beyond the scope of this thesis. Hence, we break the general pinball control problem into two levels — the high-level controller and a low-level flipper controller.

Such a high-level controller would want a low-level flipper controller to have certain abilities. It would want to say, "Hit this, then this, then this" and have the low-level controller figure out how that should be done. Thus, it would be the low level controller's job to learn accuracy — how to hit a given object from any input configuration and send the ball back to the flippers without losing it. This kind of control is on the level of an expert player. Other issues involved are locating the "good" places to hit and a representation for goals. Rather than try to reach this level directly, we choose to investigate a slightly simpler problem first, that of maximizing a static reward function, in our case the score received for a ball. If we can learn this simpler problem, it seems reasonable that we can learn the accuracy problem. Indeed, if we were to set the score of the desired item to be non-zero and all other scores to be zero, we should learn how to hit that object consistently.

For this reason, we assume that the pinball table does not change scoring states and that all objects are static (some machines have moving objects). Explicitly, we mean that score values for all of the objects are constant over time; there are no jackpot modes that can be entered by performing a sequence of actions, for example. The goal of the controller, then, is to maximize the score that it receives per ball. Clearly this could be potentially infinite if the controller found a score-receiving cycle — a way to keep the ball going forever. Due to the stochastic nature, this is probably unlikely. It would be interesting if such a cycle were discovered.

2.2 Description of the Pinball Board

Figure 2-1 shows the actual board that the controller will learn how to play. The little notch on the right is worth the most; it is difficult and dangerous to hit, however. Things higher on the board are worth a lot more, also, so a good controller should try to get the ball up there often to bounce around. The left passage gives a score for hitting the inside edges — the longer the ball is in contact with the edge (e.g. rolling on it), the more points are awarded. Good controllers should try to stay in this passageway a long time.

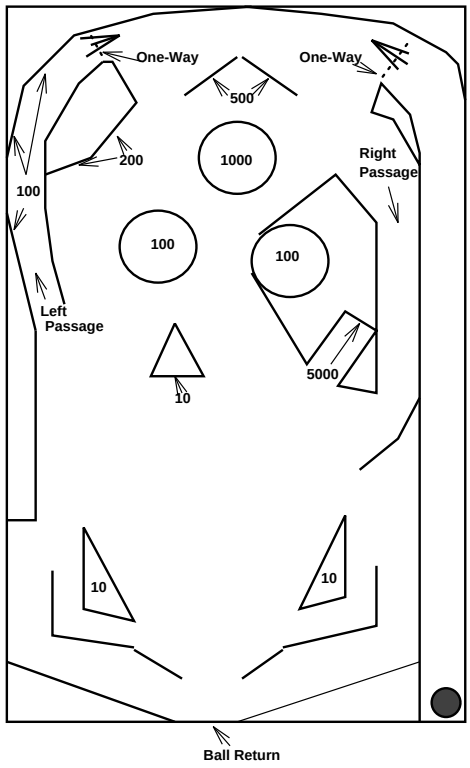

Figure 2-1: The Pinball Table

Many of the objects on the board "kick" the ball away when it strikes them. The triangles near the flippers are notorious for this. Notice the triangle in the center of the board. It often slams the ball down the center of the flippers so they can't hit it if they happen to flip the ball into it. This leaves very small "lanes" that are safe to launch into — the passageways on the right and left and very slim areas to either side of the central triangle. This is much of the difficulty of the problem — sensitivity to the control. A small difference in flip time can cause a huge difference in the outcome.

2.3 Human Results

Human subjects were asked to play the game at their leisure. I wrote software to collect and generate statistics for every player[1]. Figure 2-2 shows the pooled score

[1]This is not as easy as it sounds; there are no Unix library calls to lock a file across the filesystem so that players on two machines can't write to it at once. The solution involved renaming the file to act as a semaphore.

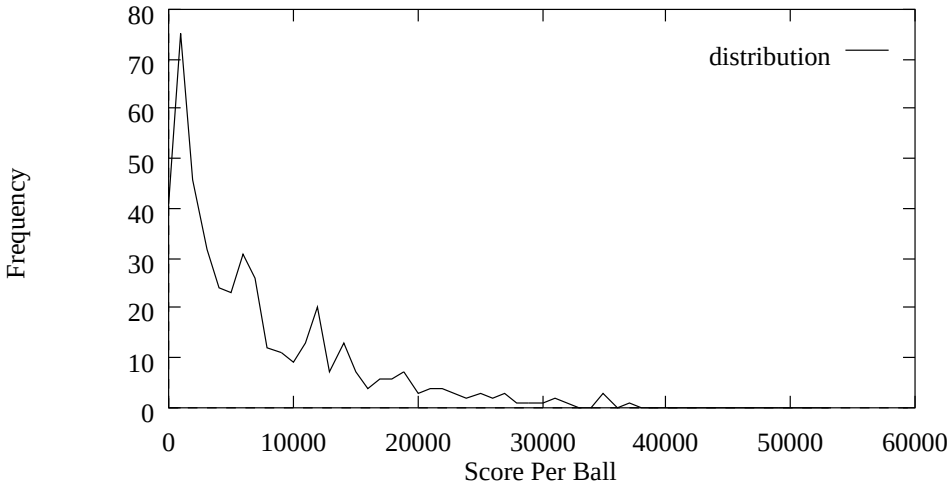

Figure 2-2: Distribution of Human Scores Per Ball

Subject	Number balls	Mean per ball
Subject 1	60	5665
Subject 2	51	8659
Subject 3	67	10494
Subject 4	263	8501
Subject 5	84	7957

Table 2.1: Means for Human Subjects

distribution for all the human players. Notice that it is difficult to get zero score, fairly easy to get one thousand or so, and seemingly exponentially more difficult to get anything higher than that. Indeed, this looks quite like the distribution for an exponentially distributed waiting time, as we might expect of a random process.

The mean of the pooled ball scores is 8388. Table 2.1 shows the mean per ball and number of balls for the subjects who took part in the experiment for more than 20 balls (enough to hopefully get a good estimate of the mean). This gives us a benchmark for a computer player. Something in the range of 8000 per ball should be considered comparable to the average human player.

Chapter 3

Pinball Simulation

It's more than a game. It's an institution.

— Thomas Hughes, 1822

This chapter covers the design of a pinball simulation. This chapter is meant to stand on its own as a description of the main skeleton of a pinball simulation. The reader may skip the chapter without any loss of continuity if they are already familiar with or uninterested in physical modelling and simulation.

The pinball framework is intentionally abstract and modular; this type of simulation begs an object-oriented approach. It is written in a fairly mathematical, abstract manner to explicitly illustrate the object-oriented structure. Other implementers can add details easily as they wish. Details on extensions and small implementation problems are discussed. We first present the data structures and then operations on them.

3.1 Design

Pinball is a dynamic system with mostly rigid collisions. Two design approaches immediately present themselves — small time-step numerical integration using spring-damper-mass models for all of the objects or larger time-step constraint-based integration using physical approximations.

The former is more physically accurate since the integration is done on a finer scale, but pinball has extremely rigid objects and collisions. This method would require a tiny time step to achieve that kind of rigidity. Such a small time step would make a real-time simulation out of the question for a normal workstation. Also, calibrating all of the surface attributes to have appropriate values is time consuming.

The latter method was chosen for the simulation. It uses a higher time-step so we don't use up as much machine time in the integration calculation. When the ball is going to hit an object on a given time-step, we stop it short and change its velocity by a reflection from the surface. The spin of the ball is not modelled. This would take another integration step and I suspect it would not improve the quality of a simulation enough to justify the extra cost. This paper describes the design and development of this constraint-based pinball simulation. Much of the ideas can be used in simulating other dynamic systems with rigid collisions.

3.2 Data Structures

A pinball board consists of several distinctive features: a ball, a set of flippers, a set of obstacles and internal state with such elements as score, balls remaining, etc. We can abstractly describe a pinball board as the five-tuple (B, F, O, L, S) where B is the ball structure, F is the set of flipper structures, O is the set of obstacles, L is the launcher, or plunger, and S contains the board's other state. A game of pinball can be described by these structures and actions performed on them.

3.2.1 The Ball, B

A pinball is abstractly represented as the five-tuple ($\mathbf{X}$, $\dot{\mathbf{X}}$, $\ddot{\mathbf{X}}$, r, $speed_{max}$) with a position vector $\mathbf{X}$, a velocity vector $\dot{\mathbf{X}}$, an acceleration vector $\ddot{\mathbf{X}}$, a radius r, and a maximum allowable speed, $speed_{max}$.

The vector quantities are variables, in most cases. In games where "nudging" the table, ramps and other objects are allowed, the acceleration may not be just a constant gravity force.

$Speed_{max}$ is used to keep the ball from going too fast. A normal table has friction, and by clamping $\|\dot{\mathbf{X}}\|$ at $speed_{max}$ we keep the ball at a physically realistic speed without having to worry about friction calculations. This number was chosen empirically.

3.2.2 The Launcher, L

The launcher in a real pinball machine is a spring that launches the ball from the launch chute onto the table. A launcher is modelled simply as a percentage P_{val} that represents how far back it is pulled. When the ball is launched, we can use this number to decide its initial velocity. The only operations on this structure are to set and retrieve the value, so we omit them from the next section.

3.2.3 Obstacles, O

An obstacle is something that impedes the motion of the ball. To simplify calculation, obstacles on a pinball board are broken into two types of primitive — circles and line segments. Both types have some attributes in common, so we can think of an obstacle as a subclass of this set of common attributes. Hence, call the common set the *attributes* of the obstacle and the structure of it (for instance, the center and radius of a circle) the *shape*.

Attributes

The attribute set contains a reflectivity constant ρ, a *score* for being hit by the ball[1], an action method *action* called when the object is struck, and a normal vector $\hat{\mathbf{n}}$.

The reflectivity constant is a non-negative real number used during a collision as an energy transfer parameter. Values less than one indicate an energy loss and greater than one an energy gain. For instance, objects like "kickers" will have a ρ greater than one.

[1]For a simulation that changes scoring states, this should actually be some function of the board state.

The *score* is used by the *action* method to change the board's score state (it tells how much hitting this obstacle is worth). An *action* is a function that takes an obstacle and the entire pinball board as arguments and is called when the obstacle is struck by the ball. This allows it to move the ball around, add scores, or any other effect one would like to have. For instance, a *ball-eater* method would reset the ball in the launcher and decrement the number of balls remaining.

The normal is used for reflection during a collision. For a line segement obstacle, it is constant. For a circle, it is changed dynamically depending on the collision point of the ball so that the collision function is the same for both obstacle shapes.

Shape

A *circle* obstacle is represented simply as its center point $\mathbf{X}_{center}$ and its radius r. The surface normal attribute for a circle is set during a collision as the unit vector along the radius towards the center of the ball (i.e. through the collision point).

A circle with $r = 0$ is useful as a *point* object. Points are used for collisions with corners of objects, as described below.

A *line segment* obstacle is represented as a *directed* edge defined by two points, $\mathbf{X_1}$ and $\mathbf{X_2}$. It is directed in that there is only one surface normal, pointing leftward as one traverses the edge from $\mathbf{X_1}$ to $\mathbf{X_2}$. This is important because lines are considered one-way — a ball hitting the back side (side away from the normal) will pass through it. This allows for "gates" and "trap doors" in the game.

Objects

One can collect these primitives into a higher-level structure called an *object.* There are certain obvious groupings we can define in terms of *obstacles* as we create a pinball board.

First, a two-sided line is useful for walls in a pinball game. This is just two line obstacles with the point order swapped on one, producing a normal in the other direction.

Second, we should put point obstacles on the ends of lines so that a ball striking

the tip of a line will reflect correctly, rather than around the surface normal of the line. For instance, a ball falling directly onto the top of a vertical line segment should not reflect around the normal, which would essentially have no effect, allowing the ball to fall through it. Rather, it should bounce upwards and toward whatever side its center of mass was on. Placing a point on the tip of the segments will have this effect. Figure 3-5 shows an example of the ball bouncing on a point.

Finally, a poly-line object can be made as a collection of connected one-sided or two-sided lines with points where they connect.

3.2.4 Flippers

A flipper is abstractly represented as a nine-tuple ($\mathbf{X_{pivot}}$, l, θ, $\dot{\theta}$, θ_{max}, θ_{min}, $type$, ρ, μ) containing a pivot point $\mathbf{X_{pivot}}$, a length l, an angle θ, an angular velocity $\dot{\theta}$, a maximum angle θ_{max}, a minimum angle θ_{min}, a $type \in \{right, left\}$, a reflectivity constant ρ and a rotational moment μ.

To simplify the drawing and calculation, a flipper is just a line segment of length l that rotates about $\mathbf{X_{pivot}}$. The flipper "down" position is at $\theta = \theta_{min}$ and the "up" position is at $\theta = \theta_{max}$.

In the implementation, a flipper at any time-step is a line segment obstacle with a point on each end. These obstacles are updated as the flipper moves. ρ is the reflectivity constant used by these three obstacles.

The moment μ is used in a collision with the ball when the flipper is moving. It is a measure of the amount of momentum transferred to the ball by the mass of the flipper and was chosen by trial-and-error to get effects similar to a real pinball table.

3.2.5 Board State

This set can contain many elements, dependent on which pinball game we are playing. For instance, there may be constants for what score to give extra balls at, what special lights are lit, etc. We ignore these in this discussion. Some elements are necessary for the simulation, and only these are described. We need to know how many balls

the player has remaining before the game ends, his score, the gravity constant of the table, the time step, and the position of the launch point.

The gravity value of the table is the component of gravity in the plane of the table. Since the table is tipped upwards in the back, the ball will have an acceleration toward the bottom of the table, and should have no component in the left-right direction. The value was chosen by trial and error to produce realistic looking ball motions.

The choice of time step value is discussed later.

3.3 Operations on the Structures

We can describe several operations on the structures described above to simulate the pinball game. As a convention, we define the coordinate system of the table so that positive y is toward the top of the table, and positive x is toward the right.

3.3.1 Launching the Ball

When the player launches the ball in a real game, the launcher provides an impulse force to the ball such that it rolls up the launch chute. We don't work out the force equations here, but one can find that the momentum transfer of the launcher to the ball is proportional to how far back the launcher is pulled, the spring constant, the ball mass and the launcher mass. Thus, the ball's upward velocity after the launcher is released can be assumed to be some constant times P_{val}. Since the ball's velocity can't exceed $speed_{max}$, we can make $speed_{max}$ this constant, so that the ball leaves at maximal velocity when the flipper is all the way back.

To keep the ball from being launched exactly the same every time, which is unrealistic, we add a small amount of noise to the constant (one to five percent is a reasonable amount). After a launch, the ball's velocity is:

$$\dot{\mathbf{X}}_x = 0$$

$$\dot{\mathbf{X}}_y = P_{val}\,(1 - R)\,speed_{max}$$

where R is a random variable producing a real number between zero and the maximum percentage of noise desired. We subtract the noise since any $\dot{\mathbf{X}}_y \geq speed_{max}$ gets set to $speed_{max}$; thus, it would have no effect if added.

3.3.2 Updating the Ball

Several things can happen when we update the position of the ball. It could just move to its new location without incident, strike an obstacle, or strike a flipper. We discuss the cases separately. We assume for these sections that the flipper objects have already been updated, if any are moving, and discuss flipper update later.

Normal Integration

Let us assume the ball will not hit anything in the current time step. The check for this is described below. In this case we just perform the normal numerical integration. We use the slightly modified Euler integration defined as

$$\dot{\mathbf{X}}_{new} = dt\ddot{\mathbf{X}} + \dot{\mathbf{X}}_{old} \tag{3.1}$$

$$\mathbf{X}_{new} = dt\dot{\mathbf{X}}_{new} + \mathbf{X}_{old} \tag{3.2}$$

where $\ddot{\mathbf{X}}$ is calculated accordingly.

The modification to the normal Euler integration is that we use $\dot{\mathbf{X}}_{new}$ in the second equation rather than $\dot{\mathbf{X}}_{old}$. This is often more stable for larger dt because it "anticipates" quick changes in the velocity. Empirical evidence with the simulation supports this. The clamping of $||\dot{\mathbf{X}}||$ at $speed_{max}$ is done after the calculation of $\dot{\mathbf{X}}_{new}$, if necessary, to keep the ball from moving too far.

Collision Detection

Collision detection can be done by calculating the intersection of the ball's equation of motion with all of the obstacles, including the flipper obstacles. If the time of an

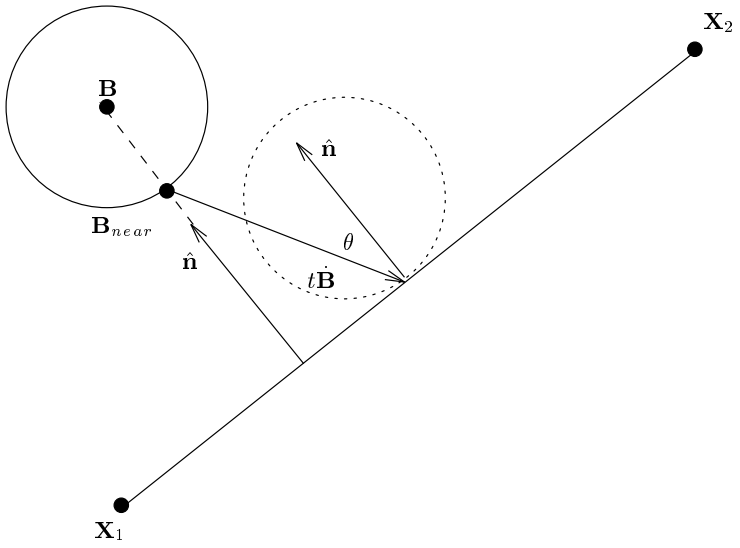

Figure 3-1: Collision detection with a line obstacle

intersection is less than or equal to the time-step, but greater than zero, there is a potential collision with that object, since if we updated the position we would move the ball through the object. Searching over all the obstacles[2], we pick the closest such intersection (if there is one), and call it the collision. If no collision, we would just finish the integration as above. If there is a collision, we would calculate the reflection as described later.

Note that since we use the new velocity to update the ball's position, we must use the new velocity to check for the collision. So we update the velocity before we do the check, but hold off on the position update until we decide if there is a collision.

There are two types of collision detection, line segments and circles. For a segment, we first calculate the inner product $\dot{\mathbf{B}} \cdot \hat{\mathbf{n}}$. If it is greater than zero, we are moving in

[2]This is not the most efficient way to search for collisions, since the ball can really only hit the objects near it; in particular, it can only hit objects within the circle centered at its position with radius $speed_{max}\, dt$. For a more efficient search, we could divide the board into a grid and calculate the set of objects reachable from each grid square ahead of time. Then, when we do the search, we just find out what square the ball is in and only check intersections with reachable obstacles. A good grid size is $speed_{max}\, dt$, since then we know that if a ball is in a certain square, it can only hit obstacles that are within that square and the three to eight squares that surround it (depending on whether it is in the middle, corner, or along an edge).

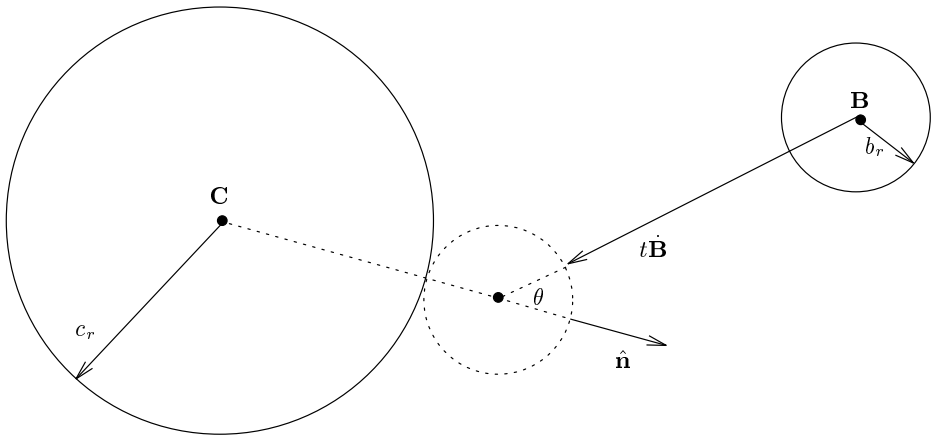

Figure 3-2: Collision detection with a circle obstacle, showing how the normal is set dynamically during a collision.

---

the same direction as the normal, so we should not check for an intersection because line segments are one-sided — we can only hit on the side with the normal facing us. Next we compute the time it takes before the point on the ball *nearest* to the line segment hits it (see Figure 3-1). Let $B = (\mathbf{B}, \dot{\mathbf{B}}, \ddot{\mathbf{B}}, r, speed_{max})$ be the ball and $\mathbf{X}_1$ and $\mathbf{X}_2$ be the endpoints of the line segment and $\hat{\mathbf{n}}$ be the normal to the segment. We calculate the nearest point $\mathbf{B}_{near}$ as follows:

$$\mathbf{B}_{near} = \mathbf{B} - r\hat{\mathbf{n}} \tag{3.3}$$

Next we solve the following system:

$$\mathbf{B}_{near} + t\dot{\mathbf{B}}_{new} = \mathbf{X}_1 + u(\mathbf{X}_2 - \mathbf{X}_1) \tag{3.4}$$

This will give us t, the amount of time before the ball hits the line which the obstacle (segment) lies on, and u, where on the line the ball will hit. Thus, if $0 \leq u \leq 1$, the ball will hit on the segment and if $0 \leq t \leq dt$ the ball will hit in the given time-step. In this case, we have a potential collision (there may be a closer one).

For a circle, the intersection is slightly more difficult. Let $B = (\mathbf{B}, \dot{\mathbf{B}}, \ddot{\mathbf{B}}, b_r,$

$speed_{max}$) be the ball and $(\mathbf{C}, c_r)$ be the circle obstacle's shape. (See figure 3-2) We calculate the intersection of a moving circle with a static one by finding out when the distance between their centers is equal to the sum of their radii, i.e. solving for t in the following equation:

$$\| (\mathbf{B} + t\dot{\mathbf{B}}) - \mathbf{C}\| = b_r + c_r \tag{3.5}$$

Since it is a quadratic, there could be zero, one or two solutions. Obviously, there is no collision for the first two cases, since for one solution they are tangent and just miss. For the two solution case, both t_1 and t_2 must be greater than zero for a potential collision to occur, or else we are inside the circle or already passed by it. We choose the smaller (nearer) solution.

Collision with Obstacles

When we decide there is a collision with a certain object, there are two things we wish to do. First, we calculate the reflection from the surface, modified by any surface parameters. Then we call the action method for that obstacle.

We move the ball so that it is tangent to the surface at the collision point[3]. If $t_{collision}$ is the time until the nearest collision, then this is

$$\mathbf{B}_{new} = \mathbf{B}_{old} + t_{collision}\dot{\mathbf{B}}_{new}$$

Next, we assume a perfect reflection since the objects are rigid — angle of incidence equals angle of reflection. To do this reflection, we need the surface normal. If the obstacle we collide with is a line, we already have this value in the attribute set. If it is a circle, then we calculate it as a unit vector from the center of the obstacle toward the center of the ball, that is

$$\hat{\mathbf{n}} = \frac{\mathbf{C}_{center} - \mathbf{B}_{center}}{\|\mathbf{C}_{center} - \mathbf{B}_{center}\|}$$

[3]Actually, one might put the ball a little out from the surface to deal with roundoff error, i.e. the next collision detection might get a $t = 0$ collision before the ball can bounce away.

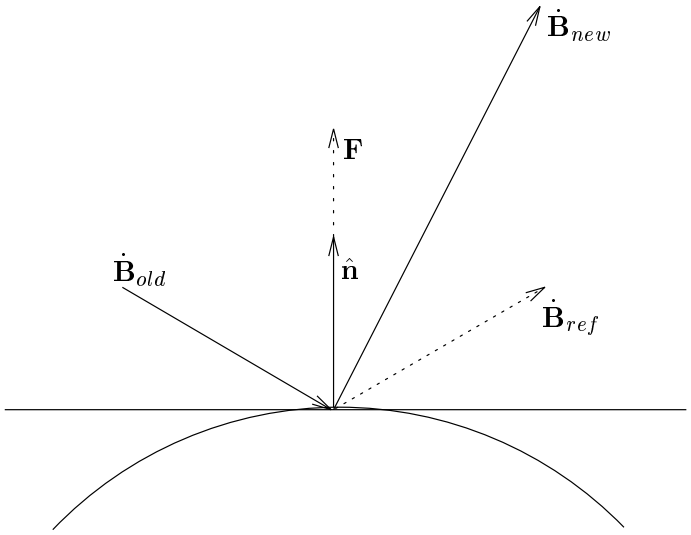

Figure 3-3: Reflection of the ball's velocity vector, showing how energy is added to the reflected vector along the normal to the surface.

Then we reflect the velocity vector of the ball through the normal and reverse its direction. A description of how to do this reflection can be found in [Foley *et al.*, 1990], pp. 730-731.

To approximate energy change in the reflection, we find the component of the new velocity vector along the surface normal, scale it by the modified reflectivity constant ρ, and add it to the reflected velocity.

$$\dot{\mathbf{B}}_{new} = \dot{\mathbf{B}}_{old} + (\rho - 1)(\dot{\mathbf{B}} \cdot \hat{\mathbf{n}})\ \hat{\mathbf{n}}$$

(see figure 3-3) Effectively, this is either forcing or damping along the normal with an amount proportional to ρ and the angle of incidence on the surface. Thus, a ball hitting an obstacle straight on will get more of a kick (or damping) than one which glances off. Recall that a *rho* of 1 is a perfect collision, greater than one is kicking, and less than one damping.

Since we have already updated the position of the ball and changed its velocity, we skip the update in Equation 3.2. If we were to do this update, we may pass through

an object, since we haven't done collision detection along the new, reflected velocity.

Finally, we call the action method with the obstacle and the board as arguments. This allows us the change the score, move the ball around, or possibly make the person lose a ball.

Collision with a Flipper

There are two cases when we hit a flipper — it is either moving or stationary. If it is stationary, then we can calculate the collision as a normal segment or circle obstacle as described above. If it is moving, however, we must deal with the extra force imparted by the angular momentum of the flipper.

The ball could hit the point obstacle on the end, the line in the middle, or the pivot point. If we hit the pivot, no energy is transferred, so we do a normal collision. It is the other two cases we discuss here.

First we perform the normal reflection as described above, along with the appropriate forcing term. Next we find the moment arm length, l_{moment}, of the flipper. This is how far along the flipper from the pivot the collision point is. If we hit the ball with the tip of the flipper, more energy is transferred than if we hit it with the middle. Using this length, we calculate the "imparted velocity" vector, $\mathbf{E}$, along the normal:

$$\mathbf{E} = l_{moment}\ \mu\ \dot{\theta}\ \hat{\mathbf{n}}$$

and add it to the reflected ball velocity, $\dot{\mathbf{B}}$. Here, μ and $\dot{\theta}$ are in the flipper representation. Intuitively, the amount of extra energy given to the ball is proportional to the speed that the collision point is moving at, $l_{moment}\,\dot{\theta}$, scaled by a moment of inertia parameter, μ. This parameter was chosen by trial and error so that the ball's exit path covered a wide range of angles (as in a real game) depending on where on the flipper the ball hit.

Again, we do not do the position update, since we have changed the ball's state.

3.3.3 Updating the Flippers

If any of the flippers are moving, i.e. the player has pressed or released the button to activate them, then we must perform a numerical integration on them as well. This is simply

$$\theta_{new} = \theta_{old} + dt\ \dot{\theta}$$

We consider the flipper's speed to be one of $\{-\dot{\theta}_0, 0, \dot{\theta}_0\}$ where $\dot{\theta}_0$ is a pre-defined property of the flipper. It should be small enough so that collision detection, described below, works. We also want to clamp θ so that $\theta_{min} \leq \theta \leq \theta_{max}$, so that the flippers stop at the correct places.

We must do collision detection for the flippers as well. When the flipper moves upward by one time-step, it may intersect with the ball. If this happens, we must preserve the rigidity and move the ball upward so that it is tangent to the surface, as if the flipper pushed it. Also, we must perform the flipper collision as described above, with a small modification — we only reflect if the ball is moving *toward* the flipper, i.e. the ball's velocity has a negative component along the flipper segment normal. Either way, we add the flipper momentum to the ball.

To test whether we have struck the ball when we update the flipper, we find the distance d of the center of the ball, $\mathbf{B}$, from the flipper line segment. If $0 \leq d \leq b_r$, then the center is within a ball's radius of the line, and we have intersected it. Then we check to see whether the ball is within the actual segment of the flipper. One way to do all this is to calculate the line intersection described above for collision detection, using $-\hat{\mathbf{n}}$ as the velocity instead (see Equation 3.6). Using the calculated t and u, one can decide whether the line intersects the circle of the ball.

$$\mathbf{B}_{center} - t\hat{\mathbf{n}} = \mathbf{X}_1 + u(\mathbf{X}_2 - \mathbf{X}_1) \tag{3.6}$$

where $\mathbf{B}_{center}$ is the ball center, $\hat{\mathbf{n}}$ is the surface normal of the flipper segment, and $\mathbf{X}_1$ and $\mathbf{X}_2$ are the endpoints of the flipper segment.

This will only work if the tip of the flipper moves less than the diameter of the ball

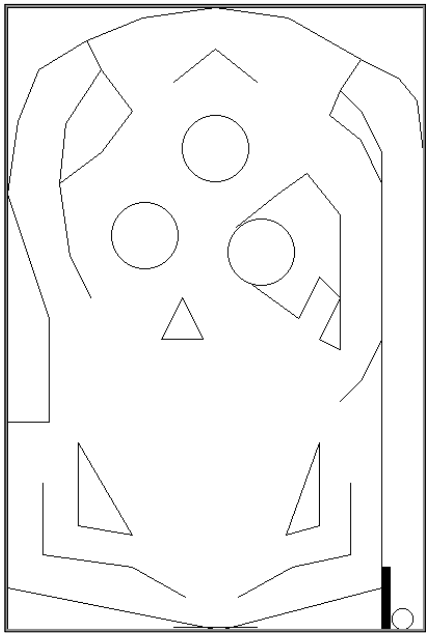

Figure 3-4: A Sample Pinball Board

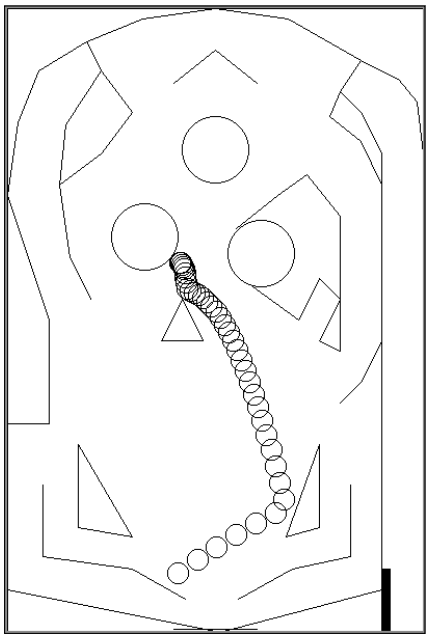

Figure 3-5: Collision With A Point Obstacle And A Kicking Bumper

on a single time-step. We must choose the flipper's natural speed, in relation to the time-step, so that this is true. If the flipper's length is l, then $l\ \sin(\frac{\dot{\theta}\ dt}{2}) < ball_{radius}$ must hold.

The flipper update can be done before or after the ball update.

3.4 Implementation and Possible Improvements

The above operations were implemented in C [Kernighan and Ritchie, 1988] and using the Xlib library. Our implementation redraws the ball every n integration steps, where n can be specified. The simulation runs at a realistic speed (compared to a real machine) on a Sun4 or DecStation 5000, even using the linear exhaustive collision detection search. In addition to the above operations, a **loadBoard** procedure that loads a text file describing a pinball board was added, so that boards could be designed in a file by hand or with a board editor (to be implemented in the future).

A screen dump of a board is shown in Figure 3-4, and a time lapse sequence

showing the ball bouncing on a point obstacle from freefall is shown in Figure 3-5. Notice the speed increase of the ball (separation between subsequent ball positions is larger) after collision with the lower bumper due to the "kicking" action.

The drawing functions in our implementation are simple — just lines and circles — to demonstrate the underlying structure of the game. One would probably like the background to be more attractive than this rudimentary line drawing. It would not be too difficult to draw a bitmap for the background and have the (undrawn) obstacles define the edges of the objects drawn on the bitmap.

Besides faster collision detection, there are several other improvements that could be made. First, we can allow obstacles to have a visibility boolean value. This will let us make things vanish and appear at various times, so the shape of the board can change state as well.

One could also make an *object* which is composed of a set of obstacles and some state — a bank of flags for instance. A collision would call the action method with the obstacle and the board, but the obstacle would know which object it belonged to. Using the flag bank example, we might only change the board state when all three of the flags in the bank have been hit, and then reset them (this uses the visibility boolean described above).

We might also implement a non-reflecting collision object, call it a *proximity* object. These would call the action method if they were collided with, but do no reflection or affect the ball. Many pinball games have "slots" that the ball can roll through for various effects. These slots could be implemented using a proximity object.

Holes are common in new pinball machines. A hole can be implemented much like a circle, but with a special collision detection. We want to know when the ball is contained within the hole, not when it hits the outer edge. That is, we want to know when the center of the ball is within the hole, since it will then fall in. This is the same collision as with a normal circle, except we want the distance between the centers to be the radius of the hole or less for a collision, rather than the sum of the radii.

3.5 Summary

Using the ideas and algorithms in this paper, a reasonably accurate real-time pinball simulation can be implemented easily. Many extensions are straightforward. Some concepts may also be applied to other simulations of a ball (or point mass) moving in an area with obstacles, such as pool (of course, angular momentum must be dealt with for pool).

Chapter 4

Representation of the Ball State

Urge the flying ball.
—Thomas Gray, 1716

This chapter will describe the representation of the ball state (a *trajectory projection*) used by the control algorithm and how data is collected from the simulation and processed into this representation. Readers not concerned with the actual calculation of the representation may read just Section 4.1 to get a description of a trajectory projection and continue to the next chapter.

Note that an important consideration in the design of the data collection system was that it should also work if camera inputs from a real pinball machine were used rather than a simulation.

4.1 Trajectory Projection

A normal computer vision system returns an (x, y) coordinate each time-step (the center of the ball). To mimic this, the data collection system samples the position of the ball from the simulation after each time step and quantizes it to some given camera resolution. Thus, we do not have the perfect, accurate state information of the ball we could get from the internal values of the simulation (that would be cheating).

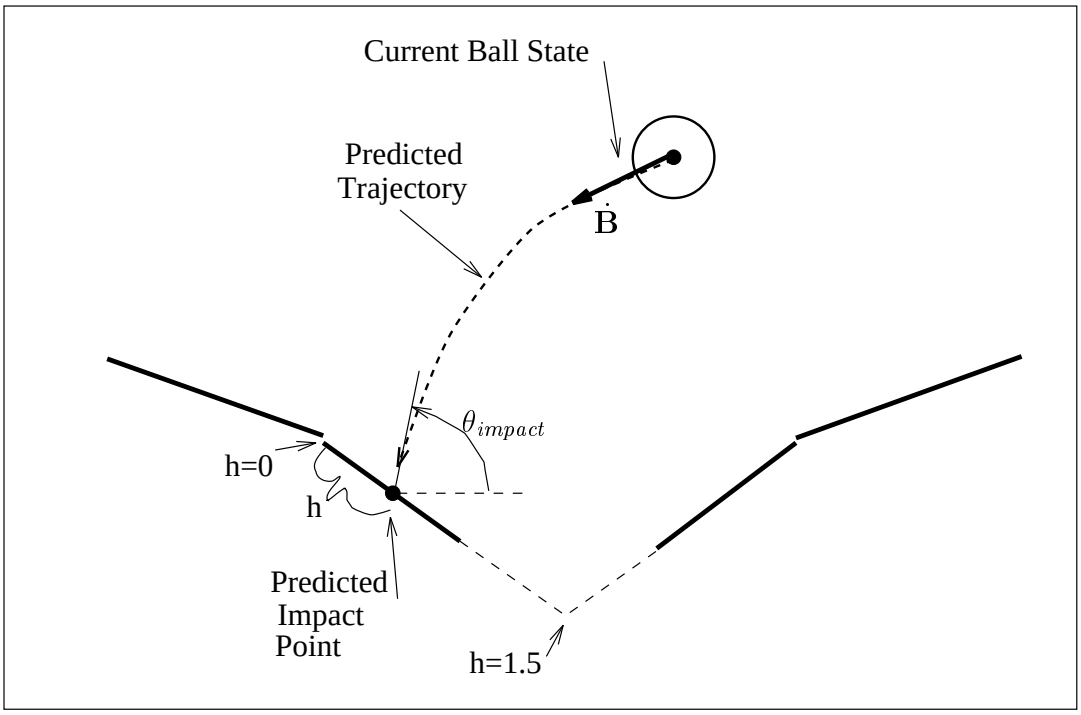

Figure 4-1: A trajectory projection is the set of impact values along the predicted ball trajectory.

Is this sequence of numbers a good representation for the control algorithm? Probably not. Using the raw ball position would make the input to the controller different every time step, though qualitatively nothing has really changed about the input between steps — the trajectory is probably the same (assuming we are in free fall). Ideally, we would like a representation that is static over time if the ball is on the same trajectory. One good reason for this is that similar trajectories probably have similar control outputs, and an algorithm should be allowed to take advantage of this. Also, if our controller takes a long time computing when to flip, we don't want to give it a different input every time step. It may get behind in processing the samples before long.

The representation chosen is called a *trajectory projection.*[1] First, we estimate the current trajectory of the ball from the position sequence. Then we project it forward in time to predict the impact with the flippers if they were to stay in the "down" position. In particular, we define the trajectory projection to be the vector (f_{impact},

[1]Andrew Moore gets credit for suggesting this representation.

s_{impact}, θ_{impact}, h_{impact}, d_{impact}). (See Figure 4-1). f_{impact} is which flipper the ball will hit and in our case is binary in value ($left$ or $right$), since we don't have extra flippers (although these could be handled similarly). s_{impact} is our estimated speed of impact on that flipper. θ_{impact} is the angle made with the horizontal that the velocity is expected to have at impact (i.e., the angle formed by the tangent of the trajectory at the impact point). h_{impact} is a parameter describing how far along the flipper's line from the pivot that the ball will hit. d_{impact} is the distance from the impact point to the ball's current position.

The first four elements of the input should not change much if the ball is on the same trajectory, though the d_{impact} will change. This d_{impact} is necessary to know when we should actually apply the action.[2]

4.1.1 Calculating the Trajectory

The ball will follow a parabolic trajectory that opens downward since we assume that the only acceleration is that of gravity. So the ball's trajectory will satisfy:

$$\ddot{\mathbf{B}}_x = 0 \tag{4.1}$$

$$\ddot{\mathbf{B}}_y = -g \tag{4.2}$$

where g is the gravity constant of the table. Integrating these twice produces a parametric system of equations for the ball state:

$$\mathbf{B}_x = v_{x0} \cdot t + x_0 \tag{4.3}$$

$$\mathbf{B}_y = \frac{-g}{2} \cdot t^2 + v_{y0} \cdot t + y_0 \tag{4.4}$$

where v_{x0}, v_{y0}, x_0, y_0 and g are the parameters that we need to solve for to specify the actual trajectory.

[2]In retrospect, it might have been better to use the estimated time left before impact instead of distance to impact, since this is not dependent on the camera resolution. Still, the values will be close to proportional at a given ball state, so it should not matter.

We use the method of general least squares (GLS) to calculate these values, fitting a quadratic through the y-coordinates over time and a line through the x-coordinates over time. That is, given a time-sequence of n (x,y,t) ball location coordinates, we do a least squares fit through each of the two sequences (x_i, t_i) and (y_i, t_i). Since GLS is described extremely well in *Numerical Recipes in C* [Press *et al.*, 1988], we omit the details.

This direct use of GLS tends to be noisy since we often only look at a small part of the parabola, and there are not enough points to make a good fit for the quadratic coefficient. To make the prediction better for smaller sections, the gravity constant of the pinball table is assumed fixed and calculated in advance. We can have the GLS fix this parameter at the given value and give us the best values for the other two parameters. *Numerical Recipes* [Press *et al.*, 1988] talks about this option in the GLS section also.

Our implementation uses the known value of g directly from the simulation, but this number can be computed in the calibration phase of a real table by letting a ball roll from rest several times and estimating the acceleration. The simulation was actually tested with this constant 50% away from the real value, and it still did quite well. Hence, this rough table calibration should be enough to get good results.

Choosing the number of points to use in the least squares is also slightly tricky. We want the least squares calculation to work well under various ball speeds. For example, a fixed number is bad since at slow speeds, we won't get enough differentiation between points to get a good fit. At high speeds, we'd rather not use a lot of points since we could have just had a collision and many of the old points might not be on the current trajectory. To avoid these problems, we find the first n that satisfies the following rule, where $\mathbf{B}_i$ is the ith point in the past, assuming the current ball location is $\mathbf{B_0}$:

$$||\mathbf{B}_n - \mathbf{B}_0|| \geq d \tag{4.5}$$

where d is a parameter chosen by the experimenter to acheive good results. We found about half a flipper length to be a good value. According to this rule, at slow speeds

we use a lot more points (and have more time to do the calculation, so can afford it) and at high speeds much less, as desired.

If we collide with a bumper toward the bottom of the screen, the prediction may be incorrect for a few steps while the points from the old trajectory are still being used in the prediction. The old points will be flushed out soon enough and the prediction will become better over time, so this is not a problem. As another option, we could do a *Chi square* test on the calculated parameters to see how well the points fit the trajectory model. If they fit well, all the points are probably on the same freefall trajectory. If there is a collision, our model will suddenly not fit well.

Notice that if the ball is toward the top of the board there are a lot of collisions occurring that we do not care about. To avoid this wasted computation, we only do this trajectory calculation when we are in the critical area — the area of the board above the flippers where there are relatively few objects. This area can be decided empirically and defined as some y value above which we do not do the calculation.

4.1.2 The Problem of Rolling

We have overlooked a major problem until now — if the ball is rolling on the "shelf" beside a flipper, or on the flipper itself, the least squares solution will fail to fit a good parabola since it assumes a constant acceleration in the y-direction and none in the x. When we are rolling, there is a normal force applied to the ball, which has components in both directions. Thus our fit would be incorrect, and indeed has the effect of having a changing prediction over time — exactly what we do not want! To avoid this, we make rolling on the flipper a special case.

We can check for the rolling case several ways. A good way is to put some sort of sensor in the channel above the pivot point. When the ball crosses it, it trips the sensor and we know that the ball will be in a rolling state. Another method would be to put a contact sensor on the flipper. This could not only tell us if the ball was rolling, but could also tell when the ball has hit the flipper. We simulate the latter sensor.

There are many ways to actually handle the rolling case. We could make a special

case in the controller. Instead, we would like the same controller to work for all trajectory projections, rather than making a special controller for rolling. For this reason, we define an "appropriate" trajectory projection for the rolling state.

First, we assume it will impact the point in the center of the flippers ($h_{impact} = 1.5$ in our simulation), which it is clearly headed towards when rolling. We also set f_{impact} to be the flipper we are *not* rolling on. This makes sense since we really are headed *toward* the other flipper. We set s_{impact} by using a technique similar to how we choose which points to use least squares on — we find the first point $\mathbf{B}_n$ at least some small distance ϵ (say 10% of the flipper length) away. We calculate the real distance between the current ball location $\mathbf{B}_0$ and $\mathbf{B}_n$ — call it d — and divide it by the amount of time that passed between the two points ($n\,dt$, assuming constant time-step of dt). This is an approximation of the current speed, which should be close to the impact speed. Finally, the impact angle is set to be the angle that the "down" flipper forms with the horizontal.

4.1.3 Calculating the Trajectory Projection

Obviously we need several important pieces of information besides the ball location sequence — namely, the location of the flippers in the "down" position — otherwise we could not predict impact values. In order to do this with a real machine, simple calibration will have to be done beforehand using markers to define the pivot and tip of each flipper. For the simulation, we use the real values.

Now we can calculate impact points on each flipper line using the parametric ball trajectory we calculated above. Since the ball might still be hittable if isn't going directly at a flipper, (for example, coming in at an angle toward the center) we also consider the projection of each flipper segment to the point where these projections intersect as a valid collision point. (see Figure 4-1). Thus, a collision point anywhere on the 'V' formed by the flippers and these projections is considered "hittable." It is straightforward to calculate the intersection of the trajectory with the flipper lines

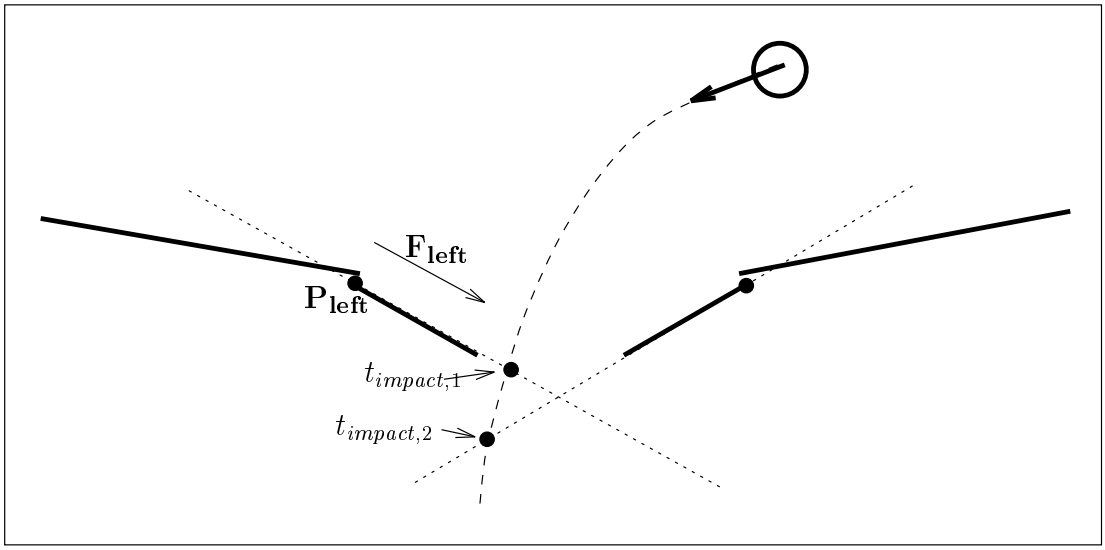

Figure 4-2: Calculation of the Trajectory Projection

by setting the equation of the flipper line equal to the ball trajectory:

$$\mathbf{F}_x \cdot h + \mathbf{P}_x = v_{x0} \cdot t + x_0 \tag{4.6}$$

$$\mathbf{F}_y \cdot h + \mathbf{P}_y = \frac{-g}{2} \cdot t^2 + v_{y0} \cdot t + y_0 \tag{4.7}$$

where $\mathbf{P}$ is the pivot point of the flipper and $\mathbf{F}$ is the vector pointing from the pivot to the end of the flipper as shown in Figure 4-2. Solving this system for t and h gives two times of impact with each flipper line, since it is quadratic. We ignore the times that happened in the past, because they clearly do not matter. This leaves a future impact time of the ball onto each flipper line. The nearer of the two is the correct time, giving us which flipper the ball is going to hit. We also use the h that we calculated for that flipper as the h_{impact} in our trajectory projection. In Figure 4-2 we would choose $t_{impact,1}$ because it is the closer one.

Using this t_{impact} we can use the derivative of the ball trajectory:

$$\dot{\mathbf{B}}_x = v_{x0} \tag{4.8}$$

$$\dot{\mathbf{B}}_y = g \cdot t + v_{y0} \tag{4.9}$$

to calculate the speed of impact, s_{impact}. This is just the magnitude of the ball velocity evaluated at t_{impact}:

$$s_{impact} = \sqrt{{v_{x0}}^2 + (g \cdot t_{impact} + v_{y0})^2} \tag{4.10}$$

We calculate the angle of impact, θ_{impact}, by finding the angle made by the tangent of the trajectory (the derivative) at the point of impact:

$$\theta_{impact} = tan^{-1}\left(\frac{g \cdot t_{impact} + v_{y0}}{v_{x0}}\right) \tag{4.11}$$

Finally, d_{impact} is simply the distance from the current ball position to the collision point. We find the collision point by evaluating the trajectory equation at t_{impact}.

Chapter 5

The Controller Architecture

So we grew together,
Like to a double cherry, seeming parted,
But yet an union in partition;
Two lovely berries moulded on one stem;
So, with two seeming bodies, but one heart.
— Shakespeare, *A Midsummer Night's Dream*

This chapter describes an abstract pinball controller architecture (henceforth called a *controller*) and in general how a controller should use the trajectory projection to decide which flipper to flip and when to flip it. This is an architecture rather than an algorithm because it defines a class of control algorithms, not the specifics of them. After defining the architecture, we discuss the implementation we chose to use.

A pinball controller is really made of two separate components — a launcher controller and a flipper controller. The launcher controller is quite simple. We describe it and assume that it runs on its own in parallel with the flipper controller. This chapter concentrates on the flipper controller, which is much more interesting.

5.1 Launcher Controller

The launcher controller's job is to pull the launcher a certain distance and let it go whenever the ball is in the launch chute. We assume that we have a sensor for this. The question is, how far should it pull it? Clearly, we want to launch the ball so that we have a chance to hit it. A launch that ends up losing the ball before we can take a swing at it is not very good.

We chose the simple solution of trying a finite set of values and finding out which one led to a hittable state the highest percentage of the time. We need a percentage because there is noise in the launch, so the ball may end up in a different initial state for the same launcher value. Running this search over twenty different values, for ten times per value, the launcher controller decided that pulling it 95% of the way back was the best. This launch actually causes the ball to roll down the right passageway (see Figure 2-1) and directly at the left flipper every time. Thus, we are guaranteed a chance to hit it every time.

5.2 Flipper Controller

This section describes the flipper controller architecture and motivations for the design. We use the term *hittable state* sometimes. A hittable state is defined as a trajectory projection that will hit on the 'V' formed by the two flippers (even though a ball going straight down the center is really unhittable; the exact notion of hittable is too hard to describe simply). Any other trajectory projection is considered unhittable and is actually ignored by the flipper controller. We don't want to waste time trying to hit balls that are unhittable.

5.2.1 Analysis and Design Motivation

The flipper controller has to decide when to flip and which flipper to flip. To inspire the design of such a controller, the author played pinball and did some introspection. A simple approach to the problem was to predict the impact point, and flip the correct

flipper when the ball was a certain distance away. We use a similar method for the flipper controller.

The flipper controller will be given a trajectory projection (f, s, θ, h, d), as described in Chaper 4. Using these values, we must decide when to flip and which flipper to flip. As suggested in the previous paragraph, we calculate when we are going to flip as a *critical distance* from impact ($d_{critical}$) at which we will flip. $d_{critical}$ will be a function only of f, s, θ and h. When $d \leq d_{critical}$, the controller will activate a flipper. We don't specify how it will choose the flipper or the actual function used to calculate $d_{critical}$.

The final issue is when to release the flipper. We choose to release it after the ball has been successfully launched, or is lost. We define how a launch is discovered in Section 5.2.5. Another way could be perhaps to decide an amount of time to flip for, but this seems too difficult and not very useful.

5.2.2 The Action Function

Abstractly, then, a flipper controller can be reduced to a function from a trajectory projection (minus the d) to a $d_{critical}$ and choice of flipper, called the *action*: $A : T \rightarrow < d_{critical}, f >$. T here is called an *input*, and is the trajectory projection minus the d value, i.e. the vector (f, s, θ, h). We will often call the controller function A the *action function*, since it computes the action.

Given that we are trying to get a good score, clearly there are some actions that are better than others. The problem is that when we approach the problem initially, we have no idea what "good" values are. Indeed, we have no idea which values will actually succeed in even hitting the ball. Also, the problem is stochastic, so that some "good" values may only be good a fraction of the time. We assume that there exists an action function that gets the best score possible on average. Call this an *optimal action function*. The problem is to decide how to find that function. First, we formalize the general structure we have developed.

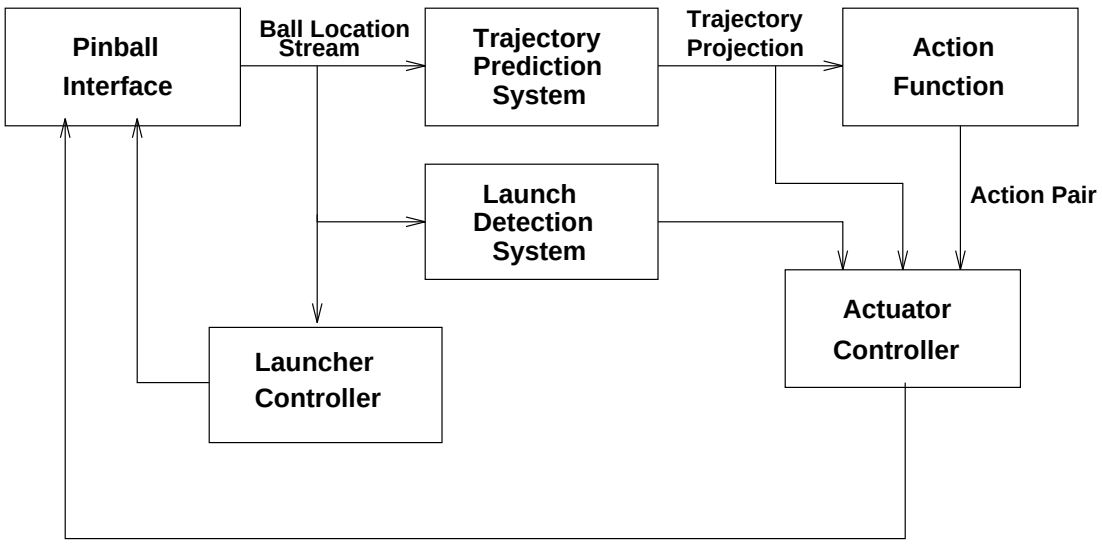

Figure 5-1: The flipper controller architecture

5.2.3 The Controller Architecture

Figure 5-1 gives a graphical description of the flipper controller architecture. The pinball interface — either a simulation or a combination vision system/flipper actuator interface — produces a stream of ball positions. This is passed to three subsystems: the trajectory prediction system, which produces a trajectory projection; the launch detection system, which tells if the ball was launched that step; and the launcher controller, which launches the ball if it is in the launcher.

The trajectory projection is then passed on to the action function, which calculates the action pair, $< d_{critical}, f >$. This action pair, the trajectory projection, and the launch system output are given to the actuator controller.

The actuator controller activates flipper f if $d_{critical} \leq d_{impact}$. It releases any activated flippers when a launch is detected or the ball was lost.

5.2.4 A Piecewise Linear Action Function

The main issue is that the controller be highly reactive when a ball is in play — that is, it must respond quickly to either a change in ball trajectory or to a fast-moving ball. Hence, the action function should be simple and swift to calculate. Our

intuition is that an optimal action function is complex and probably nonlinear and discontinuous due to the sensitivity of the system. For instance, two ball trajectories that begin near each other may lead to quite different outcomes because of all the obstacles on the board. One may get a great score and another may end up losing the ball. Since it would probably be impossible to find such an optimal function in any reasonable amount of time, our only hope is to try and approximate it. Also, calculating this function for a given input is likely to be costly.

We propose a piecewise linear approximation of the action function. That is, we divide the input space into disjoint regions called *partitions* and use a linear combination of the input vector elements as a local model of $d_{critical}$ for that partition of the space. In other words, we assume that:

$$d_{critical} = a_1\,\theta_{impact} \;+\; a_2\,h_{impact} \;+\; a_3\,s_{impact} \;+\; a_4 \tag{5.1}$$

Notice that $d_{critical}$ can be negative. Rather than use the absolute value, we allow negative values. Such controllers are unlikely to do well since they will never flip and should be eliminated from an optimal controller search before long. In addition, for each partition we store which flipper to flip. We always flip the same flipper in a given partition. The motivation is that nearby inputs may have linearly similar optimal actions. This seems likely, since there are certain areas of the board that are high in score that we can reach from different inputs. The hope is that if we can hit the spot from a certain input, we can probably hit it from a input where the impact angle, say, is slightly different by merely changing the $d_{critical}$ by some constant times the difference in angle. Another good reason for a piecewise linear approximation is that it is swift to calculate; given the input, we can quickly find what partition we are in and calculate a linear function.

How should we partition the space? A ball going at the left flipper probably has a much different optimal action than one going at the right, so clearly we should partition by flipper. We also choose to partition by the impact angle, since at some angles we can hit certain objects, but not at others. It seems reasonable that the

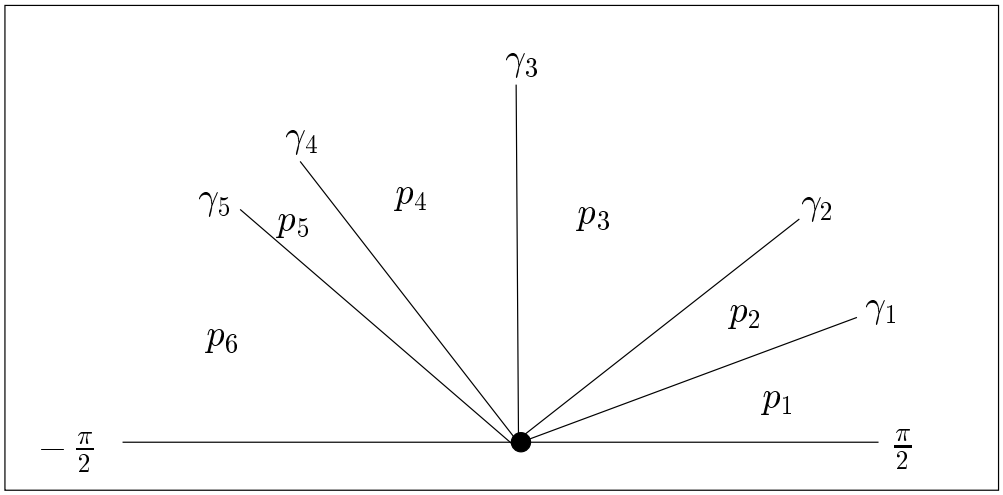

Figure 5-2: An arbitrary angle partition for $k = 6$

local model should be different for these different angle regions. The space could be divided by h also, if desired, but we choose to stop here as the optimization phase (see Chapter 6) is sensitive to the number of partitions. The actual number of angle partitions and the partition regions are parameters of the action function.

Formalizing these notions, we define the action function A as the set $\{M_{i,j} : i \in \{0,1\},\ j \in P\}$ where $M_{i,j}$ are the local linear controllers for each partition, called *minicontrollers*, and P is the set of angle partitions, defined below. i indexes the flipper partition (f in the input vector).

A minicontroller is defined as the five-tuple (a_1, a_2, a_3, a_4, f) where a_i are the coefficients of the model described in equation 5.1 and f is which flipper to flip.

P is the set of angle partitions, $\{p\}$. We partition the interval $[-\frac{\pi}{2},\frac{\pi}{2}]$ into k regions using some set of $k-1$ angles, $\{\gamma\}$ (see Figure 5-2). Thus, angle partition i, p_i, consists of all inputs whose θ_{impact} is in the interval bracketed by the nearest two angles in the set $\{-\frac{\pi}{2}, \frac{\pi}{2}, \gamma_i\}$. In other words, p_1 is between $\frac{\pi}{2}$ and γ_1, p_2 is between γ_1 and γ_2, etc. as in the figure.

It is simple to find the appropriate minicontroller for a given input vector and calculate the action $< d_{critical}, f >$ quickly. Notice that there are many unspecified parameters: the five values in each minicontroller, the number of angle partitions k,

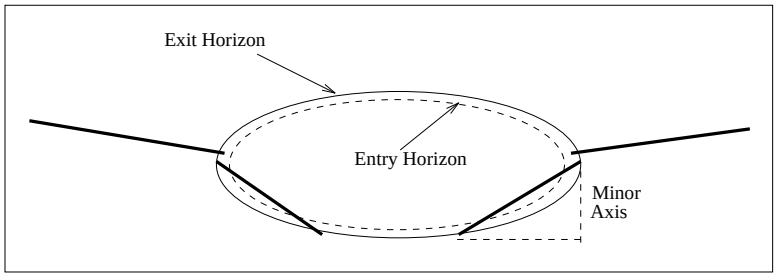

Figure 5-3: The Flipper Influence Ellipse

and the $k-1$ angle partition borders, $(\gamma_1, \ldots, \gamma_{k-1})$. There are $2k$ minicontrollers with five values each, plus the value of k itself and the $k-1$ partition borders. This amounts to $11k$ parameters; quite a formidable amount. Now that have an arbitrary action function defined, we must find a good one; this is the topic of Chapter 6.

5.2.5 Launch Detection System

How do we know when the ball gets launched so that we can release the flipper? We could put sensors on the flippers, perhaps. Instead, we use a simple vision-based algorithm. In general, when the ball is within some *influence range* of the flippers, then is not within it any longer, two things could have happened — it could have gone out the bottom and gotten lost or out the top because it was launched. This seems like a good way of deciding when the ball is launched, and thus that a flip has occurred. When this happens, we know that the last minicontroller that was responsible gets the reward or penalty for the launch.

We choose the *flipper ellipse* shown in Figure 5-3 as our range of influence. A launch is defined as the ball entering the ellipse and then leaving it. We also add hysteresis to the launch criterion so that noise in the ball tracker will not consider a jitter to be entering on one step and then leaving the next. Specifically, we say that the ball must penetrate the inner ellipse to enter the range of influence and must exit the outer. Thus, mere vision noise will not get false flips.

There is a problem with this method, however. If the ball bounces gently off the flipper, reaches its apex within the ellipse, and then falls toward the other flipper, we

do not notice that it has launched and gone to another minicontroller (and thus leave the other flipper up, which could be wrong). To fix this, we also use the current y velocity as calculated by the General Least Squares fit in Section 4.1.1. If the ball is going down for a while, then suddenly it is going up, we can interpret this as a launch — either the flipper hit it or it bounced off the flipper. We must be careful not to use this criterion when the ball is rolling, however, since the calculated y velocity is not accurate then. We leave the ellipse method to deal with a rolling ball launch.

Chapter 6

Optimization of the Action Function

Attempt the end, and never stand to doubt; Nothing's so hard, but search will find it out.

— Robert Herrick, "Seek and Find", 1591

This chapter will describe two methods used for optimizing the action function described in the previous chapter — *stochastic hill-climbing tournament* and *Hoeffding Relay Race* [Maron, 1993]. Before describing these particular algorithms, we describe the paradigm and simplifications we use for both methods.

6.1 Description of the Optimization Paradigm

In order to optimize an action function efficiently, we need to make several constraints and assumptions. For example, our parameters are real numbers; surely we can constrain our search over some small interval for each. By efficient, we mean that we want fast on-line performance — we want the system to learn a reasonably good solution quickly in real-time, even though the solution may not be "optimal" (i.e. the best available). We might also expect some kind of continuous improvement over time, but do not require it. Hence, we concern ourselves only with finding a "good"

solution quickly, where a "good" solution will be defined only as "comparable to human play." Perhaps "optimization" is a misnomer for this type of solution, but we will henceforth use the term loosely to mean an "improvement search," rather than a search for a true optimal solution.

6.1.1 Fixed Partitions

For the purposes of this research, we restrict the actual space of action functions we will search over in a given trial. In particular, we fix the number of partitions and fix the actual partition. That is, we won't allow the number of partitions or the angles that define the partition to vary in a given search. Notice that this may constrain the best solution we can achieve, however. The methods below could be extended to take this variation into account, but at a performance cost. Future research could add this extra degree of freedom and see if it really produces an increase in performance.

To make explicit the simplification actually used in our implementation, the angle was partitioned uniformly, so that each minicontroller had an equal-sized manifold of space to control. This kind of assumption is not rigorous; rather than this arbitrary partition, we would like to locate the sufficiently smooth regions of the function and use those are partitions. This is difficult and could be investigated in future research.

6.1.2 Reducing Dimensionality With Optimal Substructure

Higher numbers of partitions increases the number of parameters, making the search space higher in dimensionality. A higher dimensional problem is in general *exponentially* more difficult to optimize. In particular, if there are d parameters and we assume that each is discrete, say to n levels, then the search space is of cardinality n^d. If we could find some kind of simplification, however, we may reduce the dimension. This is where our piecewise linear solution comes in handy.

The action function consists of some number of minicontrollers, each of which has complete control over some area of the input space. It alone is responsible for getting a good score for any inputs in its partition. Of course, its action also sets up the

ball for another minicontroller to take over (if the ball isn't lost). Since there is a stochastic element involved, the exact same input and control may lead to a different next controller. It would be interesting to pretend that this is a Markov process and use a dynamic programming solution. Research by [Moore and Atkeson, 1992] has shown that this works for similar problems. (This is the subject of future research plans. See Section 7.2.)

Instead, we make the assumption that it really does not matter what the next controller is, as long as there is one (it doesn't lose the ball). Thus, every minicontroller should be greedy and try to maximize the score that it receives for any input it gets. In this way, we reduce finding a good action function to searching for a good minicontroller for each partition. Thus, our best controller will be the set of best minicontrollers. Hence, we assume that the best controller has an optimal substructure which we will exploit.

This greedy strategy effectively ignores any effect other minicontrollers might have on a given minicontroller's ability to get a good score. Hence, each can be optimized independently of the others. This massively reduces the search space. Each minicontroller has five parameters — four continuous gains and a binary flipper value. Assuming that we quantize the continuous parameters to n values each, an upper bound on our search becomes $O(k\, 2n^4)$ which is much better than the the normal $O(2^k\, n^{4k})$ we would expect. Indeed, it is linear in k, rather than exponential.

6.1.3 Constraining Parameter Range

As suggested above, we also limit the range of values that each parameter can have. Some optimization methods would call for quantizing the parameters over a certain range, but the two methods below will assume only that they fall in some interval and leave them continuous. In the implementation, we choose the range symmetric around zero. The intuition is to make it "wide enough, but not too wide." Knowing bounds on each input element and an idea of an appropriate $d_{critical}$ range emprically (like it shouldn't be larger than half the board, say), we can choose the range so that the gains give each element enough power to swing $d_{critical}$ throughout this full range.

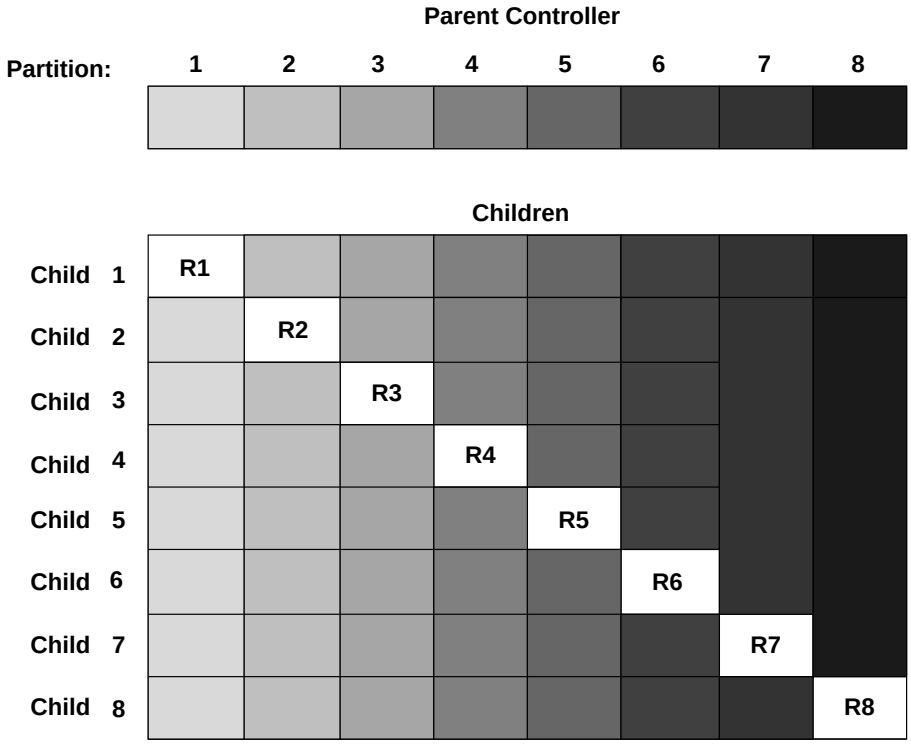

Figure 6-1: Producing a Generation From the Parent

6.2 Stochastic Hill-Climbing Tournament

The name for this method was chosen because it resembles both hill-climbing and a tournament. The idea is that we simulate a *tournament* of n rounds between some group of controllers. Each controller is run for b balls. The controller with the highest mean score per ball for that round "wins" and goes on to the next. The motivation is that this is often how good players among people are found.

There is a twist; it is a family competition. One controller in each round is the **parent** and the rest are the **children**. The children are derived from the parent and then compete against each other and the parent. This is where the hill-climbing aspect comes in.

6.2.1 Description of the Algorithm

Given the parent has k partitions, we compute its k children by randomizing one and only one of its minicontrollers for each child (see Figure 6-1). The motivation is

1. Let p be some initial parent controller. Let k be the number of partitions in p.
2. Repeat n times:
   (a) Create k children, c_i, of p as described.
   (b) Calculate mean score over b balls for p and each c_i.
   (c) Let p equal the controller with the maximum mean.
3. p is declared the winner of the tournament.

Figure 6-2: Stochastic Hill-Climbing Tournament

the independence assumption and optimal substructure described above. We expect that if we change just one minicontroller and it does better, then it is that minicontroller's achievement and therefore we've found a better minicontroller, and hence a better controller. Likewise, worse behavior will is likely to be that minicontroller's fault.

Formally, let P be the set of k minicontrollers of the parent, $\{M_i\}$. The ith child of P is then defined to be:

$$\left(\bigcup_{j \neq i}^{n} M_j \right) \cup r_i \tag{6.1}$$

where r_i is a random minicontroller. To produce a random minicontroller, we pick four random gains uniformly over the interval of allowable gains and a random binary number for the flipper value.

Thus, we are effectively performing a stochastic hill-climbing — we jump randomly in each minicontroller "direction" and pick the best new position, or stay in the same spot if all the jumps are worse. The winner of the round becomes the parent for the next round. Pseudocode for the algorithm is shown in Figure 6-2.

Choosing the parameters n, b and the initial parent p is up to the programmer. The run time of the algorithm is $O(b\,n\,(k+1))$. Given that it takes on the order of minutes to run a controller that hits the ball a few times on a DECStation 5000 (about as long as a real pinball machine, since the simulation is real-time), it takes about $(b\,n\,(k+1)$ minutes to run this tournament. For our trials, we chose n and b to

fit some given amount of time we wanted it to run in, like a day. Finally, we chose p by doing a quick random search for a controller that managed to get a score above a certain value in only one ball. We chose the value so that it basically had to hit the ball once off the launch. This gives at least a starting place that has some merit. A random controller is extremely likely never hit the ball, and we would waste our time waiting for a good one to come around in the tournament.

6.2.2 Strengths and Weaknesses

The main advantage of this algorithm is its obvious simplicity. There are clearly many weaknesses, however. First, the controller that wins the tournament might not be the controller that achieved the best mean during the tournament. To deal with this, we could globally store the best controller in the tournament so far, so that we could end up with two winners at the end, the "winner" and the one with the highest mean.

Another problem is that we may waste a lot of time on a controller that is clearly poor, for example one that misses constantly. A way around this would be to use statistical techniques to stop a bad controller before it ran for all b balls. This concept is actually used in the next optimization method.

This method is not the most direct use of our optimal substructure. It is more indirect, changing a minicontroller and then looking at how the entire controller changes, not that particular minicontroller. For this reason, and the fact that we cannot have a large b due to computational constraints, the variance of the randomness of score could be the cause for the difference in score obtained for a new child. Only really large increases or decreases will be truly be significant for a small number of balls. This suggests statistical analysis, also.

If we would like the mean to be fairly accurate and thus a good predictor of controller ability, we need a fairly large b. But n should also be large enough to get reasonable chance for the stochastic search to find a good minicontroller. So we must trade-off time for accuracy. Ultimately, it is a fairly slow algorithm.

Another problem is that the more partitions we have, the longer this algorithm

takes. It also does not take into account that some minicontrollers may rarely or never be activated, and wastes its time changing them and running the controller again.

Finally, this algorithm is not guaranteed to converge. Perhaps if we added a "temperature" parameter, which defines how far we are allowed to jump away from the parent minicontroller we are replacing and slowly reduced it, we might achieve a convergent algorithm. The algorithm as described allows us to jump anywhere in the minicontroller space every time. This kind of idea is called *simulated annealing*. A good discussion can be found in *Numerical Recipes* [Press *et al.*, 1988]. Since it takes so long to run few generations, we have no need for such a parameter currently.

6.2.3 Results

Running the tournament for only five or six generations takes a long time. The first trial used twelve partitions (six angles $\times$ two flippers) and ran for only six generations. The best controller it found by then had a mean over thirty balls of 4962. Trial two ran for five generations, using only four partitions, and got a mean for 30 balls of 3395. Table 6.1 shows the population mean for each generation as well as the mean of the winner for each generation. This population mean is not a great indicator of progress, since the randomized minicontroller might be extremely poor, but it does give some kind of idea of progress. There is a general improvement trend which quickly levels off. These would have to be run longer to continue improvement. It is really hard to do a successful hill-climbing with only six generations, so it is not surprising we only did this well.

6.3 Hoeffding Race Optimization

Adding statistical analysis should speed our search; we can eliminate controllers that are significantly worse than others. The algorithm we describe in this section uses a modified version of the *Hoeffding Race* optimization method described by Oded Maron[Maron, 1993]. It also incorporates several other important ideas in order to

Trial	Generation	Pop. Mean	Best Mean
1	1	1989	3433
	2	3912	5910
	3	3613	5879
	4	4325	8288
	5	5449	8017
	6	4692	6449
2	1	1903	2683
	2	1908	3731
	3	2371	4619
	4	3451	7436
	5	4029	6011

Table 6.1: Tournament Results

speed up the search and perhaps find better solutions.

6.3.1 Some Useful Concepts

We would like to follow intuition and do a direct optimization of each minicontroller separately in order to really take advantage of the optimal substructure assumption. But how do we know how well a given minicontroller is doing? We need some sort of criterion to compare different minicontrollers in the same given partition. For this purpose, we define the *mean score per flip opportunity*. It is not so obvious what a *flip opportunity* is.

The function to optimize When the input falls in a given partition, it is that minicontroller's responsibility to hit it — it has an opportunity to hit the ball. We keep track of the score received between that flip and the next opportunity for some other minicontroller (perhaps the same) to flip. That is the score the minicontroller receives for that flip. If there is no next opportunity to flip, i.e. the ball is lost, then the minicontroller gets a penalty for losing it and none of the score. This gives a kind of one-ply look-ahead that the last algorithm did not have. We don't want to reward the minicontroller for a score if it loses the ball getting it. That is not useful to getting

a good mean score per ball, which is what we really want to optimize. Assuming all the minicontrollers have good means per flip, the mean per ball should be high.

Minicontroller statistics calculation We will need to calculate statistics for each minicontroller; specifically the sample mean m_n and sample standard variance σ_n^2 over n flip opportunities. We want this to be incremental; we do not want to have to store all the points and use them to calculate it every time. We derive incremental functions for these in Appendix A, storing only a constant amount of state.

6.3.2 Description of the Hoeffding Race

Confidence Intervals The Hoeffding Race algorithm uses *confidence intervals* around the sample means of a set of different independent identically-distributed (i.i.d.) random variables to decide if the true mean of one of the variables is significantly different from another. A two-sided confidence interval of confidence c is just a symmetric interval around the sample mean that the random variable will produce values within with probability c. The useful fact is that the interval gets smaller as n, the number of samples, increases. Hoeffding has a strong formula for finding these intervals [Hoeffding, 1963]. We refer you to Appendix B for the actual calculation of the intervals using the Hoeffding formula as described by [Maron, 1993].

The Hoeffding Race Figure 6-3 shows a set of minicontrollers from the same partition of input space. We assume that each minicontroller has run some (perhaps different) number of times, and we have calculated their sample mean and c confidence interval. Since we are trying to maximize the mean score, we don't want to waste computation time on significantly poor minicontrollers. Any minicontroller whose top confidence interval endpoint is below the low confidence interval endpoint of another minicontroller has a small probability chance for its true mean to be larger than the other true mean, since it is unlikely for points to fall higher than its high endpoint. Clearly we can eliminate these from the running with a small chance of discarding a better minicontroller.

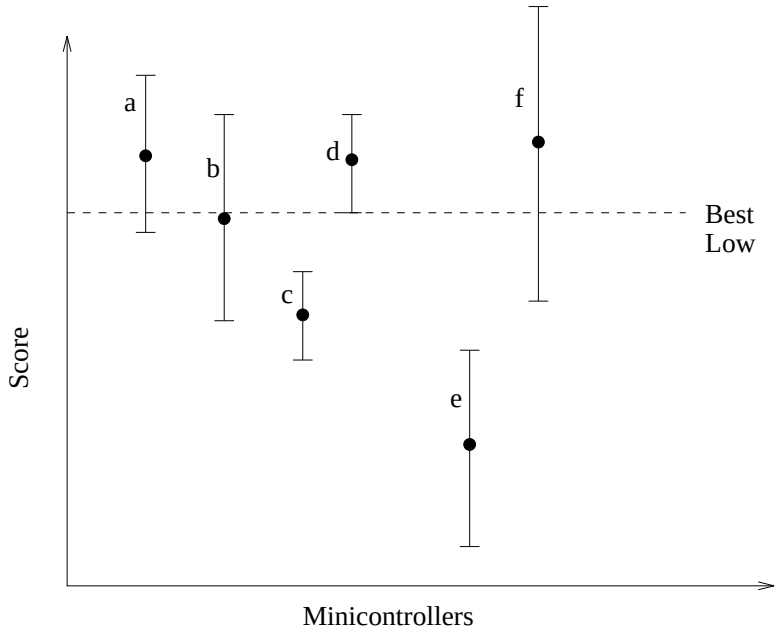

Figure 6-3: Example Hoeffding Race

The algorithm is then as follows: find the maximum lower endpoint over all the confidence intervals. In our example minicontroller d has the largest. We then eliminate any minicontroller whose top interval endpoint is less than this value. In our example, c and e would be eliminated from the race.

The normal Hoeffding Race optimization algorithm is shown in Figure 6-4. Rather than apply this directly, we make several modifications.

6.3.3 The Parallel Hoeffding Relay Race

Rather than start with a huge set of minicontrollers for a given partition and slowly eliminate them, we run a relay race. Specifically, we have a family of n controllers (sets of minicontrollers) that we race in parallel. If there are k minicontrollers per controller, then we are running k Hoeffding Races in parallel, each race with n runners (one for each controller).

Figure 6-5 lists pseudocode for the algorithm. We run each controller for one ball, keeping track of the statistics of each minicontroller for each controller. Then

1. Let $\mathcal{R}$ be a set of random variables $\{r_i\}$.
2. Repeat until $|\mathcal{R}| = 1|$ or some maximum number of times.
   (a) Sample every random variable in $\mathcal{R}$ and incrementally calculate its statistics and c confidence interval.
   (b) Let m be the maximum lower confidence level over all variables in $\mathcal{R}$.
   (c) Remove all r_i from R that have an upper confidence point less than m.
3. $\mathcal{R}$ contains the variable or variables with true means significantly better than the variables originally in $\mathcal{R}$.

Figure 6-4: The Hoeffding Race Algorithm

1. Let $\mathcal{C}$ be a set of n controllers $\{C_i\}$.
2. Repeat *gens* times:
   (a) Run each $C_i \in \mathcal{C}$ for one ball, updating each minicontroller's flip statistics as it goes.
   (b) For each partition p do:
      i. Calculate confidence intervals of the minicontroller in p for each C_i.
      ii. Find the maximum low confidence interval value over these minicontrollers. Call it l^*.
      iii. Randomize any minicontroller who has a mean and variance of 0 after 3 flips.
      iv. Randomize any minicontroller whose high confidence interval value is less then l^*.
   (c) Find the minicontroller for each partition with the best mean.
   (d) Collect these into a single controller. This is our guess at the best controller so far.

Figure 6-5: The Parallel Hoeffding Relay Race Algorithm for Pinball

we calculate the confidence intervals for all the minicontrollers. Going through the partitions one by one, we find any minicontrollers in that partition that have fallen behind and trade them off with a new minicontroller — in our case a new random one. We decide who falls behind not only with the confidence interval, but also with an added heuristic. We randomize any minicontroller whose mean and standard deviation after three balls is still zero — this controller is probably never going to get better, and we're wasting our time with it[1]. This randomizing is the relay. When one runner lags behind, someone else is relayed in that will hopefully perform better.

We run this race for some number of generations. After the race is over, the best controller is defined as the set of minicontrollers that are the best in their partition.

The fact that we keep adding new random minicontrollers keeps up from getting stuck on local maxima. If a new random minicontroller turns out to be better, soon it will eliminate the old local maximum for that partition. As soon as all the minicontrollers in a partition are similar, however, very little change will occur.

In order to speed it up, rather than start with a random initial population — which, as we mentioned earlier, usually cannot hit the ball at all — we start with a population of controllers that have managed to hit the ball and get a score with only one ball, as we did for the hill-climbing tournament.

Finally, we chose to use the 90% confidence interval. Using the Hoeffding formula actually produces a lower bound on the confidence, so our confidence could be higher. (see Appendix B) We are willing to trade off confidence for speed, since the 90% interval will not be as wide as the 99% interval, say, and minicontrollers will be eliminated faster.

6.3.4 Strengths and Weaknesses

Since we cannot just run a minicontroller for one flip, we run in terms of balls. Therefore, we are constrained to running the simulation and only those minicontrollers that are activated get themselves optimized. This is actually excellent, however, since

[1]It would probably get eliminated later by a minicontroller with a good mean, but we want to hurry it along. It is hard to not get a score, so we can assume that it is not even flipping.

we are not wasting our time optimizing minicontrollers that rarely have a chance to flip anyway. Those minicontroller partitions that get the most use will get the most optimization.

This leads directly to a weakness, however. A certain minicontroller may never be activated if another minicontroller *in the same controller as it is* is not activated. That is, perhaps the only way minicontroller M_i gets a chance to flip is when M_j flips the ball to a certain place. This could only be a problem because our launch always causes a certain minicontroller to be first. We discuss an improvement possible research direction in Chapter 7.

Another minor problem is that it is sometimes difficult to tell which minicontroller had the final responsibility, since we can change minicontrollers if the ball bounces on the way in, for instance. This is currently being improved.

A final strength is that since we run in terms of flips rather than balls, we acquire samples much more quickly than if we were using score per ball as a sample. Thus, the confidence intervals converge much more quickly than the per ball mean confidence intervals.

6.3.5 Results

Table 6.2 shows the mean for 30 balls for the sum of the best minicontrollers found after a certain number of generations. (Note that the number of partitions is equal to two flipper partitions times some number of angle partitions. e.g. 12 partitions is six angles partitions by two flipper partitions). The reason the number of generations are seemingly random is due to lack of sufficient computational resources — optimizations were run as long as possible. It is actually interesting to see results after different amounts of generations. It seems strange that the best controller was worse when run for more generations. We cannot really say much, however, since the seed controllers may have been better for one, and the difference is only the variance of the stochastic nature of the algorithm. We also run the best summed controller from the same trial at different generations. These results are in Table 6.3.

Most generations eliminated about twelve minicontrollers. It is interesting that

Partitions	Number of Generations	Popsize	Mean for 30 balls
12	34	20	3279
8	64	20	1059
8	16	20	3921
8	12	20	5583

Table 6.2: Hoeffding Relay Race Results

Generation	Mean for 30 balls
1	2112
10	1422
20	841
30	1361
40	1079
50	1462
64	1059

Table 6.3: Hoeffding Relay Race results at different generations of the same run. This had 8 partitions and a population size of 20.

most were eliminated because they had zero mean and variance after several balls. This happens often when one uses a totally random minicontroller. Only rarely did the confidence interval method eliminate one — about one every one or two generations. Perhaps this is because it had not run long enough for the confidence intervals to become small enough to eliminate other controllers. Indeed, the number eliminated by the Hoeffding method increases with the number of generations, as we'd expect, since n gets larger for the minicontrollers.

An Optimal Solution In one of the trials I was pleasantly surprised to find that one of the controllers had randomly stumble upon the true optimal solution — a cycle. The cycle was as follows:

- Launch the ball as usual, so it went down the right passageway and at the left flipper.
- Don't flip at it; instead, let it bounce off and toward the right flipper.

- Flip early so that it rolled up the right return lane.
- Let it roll back down the right flipper until a certain precise point[2], then flip it up the left passageway, receiving a score for going through the passage.
- The ball then went over the caret symbol on the top of the board and back down the right passageway, beginning the cycle again.

Unfortunately, since it did not end, I had no way of knowing what the exact controller was. I added a feature to let me get the currently playing controller whenever I wished, but it has not rediscovered the controller yet — yet another example of closing the barn door after the cows are loose.

[2]This is really difficult for a human to do repeatedly. That is why pinball machines make money.

Chapter 7

Conclusions and Future Research

...as two grains of wheat, hid in two bushels of chaff: you shall seek all day ere you find them; and, when you have them, they are not worth the search.

— Shakespeare, *The Merchant of Venice*

7.1 Conclusions

Although we did not manage to reach the human benchmark, we did fairly well. The controllers appear to make intelligent flips often, sometimes catching it and then flipping it on the roll. It sometimes makes up for this good behavior by missing a ball that it could hit, however. Perhaps if we let the search run longer, the minicontrollers that miss these shots could become better and the score (as well as the appearance of intelligence) would be higher. If the poor minicontrollers are minicontrollers not commonly activated, they may not have been properly optimized. It is interesting to observe someone watching the controller play; they will often attribute intelligence to some of its shots ("Whoah, nice shot!") , then say "Hey, how come it flipped the wrong flipper? How stupid."

Preliminary results have also indicated that the parallel optimization of minicontrollers may not work as well as we hoped. This could be due to the fact that

our independence of minicontrollers assumption does not hold. This could happen because a sum of the best minicontrollers does not take into account that one minicontroller could have been getting a good score, but then expecting the minicontroller that was most likely to follow it to deal correctly with it. In the actual controller it is a part of for the race, perhaps the next minicontroller has been sufficiently optimized, since our good minicontroller kept passing to it. But it may not have the best mean in that partition, and won't be selected for the summed best controller. Thus, our good minicontroller gets a good score and passes it to another minicontroller that always loses it. Our mean per ball will not be high in this case. Increasing the number of partitions could help this, so that finer tuning is possible. We also suggest future optimization research below which will try to optimize the passing from one minicontroller to another, rather than using our greedy approach.

Another related problem is that minicontrollers may get penalized for something that is not their fault. Consider the event that the ball is going straight down. One of two minicontroller has responsibility for this state (left or right flipper). If the ball is going straight down the middle, neither minicontroller has a chance to hit it, but is still penalized for losing it. Let us say that when the ball is going straight down at its flipper, it does quite well. Consider, then, that the minicontroller before it is always hitting the central triangle continually, and causing the ball to come down the center. A potentially good minicontroller will be eliminated because it never gets a chance to get a good score because the ball never comes straight down at the flipper. This implies a dependence on other minicontrollers. We could hopefully eliminate this problem by partitioning in the h dimension as well, so that different minicontrollers handle the ball going down the center and the ball going at a flipper.

A potential problem is that the ball is always launched to the same configuration — towards the left flipper. If we were to randomize the launch settings, the system would have a more robust set of experiences to learn from, rather than always the same point, since it would be starting in a random configuration all the time. This would discourage minicontrollers from specializing on "one-trick" shots that only work off the launch and getting stuck on local maxima.

Forcing the rolling state into a trajectory projection may not be a good idea, in retrospect, using the piecewise lienar function. In particular, there may be very different good actions for rolling and for the other states in that partition. For this reason, it seems like a good idea to have a separate minicontroller for rolling on each flipper.

The hill-climbing tournament did fairly well, showing a general improvement trend that levelled out quickly. If we save the best controller seen so far and ran it for more generations, eventually we should find better controllers. It did not fall prey to the dependence problem because it still took dependence into account — it looked only at the mean score per ball. Also, it was a more direct method, actually using the value of the reward function to optimize rather than using a different function that we thought would imply the real reward function (i.e. a greedy optimization of each minicontroller). We mentioned already that it might be a good idea to add a temperature value for choosing random minicontrollers.

Both optimization methods are fairly slow. Perhaps there are other constraints or heuristics we can make to improve the search time. If we are to hook this up to a real machine, we would like it to learn a pretty good action function in no longer than a day, say. People seem to learn a lot more quickly, and do not let the ball just roll off the flipper like some of the random controllers. Perhaps adding some amount of extra knowledge, such as "Don't let it just roll off the flipper" could improve the system. Another idea to avoid this problem is to flip the flipper closest to the ball when $d_{critical}$ is reached. This would probably improve the appearance of intelligence quite a bit, as well as speed up the searches since it reduces the search space size. This constrains the generality of the algorithm, however. It is unclear if we want ever want to flip the other flipper instead to pass the ball or something.

We might also look at constraining the gain value interval more tightly, thus reducing our search space size. This could be difficult, since we do not want to make it too small. It is not clear how to do this well.

Different numbers of partitions did not seem to affect the results significantly for the range of 8 to 14 partitions. Perhaps by allowing the actual partition sizes

to change, the number of partitions would have more effect. For the relay method, increasing the number of partitions has no real change in speed. The tournament method slows down considerably, though.

7.2 Future Research Directions

It is interesting that Moore and Atkeson got away with quantizing a continuous control problem with limited actuation (keeping an inverted pendulum upright on a wagon), pretending that it was a stochastic Markov process, then using dynamic programming to find a good control policy[Moore and Atkeson, 1992]. The reason this is interesting, as he mentions, is that there is *hidden state* in the representation. That is, a given quantized chunk of state space actually covers a range of continuous values that are hidden from the controller. My intuition is that the smoothness of the state space of this problem allowed this type of method to work well. Pinball seems much more complex and less smooth than this problem, so I would expect that a direct uniform quantization probably would not work well because it would hide too much state.

This leads to the idea that we used in this thesis — using linear controllers in a variable partitioning of state space. Indeed, this is a simpler version of Moore's current research [Moore and Atkeson, 1992], using local linear optimal controllers in variable partitions of state-space. Applying this kind of idea to our problem, we identify each of the minicontrollers as a state in a Markov process. Then we can calculate transition probabilities for one minicontroller to send the ball to another minicontroller using dynamic programming techniques. This will also allow us to find a near-optimal strategy, given that we know what score each minicontroller normally gets. If there is a cycle in the Markov process, i.e. a way to keep the ball going forever, we expect to find it with this method. By using these probabilities, we implicitly assume that the minicontrollers are not independent as we did in this research.

Other future research could include allowing the size of the angle partitions to change, as well the number of partitions. It is true that we really do not know which

areas of the function can be approximated by a linear controller. Indeed, many of the problems listed above were problems with the partitioning. This should make our methods work much better. The drawback is that this could take much more time. Possibly there are ways to speed up these methods.

Finally, we mention that some sort of hybrid of the two methods could work well. In particular, once a set of pretty good minicontrollers have been found, hill-climbing by changing the parameters slightly might produce even better solutions.

Appendix A

Incremental Sample Statistics

The sample mean for n samples x_i is defined as

$$m_n = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{A.1}$$

and the sample variance as

$$\sigma_n^2 = \frac{1}{n} \sum_{i=1}^{n} (x_i - m_n)^2 \tag{A.2}$$

We want to find a recurrence relation for these formulae so that we need not store all the points to calculate the sample variance by direct application of equation A.2. In other words, we want a formula for m_n which only uses m_{n-1} and the new sample point x_n. We also want one for σ_{n-1}^2 in terms of σ_{n-1}^2, m_n, m_{n-1} and x_n. First we do the sample mean, which is simple.

We split the summation:

$$m_n = \frac{1}{n} \sum_{i=1}^{n-1} x_i + \frac{x_n}{n} \tag{A.3}$$

Then noticing the sum is just $n-1$ times m_{n-1}:

$$m_n = \frac{1}{n} \left((n-1)\, m_{n-1} \right) + \frac{x_n}{n} \tag{A.4}$$

and simplifying:

$$m_n = \left(\frac{n-1}{n}\right) m_{n-1} + \frac{x_n}{n} \tag{A.5}$$

We get the relation as desired. We initalize $m_0 = 0$.

Next we wish to find a formula for σ_n^2. This is a little harder. We expect to save some small extra state. Expanding equation A.2 produces:

$$\sigma_n^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i^2 - 2\, x_i\, m_n + {m_n}^2) \tag{A.6}$$

Distributing the summation:

$$\sigma_n^2 = \frac{1}{n-1} \left(\sum_{i=1}^{n} x_i^2 \right) - \frac{1}{n-1} \left(2\, m_n \sum_{i=1}^{n} x_i \right) + \frac{n\, {m_n}^2}{n-1} \tag{A.7}$$

The last term we can clearly calculate directly incrementally using equation A.5. This leaves the first term (call it α_n) and the second term (call it β_n). Clearly α_n is close to the mean sum of squares of the points. We can store this as state and update it incrementally. This becomes obvious when we rewrite it as

$$\alpha_n = \frac{1}{n-1} \sum_{i=1}^{n-1} x_i^2 + \frac{x_n^2}{n-1} \tag{A.8}$$

and noticing that α_{n-1} is only

$$\alpha_{n-1} = \frac{1}{n-2} \sum_{i=1}^{n-1} x_i^2 \tag{A.9}$$

we can substitute $(n-2)\alpha_{n-1}$ in for the sum in the equation for α_n to give us

$$\alpha_n = \frac{1}{n-1} \left((n-2)\alpha_{n-1} \right) + \frac{x_n^2}{n-1} \tag{A.10}$$

and simplifying produces the relation

$$\alpha_n = \frac{(n-2)\alpha_{n-1} + x_n^2}{n-1} \tag{A.11}$$

However, it is not valid for all n, specifically $n = 1$ and $n = 2$. If we specify boundary cases of

$$\alpha_2 = x_1^2 + x_2^2 \tag{A.12}$$

Then the relation will work for all $n > 2$. Notice that we cannot define the sample variance for $n = 1$ since the denominator is zero.

Next we find a relation for β_n.

$$\beta_n = \frac{2\, m_n}{n-1} \sum_{i=1}^{n} x_i \tag{A.13}$$

Splitting the summation yields

$$\beta_n = \frac{2\, m_n \left(\sum_{i=1}^{n-1} x_i\right)}{n-1} + \frac{2\, m_n\, x_n}{n-1} \tag{A.14}$$

The term for m_{n-1} glares at us and we can write

$$\beta_n = 2\, m_n\, m_{n-1} + \frac{2\, m_n\, x_n}{n-1} \tag{A.15}$$

which is the relation desired.

The win here is that we only need to store m_{n-1} and α_{n-1} in order to calculate σ_n^2, since

$$\sigma_n^2 = \alpha_n - 2\, m_n\, m_{n-1} + \frac{2\, m_n\, x_n}{n-1} + \frac{n\, {m_n}^2}{n-1} \tag{A.16}$$

for $n > 2$ since α_n and m_n can be calculated incrementally first.

The standard way to do this is to just store the sum of points and sum of squares of the points and calculate the mean and variance more easily. However, the sum of squares could potentially get quite large, and we'd have roundoff errors and possible numerical overflow. By storing the mean sum and mean sum of squares, we constrain the state from getting too large.

Appendix B

Confidence Intervals Using the Hoeffding Inequality

Calculation of confidence intervals involves using the following equation from probability:

$$\Pr\{\,|\mu - m_n| \leq t\,\} = c \tag{B.1}$$

where c is called the *confidence*. The *confidence interval* that we care about is the interval symmetric around the current sample mean, $[m_n - t, m_n + t]$. Equation B.1 states that the probability that the real mean μ is within t of either side of the current sample mean m_n is c. In other words, we are confident with probability c that the real mean is within $[m_n - t, m_n + t]$.

The classic way of finding these intervals involves using the central limit theorem, which can be found in any introductory statistics text. It states the normalized sum of independent identically-distributed (i.i.d.) random variables approaches the normal distribution, commonly known as the Bell Curve or Gaussian. Stated explicitly:

$$\Pr\left\{\left|\frac{S_n - n\mu}{\sigma\sqrt{n}}\right| \leq t\right\} = \frac{1}{\sqrt{2\pi}}\int_{-t}^{t} e^{\frac{-x^2}{2}}\,dx \tag{B.2}$$

where S_n is the sum of n i.i.d random variables. There are two problems with this formula for our purposes. First, be need to know the *a priori* true mean μ and

standard deviation σ of the random variables, which we do not; indeed, this is what we are searching for. Instead, we use the following formula from [Hoeffding, 1963]:

$$\Pr\{m_n - \mu \geq t\} \leq e^{\frac{-2nt^2}{B^2}} \tag{B.3}$$

where B is the maximum range of the random variable. Clearly as $B \to \infty$, then the right hand side becomes unity, a trivial upper bound, since probabilities are by definition bounded above by unity. So we must be able to bound the variable values in order to gain anything from this formula. We talk about choosing B in our context later.

But Equation B.3 is not in the form of Equation B.1. First, we notice that:

$$Pr\{|m_n - \mu| \geq t\} = Pr\{m_n - \mu \geq t\} + Pr\{-m_n + \mu \geq t\} \tag{B.4}$$

Note that the Hoeffding upper bound still applies to each summand in this sum. This in turn produces

$$Pr\{|m_n - \mu| \geq t\} = 2\,Pr\{m_n - \mu \geq t\} \leq 2\,e^{\frac{-2nt^2}{B^2}} \tag{B.5}$$

since the probabilities of the summands are equal. This is because the probability distribution of the sample mean around the real mean is symmetric. Still, we are not in the proper form, since this equation is an upper bound on the probability that real mean is *outside* of the interval of length $2t$ around the sample mean. We notice that:

$$Pr\{|m_n - \mu| \geq t\} = 1 - Pr\{|m_n - \mu| < t\} \tag{B.6}$$

so that

$$Pr\{|m_n - \mu| < t\} \geq 1 - 2\,e^{\frac{-2nt^2}{B^2}} = c \tag{B.7}$$

This is a lower bound on the confidence interval. It says that the true mean will be within the symmetric interval of length $2t$ around the sample mean greater than or equal to some probability c. Given a c, it is simple to calculate the interval by

solving for t:

$$t = \sqrt{\frac{B^2}{-2n} \ln \frac{1-c}{2}} \tag{B.8}$$

For our purposes, we can afford to be a little sloppy. For this reason, even though a score is potentially infinite, we choose to bound B above by two standard deviations above the mean — indeed the value should fall within this range 95% of the time. And, besides, if a score is infinite, we will never end up calculating the interval, since we never lose the ball!

Bibliography

[Foley *et al.*, 1990] James D. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics: Priciples and Practice.* Addison-Wesley Publishing Co., Reading, Massachusetts, 1990.

[Hoeffding, 1963] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 1963.

[Kernighan and Ritchie, 1988] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[Maron, 1993] Oded Maron. Finding good models fast using Hoeffding Races. In preparation, 1993.

[Moore and Atkeson, 1992] A. W. Moore and C. G. Atkeson. Memory-based Reinforcement Learning: Memory-based Reinforcement Learning: Converging with Less Data and Less Real Time. In preparation, 1992.

[Press *et al.*, 1988] William H. Press, Brian P. Flannery, Saul A. Teukolksy, and William T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, Cambridge, 1988.

[Rogalski, 1992] Grzegorz Rogalski. Robo-Pinball: Interfacing a Computer and a Vision System with a Pinball Machine. S.B. Thesis, Massachusetts Institute of Technology, May 1992.

[Schaal, 1993] Stefan Schaal. Mit artifical intelligence lab, 1993. Personal communication.