Action Selection for Hammer Shots in Curling: Optimization of Non-convex Continuous Actions With Stochastic Action Outcomes

by

Zaheen Farraz Ahmad

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Optimal decision making in the face of uncertainty is an active area of research in artificial intelligence. In this thesis, I present the sport of curling as a novel application domain for research in optimal decision making. I focus on one aspect of the sport, the hammer shot, the last shot taken before a score is given, and how selecting this shot can be modelled as a low-dimensional optimization problem with a continuous action space and stochastic transitions. I explore the unique research challenges that are brought forth when optimizing in a setting where there is uncertainty in the action outcomes. I then survey several existing optimization strategies and describe a new optimization algorithm called Delaunay Sampling, adapted from a method based on Delaunay triangulation. I compare the performance of Delaunay Sampling with the other algorithms using our curling physics simulator and show that it outperforms these other algorithms. I also show that, with a few caveats, Delaunay Sampling exceeds the performance of Olympic-level humans when selecting strategies for hammer shots.

To my family

because otherwise my mom would never let me hear the end of it.

Acknowledgements

The utmost of gratitudes go to my supervisor Robert C. Holte for his mentorship and guidance throughout the span of my degree. I will always be thankful to him for providing me with the freedom to explore and to learn while also giving me direction when I needed it.

I would also like to thank Michael Bowling for all the advice he gave me and for his work as a collaborator and on my committee. I would also like express my gratitude to Mark Lewis for being a member of my committee and for his insight.

Much of the work done in this thesis is part of a larger project and I would like to thank the other members of the project, Timothy Yee and Viliam Lisy. I would also like to particularly thank Marlos Machado. He has been an invaluable friend in the last few years, both academically and personally. I have had many other friends that have been there for me throughout my Master's program, to whom I am grateful, but I would like to notably thank Sheehan Khan and Richard Valenzano.

Finally, I would like to thank my parents. They have continually been supportive of me and they never let me quit when I was down. I could never have done this without them.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Problem Definition

Sequential decision making is an active area of research in artificial intelligence. Adversarial games are a common test bed for designing agents to solve decision making problems. In fact, success in developing artificial intelligence agents that can beat humans in checkers, chess and, more recently, poker and go were great milestones for research in the field. Curling is an Olympic ice sport that can be modelled as a two-player adversarial game. Unlike other games such as chess and poker, curling possesses a continuous state and action space. In addition, execution error in the actions provides the added difficulty of stochastic transitions. Using curling as a domain for AI research gives rise to a new problem: sequential decision making *in the face of action uncertainty* - a new twist to an old game, so to speak.

A game of curling is played between two teams. The game is played in a number of rounds (usually 8 or 10) called "ends". In each end, teams alternate sliding granite rocks down a sheet of ice towards a target. When each team has thrown 8 rocks, the score for that end is determined and added to the teams' scores from the previous ends. The rocks are then removed from the playing surface and the next end begins. The team with the highest score after the final end is the winner.

As a consequence of scoring each end separately and clearing the sheet after each end is finished, the strategy for any end can be thought of as independent from the strategy of the other ends. While it is true that during each end the objective should be to choose actions taking into consideration the long-term effects that increase the chances of winning the game overall, this can be achieved while keeping the strategies for the ends separate if the actions are evaluated in terms of the overall effect on the game. Therefore, choosing a strategy for each end can be viewed as a separate sequential decision making problem.

In most approaches to finding a solution for a problem of this nature, the problem is modelled as a *game tree*. The tree comprises all the possible sequences of decisions that can be made. Each *node* of the tree is a decision point where the branches from a node are the

possible actions that can be taken at a decision point. Each path from the root of the tree to the leaf is a possible sequence of decisions. The objective of AI agents for these decision making problems is to search for the best or *optimal* sequence in the tree. In games where the possible set of actions is discrete and small, it is possible, given current hardware and technology, to evaluate every possible sequence when searching for the optimal sequence. In games with a larger space of actions, it can become intractable to enumerate every possible sequence of actions. Agents for these settings rely on algorithms that learn to prune out suboptimal sequence to find the optimal sequence. However, typical algorithms applicable for games with discrete move sets cannot be applied to curling effectively.

Due to the continuous nature of the actions in curling, each decision point in the game tree for curling possesses an infinite set of possible actions. This is further compounded by the stochastic uncertainty of the actions. This results in a progressively widening tree with an infinite branching structure. Methods that rely on a small number of discrete actions cannot be applied directly to the continuous setting.

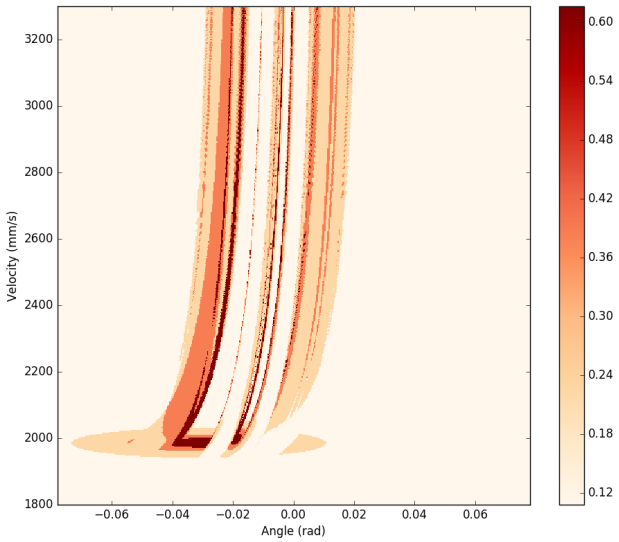


Figure 1.1: Noise-free heatmap of hammer shot outcomes.

To simplify this difficult problem, in this thesis we only look at the final decision point of the game tree for curling. This decision point reflects the last shot of an end called the hammer shot. After an action is taken for a hammer shot, a score is observed. The problem is thus reduced to a low-dimensional optimization problem. Although it is only the final decision to be made in an end, the hammer shot is of the utmost importance because it largely determines the outcome of the end. In fact, the strategy for either team in an end is mainly influenced by the hammer shot — the team possessing the hammer takes actions

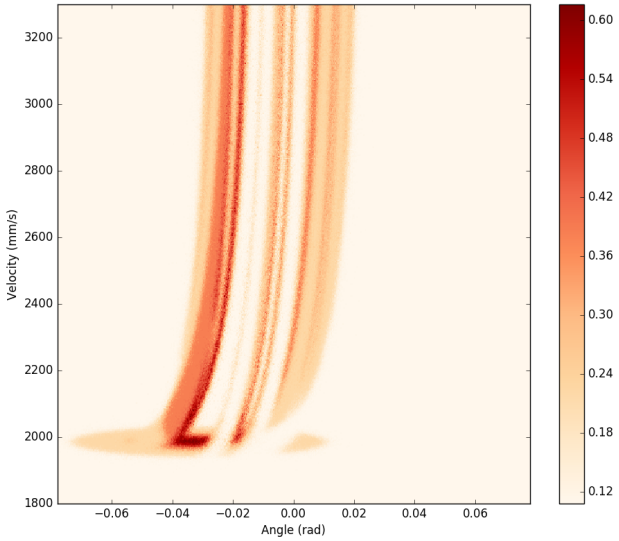


Figure 1.2: Noisy heatmap of hammer shot outcomes.

throughout the end to set up a good final shot situation while the other team takes actions to prevent it. This focus on the hammer shot removes the need to explicitly reason about the opponent, while still leaving the substantial challenge of efficiently identifying a near-optimal action in a continuous state and action space with stochastic action outcomes and a highly non-convex scoring function. This work is part of a larger research project that uses search methods to select all the shots in an end [Yee *et al.*, 2016] and, ultimately, to plan an entire game.

To illustrate the difficult nature of even this restricted optimization problem, Figure 1.1 shows a heatmap for a typical hammer shot. The shading represents the score — a darker shade is a higher score for the team throwing the hammer shot — as a function of the two main continuous action parameters, angle (θ, the x-axis) and velocity (v, the y-axis). This is a deterministic heatmap: it shows the exact score if shot (θ, v) is executed without error. Finding an optimal shot means finding the darkest parts of this heatmap. As can be seen they constitute a very small portion of the action space and are often surrounded by less desirable outcomes (light regions). Since action execution is stochastic, the expected value of shot (θ, v) is the average noise-free values of shots, (θ', v'), weighted by $p((\theta', v')|(\theta, v))$, the probability that (θ', v') is executed given that (θ, v) is the intended shot. This essentially blurs the deterministic heatmap (illustrated in Figure 1.2), making the darkest regions even smaller (only the central region of the rectangular region where $\theta \approx -0.035$ and $v \approx 2000$ is optimal in expected value in the heatmap shown in Figure 1.2).

1.2 Thesis Contributions

This research makes two main contributions. The first is to adapt Surovik and Scheeres [2015]'s non-convex optimization method to our problem. They use Delaunay triangulation on a set of sampled points to discretize the continuous action space and focus subsequent sampling in regions that appear promising. Our contribution is to add a final step, in which a shot is selected by treating the most promising regions as "arms" in a multi-armed bandit problem. We call our method Delaunay Sampling (DS).

The second contribution is to evaluate the effectiveness, for hammer shot selection, of DS and a representative set of existing algorithms for non-convex function optimization. For this we use a curling simulator and actual hammer shot situations from the 2010 Olympic Winter Games. DS is shown to be computationally cheaper while achieving superior results to these established algorithms. We also compare DS's average expected value on the Olympic hammer shot situations with the average outcome achieved by Olympic-level curling teams themselves, showing a statistically significant improvement over these human experts, with some caveats.

One particular insight gained from this research highlights the challenges that arise when trying to optimize decision making in a setting where the uncertainty lies in the execution of the action as opposed to uncertainty in the value observed by taking an action. Some optimization approaches that account for uncertainty only focus on situations where it is modelled as the latter. As a consequence, it is difficult to implement in our setting, optimization approaches that explicitly require the knowledge of the noise in the observations beforehand. The novel method that we describe in this research is one way to overcome this challenge and does so by not explicitly modelling the noise but by implicitly learning the variance in the action space of the function being optimized.

A final insight garnered from our experimental results is that humans sometimes have difficulty in recognizing optimal actions in decision making situations such as curling. The experiments show that humans, at times, incorrectly assess the expectations of stochastic actions and that AI techniques can choose strategies that are better than what humans would choose.

1.3 Outline

This thesis proceeds as follows. Chapter 2 provides the necessary background on function optimization and describes the knowledge of curling needed to understand the work in this thesis. It also briefly describes the curling simulator that we used to model and explore the problem.

Chapter 3 provides a description of the other approaches we examined in our investiga-

tion of the optimization problem. Chapter 4 gives a thorough discussion of the Delaunay Sampling algorithm that we adapted for the application of optimization of hammer shots in curling.

Chapter 5 describes the experiments that have been performed to study the performance of our adapted algorithm and to compare them with the performance of the existing algorithms. Chapter 6 compares the performance of our algorithm to human standards and explores several cases where we observed large differences between the performances of DS and humans.

Chapter 7 briefly goes through various works that are similar to our problem and other related works. Chapter 8 concludes this thesis, providing a summary of our findings and what we have learned and lists out possible future directions for this research.

Chapter 2

Background

This chapter covers the background knowledge essential to understanding the later parts of the thesis.

2.1 Optimization

In computing science and mathematics an *optimization* problem requires one to find the minimum or maximum solution to a function [Boyd and Vandenberghe, 2004]. The requirement for the minimum or maximum depends on the application of the optimization problem but for this thesis we will focus on finding the maximal solution.

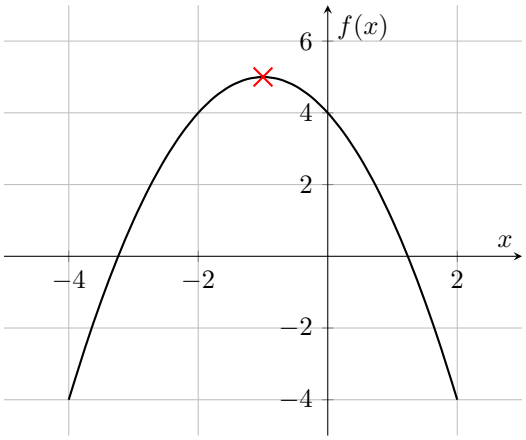


Figure 2.1: Example of a convex 1-dimensional function

Suppose there is a continuous function $f : \mathcal{X} \to \mathbb{R}$ defined over the domain $\mathcal{X} \subseteq \mathbb{R}^d$ where d is the *dimensionality* of the domain i.e., the number of variables of the function. Optimization of f seeks to find a point $\mathbf{x}^* \in \mathcal{X}$ such that for any $\mathbf{x} \in \mathcal{X}$, we have $f(\mathbf{x}^*) \geq f(\mathbf{x})$. For the *objective function* $f(\mathbf{x})$ being optimized, $\mathbf{x}$ is called the *optimization variable* and $\mathbf{x}^*$ is known as the *optimum solution*. Figure 2.1 illustrates a simple 1-dimensional function and the optimum point of this function, which is marked as a cross. In this figure, the optimum point of this function is at $x = -1$ with a solution of $f(x) = 5$.

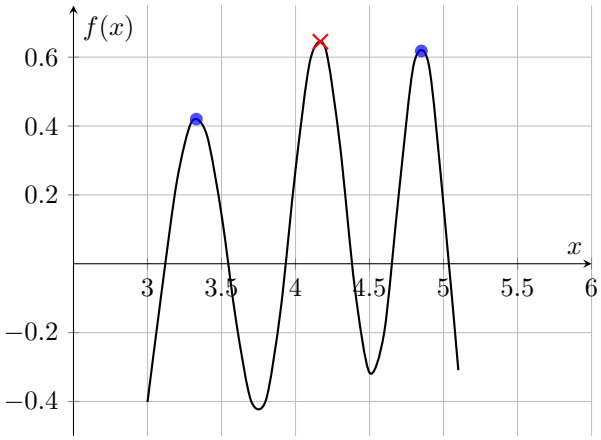


Figure 2.2: Example of a non-convex 1-dimensional function

In artificial intelligence, optimization is used in various applications such as parameter tuning for machine learning algorithms [Snoek *et al.*, 2012] and control of robot movements [Lizotte *et al.*, 2007]. The importance lays in modelling a certain problem as an optimization problem and choosing the right approach to solving it. The approach used in solving a specific optimization problem is mainly determined by the properties of that optimization problem.

2.1.1 Convex Optimization

A convex optimization problem is one that maximizes a function f where $f : \mathcal{X} \to \mathbb{R}$ is a convex continuous function. That is, the function satisfies the constraint

$$\forall \mathbf{x}_1, \mathbf{x}_2 \in \mathcal{X},\ 0 \leq \alpha \leq 1 : f(\alpha \mathbf{x}_1 + [1-\alpha]\mathbf{x}_2) \geq \alpha f(\mathbf{x}_1) + [1-\alpha] f(\mathbf{x}_2) \tag{2.1}$$

The function plotted in Figure 2.1 is a convex function — a straight line between any two points of the function lies entirely below the function. Convex optimization problems can be quickly solved using linear programming (depending on the function), gradient descent methods or quasi-Newton methods [Boyd and Vandenberghe, 2004]. However, not all problems can be easily transformed into a convex optimization problem and we must rely on other methods to solve them.

2.1.2 Non-convex Optimization

When an unknown function is possibly not convex, finding the optimum point of such a function is known as *non-convex optimization*. Although there are several methods for non-convex optimization, unfortunately, there are no reliably effective algorithms to tractably solve such problems [Boyd and Vandenberghe, 2004].

Local Optimization

In non-convex functions, there may exist several points that are *local optima*. A locally optimum point maximizes the objective function with respect to the neighborhood of points near it. That is, a point $\mathbf{x}$ is a local optimum of a function if

$$\exists \delta > 0 \; \forall \mathbf{x}' \in \mathcal{X}, \; ||\mathbf{x} - \mathbf{x}'|| \leq \delta \implies f(\mathbf{x}) \geq f(\mathbf{x}'). \tag{2.2}$$

However, it is not guaranteed to be the maximum point over the entire domain of the function. Local optimization techniques seek to find a local optimum at the cost of finding the optimum point $\mathbf{x}^*$. These techniques are normally used if a local solution to an optimization problem is "good enough" with respect to the requirements threshold. Figure 2.2 shows an example of a non-convex function with three local optima and 1 global optimum point. The point marked with a cross is the global optimum. The points marked with dots are local optima.

Global Optimization

Global optimization, in contrast to local optimization, seeks the true *global optimum* value of the function at the cost of greater computation times. Typically, global optimization techniques are used in cases where the number of variables to be optimized are small as solving problems with larger dimensions may be intractable. However, global optimization techniques may be the best approach to non-convex optimization if the benefit of knowing the true global optimum outweighs the cost of computation time.

2.2 Curling

Curling is an Olympic ice-sport that is popular in several countries, especially in Canada. A game of curling is played between two teams of players and typically lasts 10 rounds called *ends*. At the conclusion of an end, points are tallied. The team with the highest cumulative points after all ends have been played is declared the winner.

2.2.1 Ends

Figure 2.3: Diagram of sheet (not to scale.)

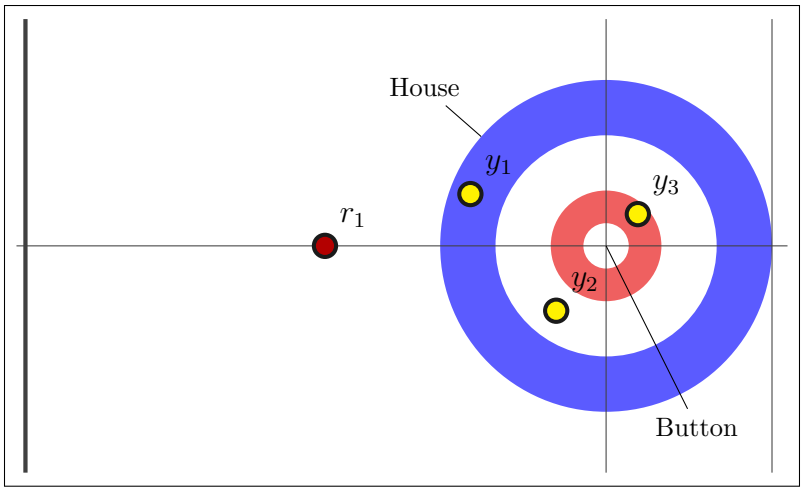


Figure 2.4: Diagram of the house with 4 rocks in play.

An end starts on an empty sheet of ice as shown in Figure 2.3. In each end, teams take turns sliding granite stones called *rocks* from one end of the sheet toward the other aiming at a bullseye-like target called the *house*. Each team gets eight *shots* during an end and after a total of 16 shots have been thrown, the end is over. Points are scored and the sheet is cleared to begin the new end.

Figure 2.4 shows an example of a close-up view of the house after a few shots have been thrown. In the diagram, the set of concentric circles depict the house. The pin at the center of the house is called the *button*. The circles labelled y_1 through y_3 are the rocks thrown by the Yellow team and the circle labelled r_1 is the rock thrown by the Red team.

2.2.2 Scoring

The score a team receives at the conclusion of an end is based on the configuration of the rocks in the house after shots have been played. The team that receives points is the team with a rock in the house that is closest to the button. The value of this score is equal to the number of rocks the team has in the house closer to the button than the closest rock of the opposing team. For example, let us imagine Figure 2.4 illustrates the state of the ice after all shots have been played. Even though 16 rocks have been thrown, only 4 remain in play. If the scores were tallied with this configuration then the yellow team would be awarded 3 points. The yellow team has a rock closest to the button (y_3) and they have a total of 3 rocks (y_1, y_2, y_3) in the house. The red team has no rock in the house and has no effect on the yellow team's score. However, suppose instead we have the configuration depicted in Figure 2.5 where rock y_2 is replaced with a rock r_2 so that the red team now has a rock in the house. In this case, the yellow team would receive only 1 point since they have only one rock closer to the button than the red team. In fact, any rock the yellow team where its

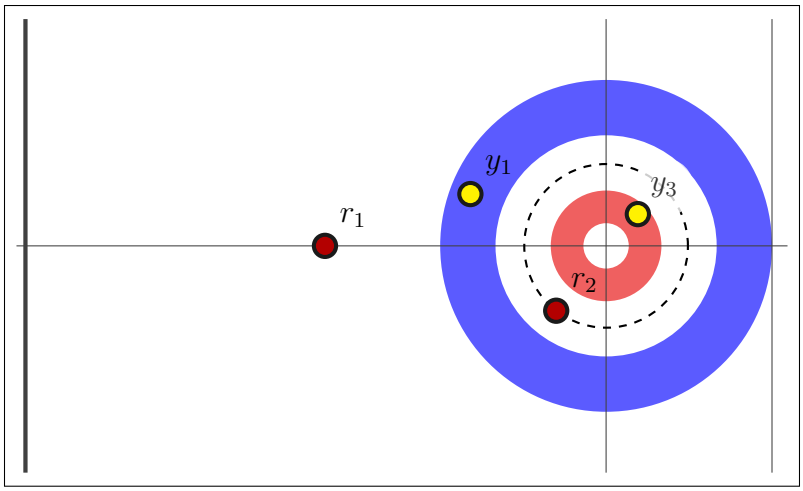


Figure 2.5: State of the ice where Yellow team scores 1.

center is on or outside the dotted circle drawn would not contribute to their score.

2.2.3 Shots

A shot is played from the far end of the ice sheet (to the left of the Figure 2.3). The shooting player pushes out with the rock in hand and releases the rock with an initial linear and angular velocity at some angle relative to the center line. The angular velocity causes the rock to travel in an arc ("curl"), deviating from the straight line path that the rock was initially on. The direction of the curl (toward the bottom or top of the figure) is determined by the sign of the angular velocity (clockwise or counterclockwise, respectively), but is little affected by the magnitude of the angular velocity. Figure 2.6 shows an example of what the path of a rock looks like. Let us suppose the shooting player desires to slide a rock to the button. One possible way to achieve this shot is to release the rock with a counterclockwise turn on the trajectory depicted as the straight line in Figure 2.6. This would result in the rock actually travelling with the path depicted as the dashed line in the figure.

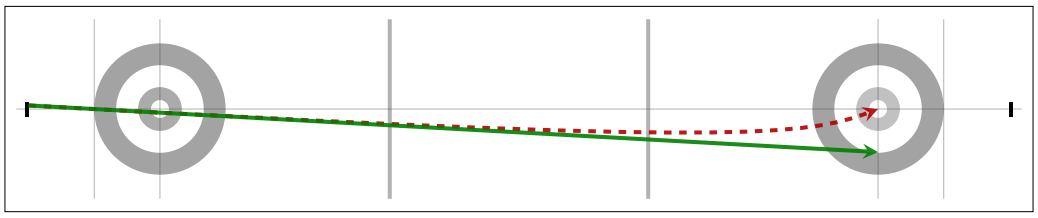


Figure 2.6: Path of rock showing curl.

Two of the team's players accompany the rock as it travels down the ice, and one or both may choose to sweep the rock at any time. Sweeping affects the deceleration of the rock (it travels further if swept) and the amount it curls (it curls less if swept.)

The intended outcome of a shot, though, is not always realized. There are two main reasons.

- **Human error.** The player throwing the stone might not perfectly deliver it at the required angle or velocity. Additionally, the skip may incorrectly judge the rock's path or the sweepers its speed, resulting in sweeping being misapplied to achieve the desired outcome.
- **Variability in the ice and rocks.** Although care is taken to make the ice conditions identical along all paths, there are differences, and the ice conditions can change as a game goes on. Similarly, every rock is slightly different in how it interacts with the surface of the ice.

2.2.4 The Hammer Shot

The last shot of an end, called the *hammer shot*, is of the utmost importance as it heavily influences the score for the end. In fact, it is so important that the team that begins the game in possession of the hammer shot has a 55.7% probability of winning the game (how this value is calculated is discussed in § 2.4.) This thesis focusses exclusively on the problem of selecting the hammer shot.

2.3 Modelling the Hammer Shot

The *state* for a hammer shot is determined by the score differential, the number of ends left to play, and the (x, y) positions of the rocks in play. Our action space is based on two simplifying assumptions. First, we treat the angular velocity of the hammer shot as a binary variable (clockwise or counterclockwise). Second, we do not have any parameters related to sweeping in our action space. Instead we integrate the effects of sweeping into the execution model of our curling simulator (see below). Our action space therefore has two continuous dimensions and one binary dimension ("turn").

The result of an intended shot is determined by two main components: a physics based simulation and an execution model. Surprisingly, the physics of a curling stone is not fully understood and is an active area of research [Nyberg *et al.*, 2013; Nyberg *et al.*, 2012; Jensen and Shegelski, 2004; Denny, 1998; Lozowski *et al.*, 2015]. So, a simulation based on first principles is not possible. The curling simulator used in this thesis is implemented using the Chipmunk 2D rigid body physics library with an artificial lateral force that visually recreates empirically observed stone trajectories and modified collision resolution to visually match empirically observed elasticity and energy conservation when rocks collide. A rock's trajectory is modelled by a deterministic simulation given an initial linear velocity, angle, and turn.

The execution model in our curling simulator represents the variability in outcomes in the execution of an intended shot. The execution model treats this variability as a stochastic transformation. If (θ, v) is the intended shot, the outcome is the deterministic simulation of a perturbed shot (θ', v') sampled from a predetermined conditional probability distribution whose mode is the intended shot. The primary purpose of sweeping is to correct for human error and variability in the ice conditions.[1] This is implicitly incorporated into the execution model as a reduction in the execution noise, i.e., an increase in the likelihood the rock's actual trajectory is close to the planned trajectory. At present, we do not model ice or rock variability: we assume all shots are subject to the same execution error. The execution model used in the experiments in this thesis come from perturbing the intended shot parameters with independent samples from a heavy-tailed, zero-mean, Student-t distribution whose parameters have been tuned to match Olympic-level human ability.

2.4 Objective Function

What objective function do we wish to optimize? The answer that usually springs to mind is points, i.e. find a shot with the maximum expected point (EP) difference. To see why this is not ideal, consider choosing the very last shot of a game in which the team with the hammer is losing by two points. Suppose shot A is 100% guaranteed to score one point and shot B has a 20% chance of scoring 3 points and an 80% chance of giving up one point. Shot A has a much higher EP than B (1.0 compared to -0.2) but it has no hope of winning the game, whereas B will win 20% of the time. B is obviously the better choice in this situation. For this reason, we focus on optimizing win percentage (WP), not EP.

WP is a function of two game state variables: n, the number of ends left to play, and δ the number of points by which the team with the hammer at the start of the end is leading (δ is negative if the team with the hammer is losing). We call a pair (n, δ) the resulting game state, or g. For example, if $g = (1, -2)$, $\mathrm{WP}(g)$ is the probability of the team with hammer winning if at the start of the final end they are down by two points. We have then estimated the WP function from data using 28,000 curling games played between 2011 and 2013.[2] For the final end of a game, $g = (1, \delta)$, we estimated $\mathrm{WP}(g)$ using simple frequency statistics for the final ends from the dataset. For the second to last end of a game we used the same

[1] Sweeping does have effects beyond reducing execution error. Sweeping the rock near the the end of its trajectory can allow it to reach a location on the ice not possible without sweeping. Furthermore, a shot far from an intended shot can be swept to achieve an entirely different purpose, such as rolling under a different guard if the executed angle is off. The primary effect of sweeping, though, is to compensate for execution error.

[2] Forfeits were treated as transitions to win/loss states. For rare states, the outcomes used in estimating the transition probabilities came from states with a similar score differential (when near the end of the game) or similar number of ends remaining (when far from the end of the game). The data used came from http://curlingzone.com, and included both women's and men's tournaments, although almost no difference was observed when restricting to data only from one gender or when including only championship level events.

data to estimate the transition probabilities from $g = (2, \delta)$ to a game state $g' = (1, \delta')$ for the final end. This tells how frequently it happened that the hammer team having a lead of δ when there were two ends to play was followed by the hammer team in the final end having a lead of δ'. With these transition probabilities and the already-computed values of $\mathrm{WP}((1, \delta'))$ for all δ', it is easy to compute $\mathrm{WP}((2, \delta))$ for all δ. The same process can then be applied to compute $\mathrm{WP}((3, \delta))$, $\mathrm{WP}((4, \delta))$, etc. With $\mathrm{WP}(n, \delta)$ defined for all values of n and δ, the "score" we return when a hammer shot with x ends remaining results in a lead of δ for the team with the hammer in the next end is $\mathrm{WP}(x - 1, \delta)$. This is the objective function our methods aim to maximize in expectation in selecting the hammer shot.

Chapter 3

Algorithms

The following description of the related algorithms will refer to terms and definitions previously described in the Background chapter.

3.1 Hierarchical Optimistic Optimization

3.1.1 The Continuous-Bandit Problem

In the general bandit problem, an agent is presented with a set of arms. Each time the agent pulls an arm (the action), it receives a reward. The reward received from an arm is drawn from a probability distribution that is associated with that arm. The objective of that agent is to select arms to optimize its cumulative reward. Upper Confidence Bounds (UCB) [Auer *et al.*, 2002] is an arm selection policy that associates each arm with a UCB-value that depends on the entire history of sequence of pulls using the following equation:

$$v_i = \bar{r}_i + C\sqrt{\frac{\log N}{n_i}}. \tag{3.1}$$

$\bar{r}_i$ is the average reward observed by the arm, i; N is the total number of iterations; and n_i is the number of times arm i was played. The second term is an estimate of the size of the one-sided confidence interval for the average reward where the expected value can be found with strong probability. At each iteration, the arm with the highest UCB-value is chosen to be played. The second term ensures that arms that have not been played in some time will be played. C is a scaling factor which influences the proportion of iterations spent exploring these actions.

The continuous action optimization setting can be treated as a continuous bandit problem [Bubeck *et al.*, 2009b; Kleinberg, 2004]. The problem is represented by a set of arms denoted by $\mathcal{X} \subseteq \mathbb{R}^d$, where $\mathcal{X}$ is the bounded environment space and d is the number of action parameters over which to optimize. At each round, the agent pulls one arm $x \in \mathcal{X}$. After each pull, the agent observes the reward obtained by taking the action at the arm.

3.1.2 HOO

The HOO algorithm approximates the reward function over the environment space $\mathcal{X} \subseteq \mathbb{R}^d$ by expanding a binary cover tree of the space on each iteration (see Figures 3.1 - 3.3.) Each node (h, i), of the tree covers some subspace $\mathcal{P}_{h,i}$ of the environment and provides an estimate of the maximum expected reward observed by sampling in $\mathcal{P}_{h,i}$ where h is the depth of the node and $1 \leq i \leq 2^h$. The children $(h+1, 2i-1)$ and $(h+1, 2i)$ of node (h, i) each cover a subspace of the parent's such that $\mathcal{P}_{h,i} = \mathcal{P}_{h+1,2i-1} \cup \mathcal{P}_{h+1,2i}$ and $\mathcal{X} = \bigcup_{i}^{2h} \mathcal{P}_{h,i}$. Since the nodes at greater depths span smaller subspaces, they provide a better estimate of the average rewards that are observed by sampling in their corresponding subspace.

Sampling using HOO is done by taking the action at the center of the subspace covered by a leaf node on each iteration. The node to be "played" is selected using B-values associated with each node; the B-value is an *optimistic* (best case) estimate of the maximum expected value observed at the node. Starting at the root, we traverse the tree following nodes with the highest B-values (ties are broken by uniform random selection.) When a leaf node is reached, a sample is taken in the space of that node, and then the B-values of all the nodes in the path are updated using the value of the reward received. Calculating $B_{(h,i)}$ of any node after N iterations depends on its U-value, an initial estimate of the maximum expected reward in the region covered by the node. The U- and B-values are calculated as follows:

$$U_{(h,i)} = \begin{cases} \hat{\mu}_{(h,i)} + \sqrt{\dfrac{2 \ln N}{n_{(h,i)}}} + \nu_1 \rho^h, & \text{if } n_{(h,i)} > 0; \quad (3.2a) \\ +\infty, & \text{otherwise.} \quad (3.2b) \end{cases}$$

$$B_{(h,i)} = \begin{cases} \min\left\{U_{(h,i)}, \max\left\{B_{(h+1,2i-1)}, B_{(h+1,2i)}\right\}\right\}, & \text{if } n_{(h,i)} > 0; \quad (3.3a) \\ +\infty, & \text{otherwise.} \quad (3.3b) \end{cases}$$

In Equation (3.2a), $\hat{\mu}_{(h,i)}$ is the average of rewards received by (h, i) and all its descendants after N rounds and $n_{(h,i)}$ is the number of times (h, i) or one of its descendants has been played after N rounds. The second term takes into account the uncertainty on the average of the rewards of the function for a node. $\nu_1 \rho^h$ is information on the shape or smoothness of the function in the partition covered by the node. The B-value provides a tight, optimistic, high-probability upper bound on the maximum expected reward of the space $\mathcal{P}_{h,i}$ covered by a node (h, i). Furthermore, the children $(h+1, 2i-1)$ and $(h+1, 2i)$ of the node provide valid upper bound values for regions $\mathcal{P}_{h+1,2i-1}$ and $\mathcal{P}_{h+1,2i}$ respectively. So $\max\left\{B_{(h+1,2i-1)}, B_{(h+1,2i)}\right\}$ is a valid upper bound for $\mathcal{P}_{h,i}$. $U_{(h,i)}$ is also another upper bound estimate for region $\mathcal{P}_{h,i}$. In Equation (3.3a), we choose the minimum of these upper bound estimates as the B-value to place a tighter upper bound on the interval in which we have a high probability of finding the expected reward.

In the initial step of HOO, the values of B and U of the root node are calculated by sampling at the center point of the the entire space.

Figure 3.1 - Figure 3.3 runs through a 1-dimensional example of how HOO works.

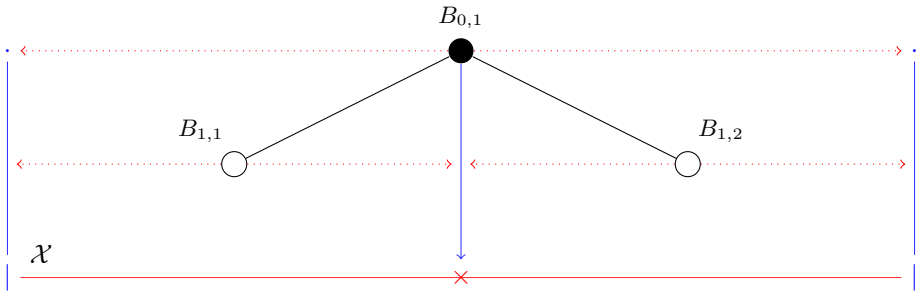


Figure 3.1: Iteration 1 of HOO

After the first iteration (shown in Figure 3.1), the root of the cover tree created by HOO provides an estimate of the maximum expected reward that can be observed in the space $\mathcal{X}$ and has a B-value, $B_{0,1}$. The dark cross on the bottom is the point in $\mathcal{X}$ where the sample was taken on this iteration. The root has two child nodes both having B-values, $B_{1,1} = B_{1,2} = \infty$.

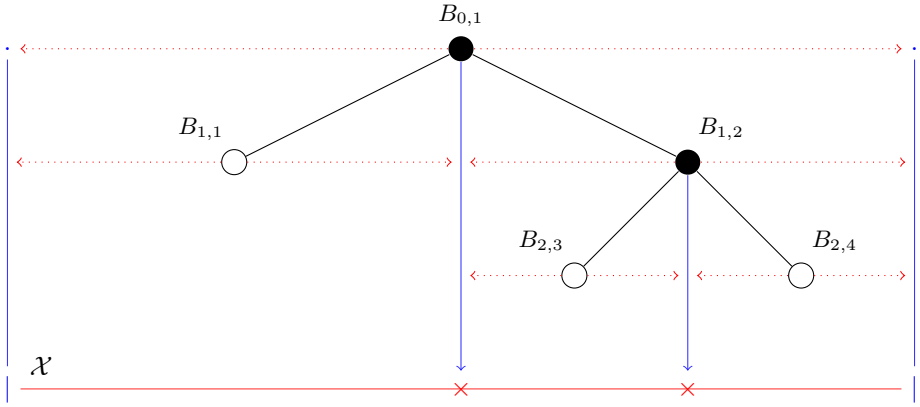


Figure 3.2: Iteration 2 of HOO

During the second iteration (Figure 3.2), the right child of the root was randomly selected to be played (each child had an equal probability of being chosen.) The crosses again show the points that have thus far been sampled. Now $B_{1,2}$ is a value other than ∞ calculated using the reward observed by sampling the arm in the space at the second dark cross and the value of $B_{0,1}$ is updated.

In iteration 3 (Figure 3.3), the left child of the root was played since now $B_{1,1} > B_{1,2}$. In the next iteration, traversal will go down whichever node between $(1, 1)$ and $(1, 2)$ would have a higher B-value and one of their children would be randomly selected to be played.

In our application, we are not concerned with finding the best sequence of actions to take

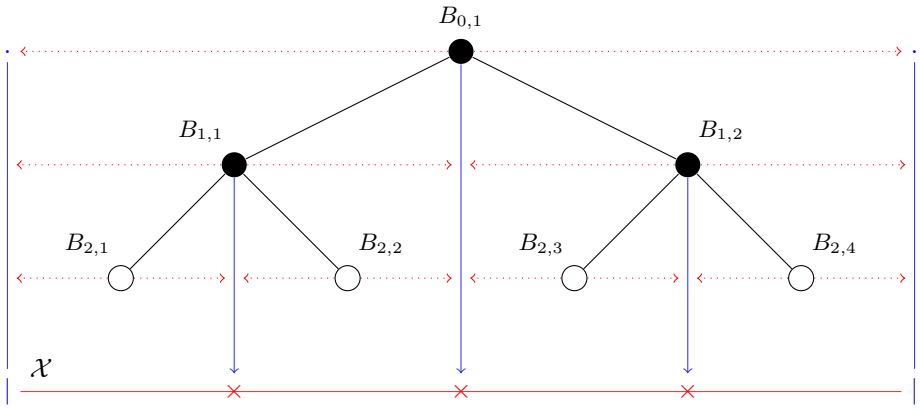


Figure 3.3: Iteration 3 of HOO

to maximize our cumulative reward. We are instead looking for the single best action to take given a set of possible actions. HOO, in our case, provides a high-probability estimate for the expected rewards of promising actions. We use a selection policy after a set number of iterations of HOO to choose the best among these actions. In our implementation, to select the best action to take, we traverse the tree following nodes with the highest average reward, $\hat{u}_{(h,i)}$ (as opposed to the highest B-values). The center point of the leaf that is reached at the end is the action to take.

3.2 Kernel Regression UCT

Monte Carlo tree search (MCTS) is a simulation-based search method to planning in sequential decision-making scenarios [Browne *et al.*, 2012; Kocsis and Szepesvári, 2006]. Every possible sequence of decisions in these scenarios can be represented as a path in a tree where each node of the tree is a decision point of the sequence. The branches from a node represent the actions taken at each decision point of the scenario. MCTS methods try find an approximately optimum sequence of actions without having to exhaustively search every possible path by eliminating decision paths with "bad" actions from the search.

Kernel Regression UCT (KR-UCT) is a variant of MCTS designed to be used in situations where stochasticity exists in the outcomes of the continuous actions of a search problem [Yee *et al.*, 2016]. The concept behind the approach is the use of *kernel regression* to provide a sense of information sharing between sampled points such that the value of a single point is estimated using the value of all other points in the environment space. This allows for information sharing between all sampled points.

3.2.1 Kernel Regression

Kernel regression is a technique for estimating the conditional expectation of a random variable given a set of data [Nadaraya, 1964; Watson, 1964]. Given a set of data points, one can estimate the expected value of any single point as the weighted average of the values of all points where the weighting is calculated by some non-linear distance function K called the *kernel*. Given a set of N points $\mathbf{X} = \langle \mathbf{x}_1, \ldots, \mathbf{x}_N \rangle$ and their observations $\mathbf{y} = \langle y_1, \ldots, y_N \rangle$, we can calculate the expected value of any point $\mathbf{x}$ in the set by:

$$\mathbb{E}(y|\mathbf{x}) = \frac{\sum_{i=0}^{n} K(\mathbf{x}, \mathbf{x}_i) y_i}{\sum_{i=0}^{n} K(\mathbf{x}, \mathbf{x}_i)}. \tag{3.4}$$

In the equation, the value of $\sum_{i=0}^{n} K(\mathbf{x}, \mathbf{x}_i)$ is known as the kernel density. This value, denoted as $W(\mathbf{x})$, is a measure of the amount of information available in the dataset about any particular point. The kernel function used is typically a smooth symmetric function e.g., a Gaussian probability distribution function. In our application, the kernel function is the probability distribution function of the execution error of the actions.

3.2.2 KR-UCT

In the work published by Yee et al. [2016], the authors used kernel regression to improve on the standard UCT algorithm [Kocsis and Szepesvári, 2006]. UCT is a bandit-based MCTS method where the actions at each node selected for the search are chosen using UCB (§ 3.1.1). In KR-UCT, the authors modify the UCB formula used for selecting an action and allow the candidate set of actions at each node of the tree to be extended by the addition of new actions.

Each node of the search tree has some action space $\mathcal{X}$ and each available action $\mathbf{x}$ is initialized to have an action value of infinity. As the algorithm progresses, the action values are updated according to a formula which is a combination of Equations (3.1) and 3.4. The value v of any action $\mathbf{x}$ in the action space of the node is given as

$$v = \mathbb{E}(y|\mathbf{x}) + C\sqrt{\frac{\log \sum_{\mathbf{x}' \in \mathcal{X}} W(\mathbf{x}')}{W(\mathbf{x})}} \tag{3.5}$$

where

$$\mathbb{E}(y|\mathbf{x}) = \frac{\sum_{\mathbf{x}' \in \mathcal{X}} K(\mathbf{x}, \mathbf{x}') \bar{y}' n(\mathbf{x}')}{\sum_{\mathbf{x}' \in \mathcal{X}} K(\mathbf{x}, \mathbf{x}') n(\mathbf{x}')} \tag{3.6}$$

$$W(\mathbf{x}) = \sum_{\mathbf{x}' \in \mathcal{X}} K(\mathbf{x}, \mathbf{x}') n(\mathbf{x}'). \tag{3.7}$$

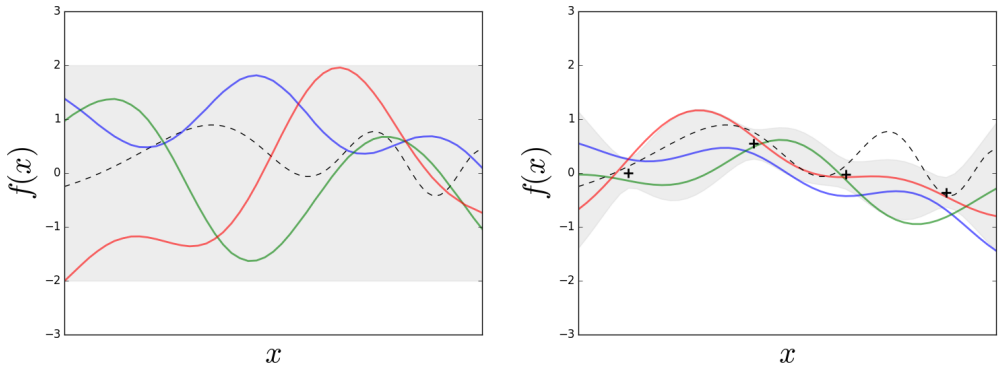
The first term in Equation (3.5) is the average reward of the action of a node. The second term is the confidence value of the action. C is the exploration constant similar to the original formula. For each node, $n(\mathbf{x}')$ is the number of times node $\mathbf{x}'$ is selected and $\bar{y}'$ is the average reward observed by $\mathbf{x}'$.

At each iteration of the algorithm, the tree is traversed by visiting nodes along the path with the highest v value. Each time a node is visited, its values of v and $n(\cdot)$ are updated. If a node is reached whose children have not all been visited, the node is expanded by adding one of the unvisited children as a new leaf to the tree. If a *terminal state* or final decision point is reached, then a new outcome is always added to the node. The mechanism by which selection and expansion of the tree occurs is detailed in the paper by Yee et al. [2016]. However, these details are not pertinent to our specific problem. Instead, we only focus on the final decision point of the tree.

In an optimization setting, KR-UCT is provided with an initial set $\mathbf{X} = \langle \mathbf{x}_1, \dots \mathbf{x}_N \rangle$ of N samples as candidate actions in the action space $\mathcal{X} \subseteq \mathbb{R}^d$ where $\mathbf{x} \in \mathcal{X}$. In our application, we used a uniform grid of samples initially. At each iteration of the algorithm, the action $\mathbf{x} \in \mathbf{X}$ from the candidate set with the maximum action value v from Equation (3.4) is selected to be played. If a new action is observed, it is added to the set of candidate actions. Then the action values of all candidate actions are updated. The modified UCB formula of KR-UCT ensures a balance between exploration and exploitation. High expected values can contribute to higher action values, ensuring that promising actions get selected and any new outcome improves the estimate of the expected value of the action. Lower information densities can also contribute to higher action values, so that actions that have not been explored as much, are played increasing the chances of finding better new actions. After the final iteration of KR-UCT, the action with the greatest lower confidence bound (the expected value *minus* the confidence value) is chosen as the optimal action to take.

3.3 Gaussian Process Optimization

Gaussian Process Optimization (GPO) [Lizotte *et al.*, 2007; Snoek *et al.*, 2012] falls under the umbrella of Bayesian Optimization techniques, a family of black-box, global optimization methods. The general idea of the Bayesian strategy is to treat the unknown objective function as a random function and set a prior for it. Function evaluations are used as data points to develop posteriors from the prior and find the next points to be sampled. The posterior distribution on the unknown function f is also a Gaussian process. GPO seeks to find the optimal point of an unknown function f through carefully choosing new points $\mathbf{x}_{n+1}$ to sample a value based on the posterior belief from previous samples. The optimal selection mechanism for a non-convex sampling horizon and prior is intractable. However, good performance can often be had by instead choosing a point that maximizes



(a) Three sets of predictions for the values of the objective function drawn from a GP prior. The dashed line is the objective function being approximated. The shaded region is the pointwise confidence interval within which we can expect to find the true value.

(b) Three sets of predictions of the objective function drawn from a GP posterior. The confidence intervals in this case are much smaller than with the prior. As a result, the chances of generating an accurate prediction for the true function values is greater.

Figure 3.4: Predictions of $f(x)$ drawn from a GP prior (a) and posterior (b).

some acquisition function as a proxy objective, e.g., choosing the point with the maximum probability of improving on the largest previously attained value.

3.3.1 Gaussian Processes

A *Gaussian process* is a distribution of functions over some continuous domain $\mathcal{X} \subseteq \mathbb{R}^d$. Any finite set of points in that space are random variables that exhibit a joint, multivariate Gaussian distribution [Rasmussen, 2006]. It is fully defined by a mean function, $m(\mathbf{x})$ and a covariance function, $k(\mathbf{x}, \mathbf{x}')$. A Gaussian process of a function $f(\mathbf{x})$ can be written as follows:

$$f(\mathbf{x}) \sim \mathcal{N}(m(\mathbf{x}), k(\mathbf{x}, \mathbf{x}')) \tag{3.8}$$

Let us assume we are given z number of points $\hat{\mathbf{X}} = \langle \hat{\mathbf{x}}_1, \ldots, \hat{\mathbf{x}}_z \rangle$ over some space and we wish to predict their values, $\hat{\mathbf{y}} = \langle \hat{y}_1, \ldots, \hat{y}_z \rangle$, from a function, $f(\mathbf{x})$, where $\hat{y}_i = f(\hat{\mathbf{x}}_i)$. We can create a Gaussian process prior with a mean vector, $\mathbf{m} = \mathbf{0}$, of length z (other mean functions exist, but a zero mean vector is commonly used) and covariance matrix, $\mathbf{K}(\hat{\mathbf{X}}, \hat{\mathbf{X}})$, calculated by the covariance function:

$$\hat{\mathbf{y}} \sim \mathcal{N}(\mathbf{0}, \mathbf{K}(\hat{\mathbf{X}}, \hat{\mathbf{X}})) \tag{3.9}$$

We can then generate values of $\hat{\mathbf{y}}$ for $\hat{\mathbf{X}}$ by drawing from this prior. Figure 3.4(a) illustrates three sets of predictions for $\hat{\mathbf{y}}$ by generating values of $\hat{y}$ along the entire function space using the prior. The true function is depicted as the dashed line. The shaded region in the

diagram represents the pointwise, 95% confidence interval from which a value for a given $\hat{\mathbf{x}}$ is drawn.

Naturally, predictions drawn from the prior are not likely to be representative of the true function values. Instead, we can compute a posterior distribution using values sampled from the actual function to produce a more accurate model. Suppose we have a set of already observed points $\mathbf{X}$ and their evaluations on the function, $\mathbf{y}$. The joint probability prior distribution of these observed points and the prediction points is given in Equation (3.10):

$$\begin{bmatrix} \mathbf{y} \\ \hat{\mathbf{y}} \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} [\mathbf{K}(\mathbf{X},\mathbf{X}) + \sigma^2\mathbf{I}] & \mathbf{K}(\mathbf{X},\hat{\mathbf{X}}) \\ \mathbf{K}(\hat{\mathbf{X}},\mathbf{X}) & \mathbf{K}(\hat{\mathbf{X}},\hat{\mathbf{X}}) \end{bmatrix}\right) \tag{3.10}$$

In realistic models, function values usually have noise associated with them (due to measuring errors, etc.) Therefore, the function values observed are given as $y = f(\mathbf{x}) + \epsilon$, where ϵ is Gaussian noise with variance σ^2. This is reflected in the computed distributions by adding the term, $\sigma^2\mathbf{I}$, to the covariance matrix of the observed points. Then, using Bayes' Rule, we can then compute the posterior distribution to get (3.11):

$$\begin{aligned} \hat{\mathbf{y}} \sim \mathcal{N}(&\mathbf{K}(\hat{\mathbf{X}},\mathbf{X})[\mathbf{K}(\mathbf{X},\mathbf{X}) + \sigma^2\mathbf{I}]^{-1}\mathbf{y}, \\ &\mathbf{K}(\hat{\mathbf{X}},\hat{\mathbf{X}} - \mathbf{K}(\hat{\mathbf{X}},\mathbf{X})[\mathbf{K}(\mathbf{X},\mathbf{X}) + \sigma^2\mathbf{I}]^{-1}\mathbf{K}(\mathbf{X},\hat{\mathbf{X}})) \end{aligned} \tag{3.11}$$

The posterior is also a Gaussian process defined by a *predictive mean* and *predictive variance.* Values drawn from this posterior provide a more accurate probabilistic model of the objective function than can be had using just the Gaussian process prior. Figure 3.4(b) shows three sets of probable function values over $\mathcal{X}$ made using the posterior over function values computed using four training points. It can be seen that the confidence intervals in this case are tighter than when predicting using just the prior.

3.3.2 Acquisition Functions

The Gaussian process posterior distribution computed in the previous section provides a probabilistic model for the unknown objective function $f(\mathbf{x})$. The accuracy of this model in representing the true function relies on the information provided by the set $\{\mathbf{x}_i\}_{i=1}^N$ of evaluated points sampled so far and our choice of covariance kernel. For rapid optimization of the objective function it is desirable to sample the next point, $\mathbf{x}_{N+1}$, in the space such that we increase our chances of finding the maximum obtainable reward.

One approach to finding the next point to sample is by maximizing some *acquisition function.* Acquisition functions model the information gained about the objective function by sampling points in the space given all previously observed samples. By maximizing an acquisition function we seek to increase the amount of information gained with each sample to quickly and accurately estimate the objective function globally.

There exist many different acquisition functions that can be used for proxy optimization of the objective function. Each of these acquisition functions provide a different measure of information gained from sampling. For instance, one such acquisition function could be to maximize the probability of improving over the current best value with each new sample. Another acquisition function could instead be to maximize the expected improvement over the current best. In our experiments, we chose to maximize a function called GP-UCB, another commonly used acquisition function that measures the upper confidence bound on the reward observed at a given point. The function (GP-UCB) is shown below:

$$a_{\text{UCB}}(\mathbf{x}_{N+1}; \mathbf{X}) = \mu(\mathbf{x}_{N+1}; \mathbf{X}) + \kappa v(\mathbf{x}_{N+1}; \mathbf{X}) \tag{3.12}$$

where $\mu(\mathbf{x}_{N+1}; \mathbf{X})$ and $v(\mathbf{x}_{N+1}; \mathbf{X})$ are the predictive mean and variance respectively of a point x_{N+1} calculated from Equation (3.11) given the set $\mathbf{X}$ of N previously observed samples. Maximizing this function selects either points with high uncertainty or points with high expected rewards as subsequent samples. The value κ determines the proportion of iterations spent sampling at points with high uncertainty.

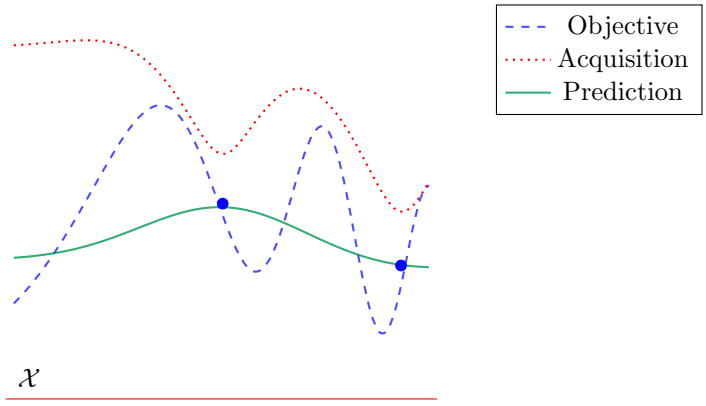


Figure 3.5: GPO Step 1.

Figure 3.5 through Figure 3.7 illustrate how Gaussian processes coupled with an acquisition function can be used to find the maximum point of the 1-dimensional, bounded function, $f(x) = \frac{1}{2}\sin x^2 + \frac{1}{2}\sin x + \frac{1}{8}x - \frac{1}{4} + \epsilon$ where $x \in [1.5, 4.0]$ and $\epsilon \sim \mathcal{N}(0, 0.1)$. We initialize the algorithm (Figure 3.5) by fitting a Gaussian process over two observations (the two dots) from the objective function (shown as the dashed line.) The number of initial points chosen depends upon the user; in our example we use only two to initialize to simplify the illustrations. The solid line shows the predictive mean about which the computed posterior is centered; predictions of the function values at any point in the space have an expected value that is given by the predictive mean. This predictive mean is then used to calculate the values of the acquisition function (dotted line), which is the function we

optimize to find the next sample point. In the first iteration, the next sample point chosen is to the left of the first two observation points, where the value of the acquisition function is the highest.

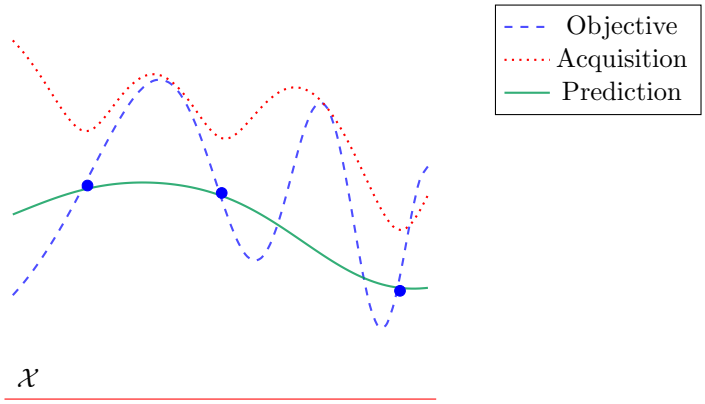


Figure 3.6: GPO Step 2.

Figure 3.6 depicts the second iteration of the Gaussian process optimization in our example. After evaluating, the point chosen to be sampled in this iteration is on the leftmost edge of the sampling space.

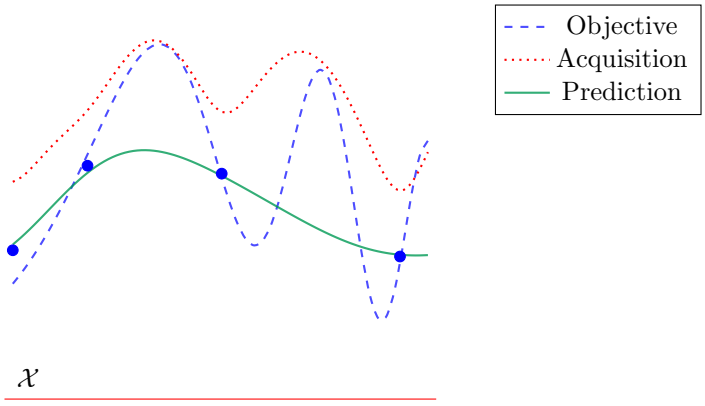


Figure 3.7: GPO Step 3.

The third iteration of GPO is shown in Figure 3.7. As we have more points with which we fit the Gaussian process, the prediction model more accurately represents the values of the objective around the optimal point. In this example, the next point chosen to be evaluated would have been the optimal point (or at least, very close to it.) Since the GP-UCB acquisition function was used, it is likely that in further iterations a few points further away will also be chosen to be evaluated as their confidence values increase over every iteration (depending on the value of κ, GP-UCB assigns higher values to points in the space where we have higher uncertainty, that is, in regions where we have not sampled

often.)

3.4 Covariance Matrix Adaptation - Evolution Strategy

Evolution strategies are iterative algorithms that attempt to optimize a function by introducing stochastic variations at each iteration [Bäck *et al.*, 1991]. CMA-ES [Hansen and Ostermeier, 1996; Hansen, 2016] proceeds by drawing a set of samples from a multivariate Gaussian distribution over an action space $\mathcal{X}$. The variables are initially assumed to be linearly independent. A new multivariate Gaussian is then constructed. The mean is the weighted mean of the sampled points, where higher weights are given to samples with larger reward values. The covariance matrix is modified from the previous covariance so as to encourage high variance in the direction that the mean is observed to be changing. This procedure is then repeated using the new multivariate Gaussian to sample points. The steps of the algorithm at each iteration are therefore

1. draw a set of samples using the multivariate Gaussian distribution,
2. shift the mean of the distribution,
3. update the covariance matrix of the distribution.

We will now look at each of these steps in more detail.

3.4.1 Sampling

At each iteration, the algorithm draws λ number of samples in the environment $\mathcal{X}$, which is a compact subset of $\mathbb{R}^d$. The reward for a sample, $\mathbf{x}_i$ is observed after being passed to the objective function $f : \mathcal{X} \to \mathbb{R}$, where $i = 1, \ldots, \lambda$. Each sample is drawn from a multivariate Gaussian distribution. The covariance matrix of the distribution is initially the $d \times d$ identity matrix.

3.4.2 Shifting the Mean

After each iteration, the mean of the Gaussian distribution is changed to increase the probability of selecting promising points. To do so, CMA first selects the top μ points with the best rewards among all samples. The mean is then augmented by adding to it the average difference of the μ sampled points from the mean. If $\mathbf{m}^{(n)}$ is the mean of the distribution at iteration n then the new mean is calculated as follows:

$$\mathbf{m}^{(n+1)} = \mathbf{m}^{(n)} + \sum_{i=1}^{\mu} w_i (\mathbf{x}_{i:\lambda}^{(n)} - \mathbf{m}^{(n)}), \qquad i = 1, \ldots, \lambda \tag{3.13}$$

where $\mathbf{x}_{i:\lambda}^{(n)}$ is the i-th best point of all λ samples at iteration, n and $\sum_{i=1}^{\mu} w_i = 1$. This centers the new mean of the distribution within the best μ points so that the current best

samples are more likely to be reproduced. The weight w_i in Equation (3.13) can be set so that the variances of all μ samples equally contribute to the update of the mean or so that the points with higher reward contribute more.

3.4.3 Updating the Covariance Matrix

After updating the mean of the distribution, the covariance matrix is updated to generate a new distribution for sampling. When updating the covariance matrix for the new distribution, the current covariance is augmented by two different components. After n iterations, the calculation of the next covariance can be done the following equation:

$$\mathbf{C}^{(n+1)} = (1 - c_\mu - c_1)\mathbf{C}^{(n)} + c_\mu \mathbf{C}_\mu + c_1 \mathbf{C}_1 \tag{3.14}$$

where $\mathbf{C}^{(n)}$, the first term, is the *current covariance*.

The second term is the *rank-μ update*. It is the covariance of the best μ sample points. Adding this term to the original covariance stretches the current distribution in the direction where there is high variance between the best μ points and the previous mean.

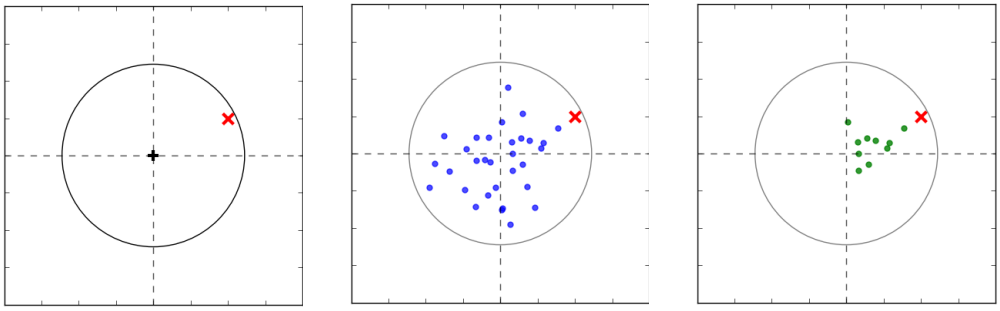
The third term is the *rank-1 update*. This is the covariance calculated from the sequence of steps taken by the mean, $\mathbf{m}$, of the distribution over the previous n iterations. This value scales the distribution in the direction of the cumulative path taken by the mean of the distribution. For instance, if the mean initially shifts from a point $\mathbf{m}^{(0)}$ to $\mathbf{m}^{(1)}$, then the distribution will be scaled in the direction of the vector, $\mathbf{p}^{(1)} = \mathbf{m}^{(1)} - \mathbf{m}^{(0)}$. In the next iteration, the distribution will be scaled in the direction of the vector, $\mathbf{p}^{(2)} = \mathbf{p}^{(1)} + \mathbf{m}^{(2)} - \mathbf{m}^{(1)}$.

The values of c_μ and c_1 are dependent on the number of parameters over which we are optimizing and a detailed description of how these are calculated can be found in the paper by Hansen et. al [2016].

Since with each iteration, the distribution is adapted such that the mean takes a step in the direction of the highest variance, CMA-ES can be viewed as a form of local gradient ascent. Over a sufficient number of iterations, the distribution will approach a local optimum point of the objective function and the probability of sampling high-valued points increases.

Figure 3.8 through Figure 3.10 will illustrate the first three iterations of CMA-ES in a 2-dimensional space, $\mathcal{X}$. In all diagrams, the $\times$ represents the location of the optimal point of the objective function, $f : \mathcal{X} \to \mathbb{R}$ and the ellipses depict the 95% confidence interval of the distribution determined by its covariance matrix.

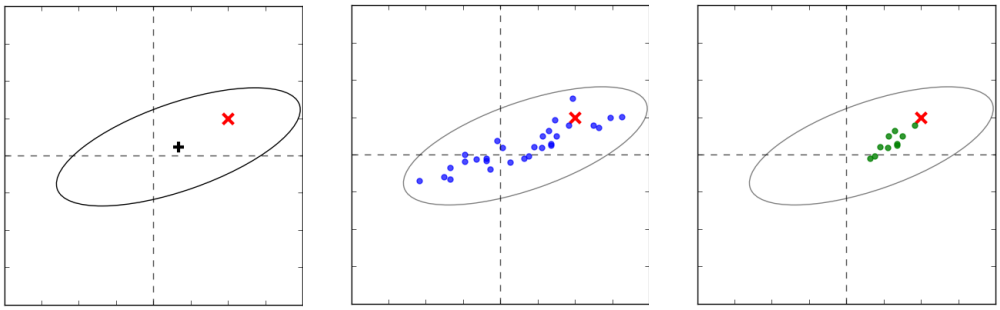
In the first iteration of CMA, shown in Figure 3.8, we have a distribution with mean, $\mathbf{m}^{(0)}$ at $(0, 0)$ and a 2×2 covariance matrix, $\mathbf{C}^{(0)} = \mathbf{I}$. The 95% confidence interval of this distribution is a circle because the parameters are initially assumed to be linearly independent of each other. Figure 3.8(b) shows all λ sample points drawn from $\mathcal{N}^{(0)}$.



(a) $\mathcal{N}(\mathbf{m}^{(0)}, \mathbf{C}^{(0)})$ (b) $\mathbf{x}^{(0)}_{1,\dots,\lambda}$ (c) $\mathbf{x}^{(0)}_{1,\dots,\mu:\lambda}$

Figure 3.8: CMA Step 1.

Figure 3.8(c) shows the μ sampled points that observed the highest reward. The new mean in the next iteration will be at the center of these points and the covariance will be updated such that the distribution is stretched in the direction of the variance of these points with respect to $\mathbf{m}^{(0)}$. The result is shown in Figure 3.9.

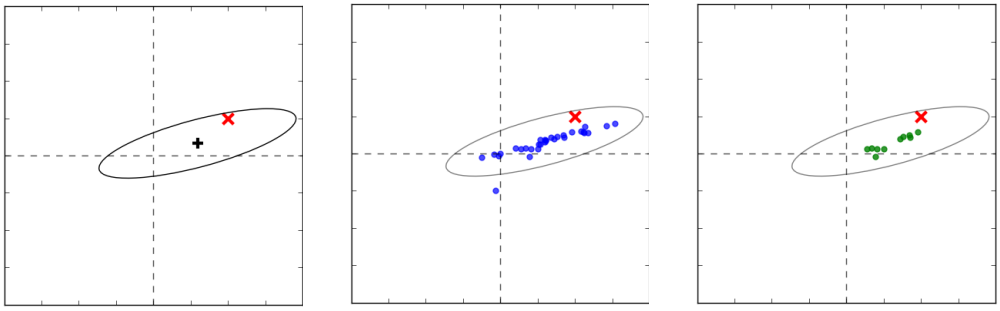


(a) $\mathcal{N}(\mathbf{m}^{(1)}, \mathbf{C}^{(1)})$ (b) $\mathbf{x}^{(1)}_{1,\dots,\lambda}$ (c) $\mathbf{x}^{(1)}_{1,\dots,\mu:\lambda}$

Figure 3.9: CMA Step 2.

Figure 3.9 illustrates the events of the second iteration of CMA. As seen in Figure 3.9(a), the ellipse shows how the distribution has changed since the last iteration. The distribution has stretched in the direction of the previous best points. Now most of the samples drawn (Figure 3.9(b)) are located in the region of the best samples from the last iteration. From here, the new set of the best μ points are selected.

Figure 3.10 illustrates the events of the third iteration of CMA. The distribution in this iteration is similar to that of the previous iteration with two main differences: the confidence interval is now smaller and the mean is centered closer to the optimal point. However, it still exhibits high variance in the same direction as before. This means that the λ samples in this iteration are selected from a smaller area and so the probability of selecting previous



(a) $\mathcal{N}(\mathbf{m}^{(2)}, \mathbf{C}^{(2)})$ (b) $\mathbf{x}^{(1)}_{1,\ldots,\lambda}$ (c) $\mathbf{x}^{(1)}_{1,\ldots,\mu:\lambda}$

Figure 3.10: CMA Step 3.

successful points has further increased. In the next iteration after this, the distribution should become even smaller, further concentrating samples in a promising region of the space.

3.5 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is a population-based, stochastic approach to optimization [Shi and Eberhart, 1998; Kennedy, 2011]. The algorithm relies on a set of particles, initially randomly placed in the environment space, that iteratively move towards local optima. At each time step of PSO, a particle, P, is at a location, $\mathbf{x}^{(P)}$, in the environment, $\mathcal{X} \subseteq \mathbb{R}^d$, of the objective function $f : \mathcal{X} \to \mathbb{R}$, where d is the number of action parameters to optimize. A particle observes a reward, $f(\mathbf{x}^{(P)})$, and the migration of all the particles depend on these rewards. In order to increase the likelihood of the particles successfully converging to a local optimum, each particle keeps track of both its own *personal best score* observed and the *global best score* observed by any particle along with its location in the environment.

At each iteration, n, the *velocity*, $\mathbf{v}_{n+1}$ with which a particle takes a *step* (a step is the addition of the velocity vector to a particle's current location in Euclidean space) is calculated as follows:

$$\mathbf{v}_{n+1} = w_n \mathbf{v}_n + c_1 \cdot r_1 \cdot [\mathbf{x}_p - \mathbf{x}_n] + c_2 \cdot r_2 \cdot [\mathbf{x}_g - \mathbf{x}_n], \qquad n = 0, 1, \ldots, N \tag{3.15}$$

where $\mathbf{v}_n$ is the particle's current velocity ($\mathbf{v}_0 = \mathbf{0}$), w_n is an iteratively decreasing weight on the previous velocity, and $\mathbf{x}_n, \mathbf{x}_p, \mathbf{x}_g$ are the particle's current location, the particle's personal best location and the global best location respectively. r_1 and r_2 are random values uniformly sampled from the closed interval $[0, 1]$. c_1 and c_2 are antagonistically

functioning weights which promote exploration and faster convergence respectively.

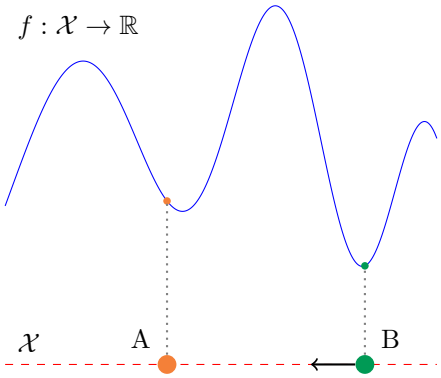


Figure 3.11: Step 1 of PSO with 2 particles where $f(x^{(A)}) > f(x^{(B)})$.

Figures 3.11 - 3.13 illustrate the first 3 steps of PSO to find the optimum point of the 1-dimensional, bounded function, $f(x) = \frac{1}{2}\sin x^2 + \frac{1}{2}\sin x + \frac{1}{8}x - \frac{1}{4}$ where $x \in [1.8, 4.2]$ using only two particles, A and B. Initially, A and B were randomly placed in $\mathcal{X}$ (shown in Figure 3.11). Here, $f(x^{(A)}) > f(x^{(B)})$. B in this case will step toward A, which will remain stationary (as per Equation (3.15).)

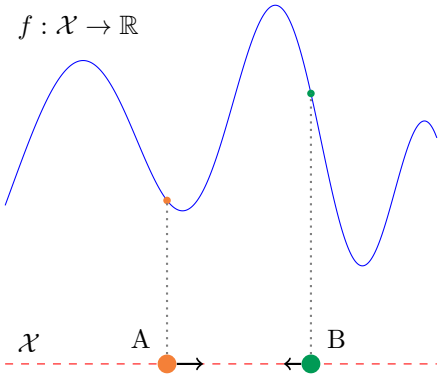


Figure 3.12: Step 2 of PSO with 2 particles where $f(x^{(B)}) > f(x^{(A)})$

In the second step of the algorithm (Figure 3.12) $f(x^{(B)}) > f(x^{(A)})$. Now, particle A will move toward B. The size of the step A takes is less than that of the B's in the first iteration since the particles are closer to each other in this iteration. Particle B will still move in the same direction as before but with less velocity only due to a diminished residual momentum from the last iteration (the first term of (3.15); the other two terms are now 0.) Both particles are however moving toward the maximum point of this function.

In the third step of PSO (Figure 3.13), still $f(x^{(B)}) > f(x^{(A)})$. Both particles will move closer still to the optimal point. As before, particle A steps toward B and particle B takes

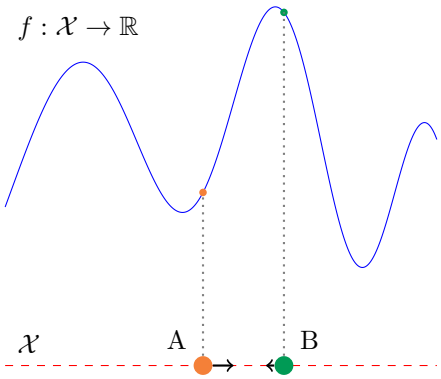


Figure 3.13: Step 3 of PSO with 2 particles where $f(x^{(B)}) > f(x^{(A)})$.

a step with the residual momentum.

In the subsequent iterations, both particles will have reached the optimal point. However, they will continue to oscillate about this point until their respective momentums reach 0. This decay of the momentum will proceed at a rate that depends on the value of w_n used in Equation (3.15). The higher this weight value, the longer their respective momentums last. Not shown in this example, are the effects of the personal best locations of the particles in other directions. If at any time in this case the particles had a personal best location near the other local optimal points, their steps toward the global best would have been decreased and the rate at which the particles converge would have decreased.

Chapter 4

Delaunay Sampling

Delaunay Sampling (DS) is a heuristic approach to function optimization that we have adapted from the method developed by Surovik and Scheeres [2015]. DS is an iterative optimization method that employs Delaunay triangulation to discretize a continuous space and focus sampling in promising regions. DS is divided into two phases: 1) Sampling using Delaunay triangulation and 2) selection of the optimal point of the function given the observed samples. The following describes the method of Delaunay triangulation and the property that makes it useful for function optimization. A description of the algorithm itself proceeds after. Algorithm 1 shows the full pseudocode for Delaunay Sampling.

4.1 Delaunay Triangulation

The following is a brief description of Delaunay triangulation with the information that is relevant for understanding DS. For more thorough details of triangulation, Delaunay triangulation and the claims made in the description, please consult the sources that are summarized in this section [De Berg *et al.*, 2000; Lee and Schachter, 1980; Shewchuk, 2002].

Triangulations are useful tools when working with point sets in a two-dimensional metric space. Triangulations divide a space into triangles based on the organization of the points. If a function f is defined on that space, triangulations can be used for interpolation of the function values in the space using sampling points of the functions as points for the triangulation. A triangulation of a finite point set $P \subset \mathbb{R}^2$ is collection $\mathcal{T}$ of triangles such that

1. $\bigcup_{T \in \mathcal{T}} T$ is a *convex hull*, the smallest convex set containing all the points in P
2. $P = \bigcup_{T \in \mathcal{T}} V(T)$ where $V(T)$ is the set of vertices/points of a triangle T
3. For every distinct pair $T, U \in \mathcal{T}$ the intersection $T \cap U$ is a common vertex, common edge or the empty set.

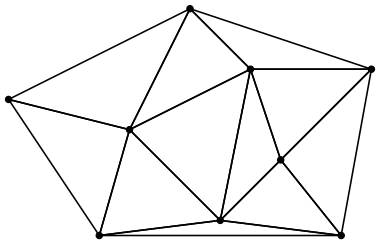
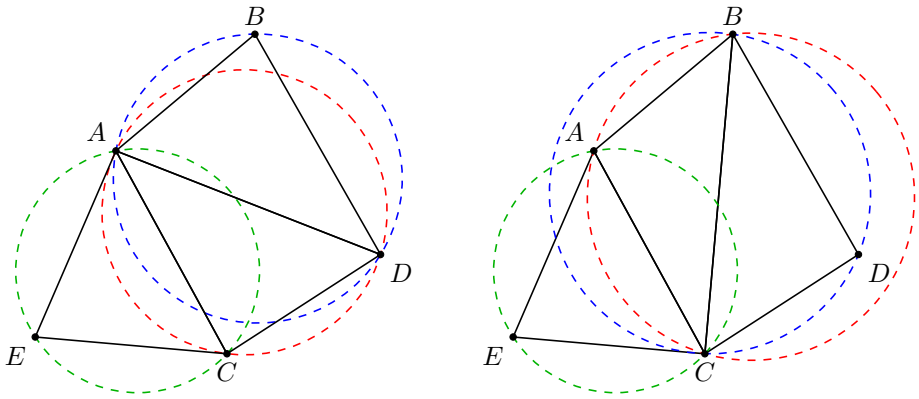


Figure 4.1: Point set triangulation.

Figure 4.1 illustrates an example of a triangulation of a point set, showing a convex hull subdivided with respect to all points in the point set.

A *Delaunay triangulation* is a triangulation of a finite point set $P \subset \mathbb{R}^2$ such that the points of P do not lie in the interiors of the circumcircles of the triangles [De Berg *et al.*, 2000; Lee and Schachter, 1980]. A circumcircle of a polygon (triangle in our case) is a circle that passes through every vertex of the polygon. Figure 4.2 shows two different possible triangulations of a point set with 5 points and the circumcircles of all the triangles. Figure 4.2(a) is a Delaunay triangulation since none of the points A through E are in the interior of any circumcircle. The triangulation in Figure 4.2(b) is not a Delaunay triangulation because point A lies in the circumcircle of $\triangle BCD$ and point D within that of $\triangle ABC$. There are several algorithms the can efficiently compute the Delaunay triangulation [De Berg *et al.*, 2000; Shewchuk, 2002; Lee and Schachter, 1980] of a point set.



(a) Delaunay triangulation (b) Non-Delaunay triangulation

Figure 4.2: Different triangulations using 5 points

By keeping all circumcircles empty of any points from the point set, Delaunay triangulation maximizes the minimum angle of all the triangles in the triangulation [De Berg *et al.*, 2000]. That is, Delaunay triangulation avoids the production of long, thin, *sliver triangles* which are undesirable for interpolation. Given point sets where the creation of sliver triangles are unavoidable, Delaunay triangulation will create fewer of these than other

triangulation methods.

As Delaunay triangulation tends to avoid producing sliver triangles, it is useful in applications such as topology mapping of a terrain. If a set of points are chosen in some terrain, we can create a Delaunay triangulation of that space. The points can then be raised by the height of the terrain at that point. Since there are no (or fewer) sliver triangles, there are no long spans of the space that are not discretized and resultantly, the mapping produced should be a more accurate mapping of the terrain. Similarly, Delaunay triangulation can be used to approximate the shape of a function during function optimization where the "height" of a point is the value of the function at that point. In higher dimensional spaces, the *simplices* produced by Delaunay triangulation are the higher dimensional counterparts of triangles (i.e., tetrahedrons when $d = 3$.)

4.2 Sampling with Delaunay Triangulation

In the first phase of Delaunay Sampling, the method described by Surovik and Scheeres [2015] is adapted in order to sample efficiently. This phase utilizes Delaunay triangulation to discretize the space given a set of samples such that we can identify promising regions for subsequent sampling.

4.2.1 Original Algorithm

The development of the original algorithm was motivated by the exploration of small celestial bodies using spacecrafts. The algorithm was designed to improve automated planning of trajectories of the non-Keplerian motion of spacecrafts during observational missions.

Automated control of spacecraft trajectories can be performed by planners choosing actions (choosing a trajectory) based on a mapping $\mathcal{M}$ of a 3-dimensional trajectory space $\triangle\mathcal{F}$ to a score space $\mathcal{S}$ where the planner takes the action that would achieve the best score. This mapping is done by an objective function that takes a trajectory and returns a score using the orbit dynamics of the spacecraft and the application-specific mission objectives. The algorithm developed by Surovik and Scheeres [2015] aims to chart this mapping using an efficient sampling-based approach.

The algorithm begins by taking a random set or uniform grid of sample trajectories $\triangle\mathbf{f} \in \triangle\mathcal{F}$ and calculating their corresponding scores $s \in \mathcal{S}$. A Delaunay triangulation mesh is then produced using this set of sample points in $\mathcal{M}$ where each simplex is bounded by 4 points. Each simplex i is then given a weight w_i. Next a set of k simplices is selected at random with replacement from the mesh. The probability with which a simplex is chosen is proportional to its weight. Each time a simplex i is selected, a point is sampled uniform randomly within it and added to the set of sampled points. The process of creating a new triangulation with the augmented set of samples and selecting subsequent samples is

repeated for a number of iterations. The weights that are given to each simplex is calculated as follows:

$$w_i = \text{volume}(i) \cdot mean(\text{lifespan}(i)) \cdot max(s + \hat{Q}_{max})^{2^{8\tau}} \tag{4.1}$$

where the first term is the volume of the simplex i and it acts as a normalizing factor. The second term is the mean lifespan of trajectories of simplex i and it directs the search towards regions that haven't been recently sampled. The final term is the gradient-augmented score where the value $\hat{Q}_{max}$ produces a gradient to represent how close a trajectory passed a high scoring sample. τ is a refinement factor that increases from 0 to 1 that causes the search to be progressively more biased towards regions with higher augmented scores.

4.2.2 Adaptation for Curling

Let us assume we are given a metric space $\mathcal{X} \subseteq \mathbb{R}^2$ over which an unknown stochastic function f is defined. Each point $\mathbf{x}_j$ in the point set $\mathbf{X}$ is an action associated with a reward $y_j = f(\mathbf{x}_j + \epsilon(\mathbf{x}_j))$. In our application, ϵ is a stochastic perturbation on the the action sampled from a student's t-distribution tuned to match Olympic-level human ability. Given a set $\mathbf{X}$ of N sample points, each point can be indexed by a value j where $j = 1, 2, \ldots, N$. Our objective is to find the action in the space that yields the highest expected reward given a *budget* of a finite number of samples.

Initially, a set of u points is chosen uniformly over the space (i.e., a set of 10×10 points in a 2-dimensional space.) Using this point set a Delaunay triangulation mesh is created and a set $\mathcal{T}$ of triangles is produced (initially the triangles are all congruent since the points are all spaced evenly.) Uniform points are chosen in the initialization to ensure that the convex hull formed from the point set covers the space $\mathcal{X}$ and no region is disregarded during optimization.

The iterative weighting, re-sampling and re-triangulation method is then performed just as it was in the original algorithm for n iterations. However, the weighting function used in DS differs. In DS the weight the triangles receive depends on the area a_T of the triangle and its score s_T. Let $V(T)$ be the set of indices of the vertices of triangle T and y_j be the observed reward by a vertex $\mathbf{x}_j$ where $j \in V(T)$. We define the weight for a triangle T as

$$w_T = a_T^{1-\nu} \cdot s_T^{\delta\nu} \tag{4.2}$$

where the score for the triangle is

$$s_T = e^{\max\limits_{j \in V(T)} \{y_j\}}. \tag{4.3}$$

The first term functions similarly to the first term of the original algorithm. The second term in our adaptation behaves like the final term of the original algorithm. However, we

do not explicitly record temporal information about the samples. Since we have potential negative rewards from our objective function, we take the reward to be a power of e. This results in strictly positive scores making it easier to calculate selection probabilities. Instead of a gradient term, in our implementation we use δ, a tunable parameter that controls the rate of exploration of the space. This, like the gradient term in the original algorithm increases the likelihood of choosing triangles with high scoring vertices during sampling. ν is the *temperature parameter*. Initially, $\nu = 1/n$ where n is the total number of iterations of DS to be performed and at each iteration ν is incremented by $1/n$. Note that if a_T is large then w_T can be large, encouraging the algorithm to refine large regions; if s_T is large then w_T can be large, encouraging the algorithm to focus on regions with higher observed rewards. As ν increases, more weight is put on refining the high-valued regions, since ultimately only the high-value regions are used in the selection phase of the algorithm.

Algorithm 1 Pseudocode of Delaunay Sampling

function DELAUNAYSAMPLING($f(), u, n, k, \delta, z$)
  $\mathbf{X} \leftarrow \{\mathbf{x}_1, \ldots, \mathbf{x}_u\}$ spread uniformly in $\mathcal{X}$
  $\mathbf{y} \leftarrow \{y_1, \ldots, y_u \mid y_i = f(\mathbf{x}_i)\}$
  $\mathcal{T} \leftarrow DelaunayTriangulate(\mathbf{X})$
  $\tau \leftarrow \frac{1}{n}$
  for $1, \ldots, n-1$ **do**
    for all $T \in \mathcal{T}$ **do**
      $s_T \leftarrow e^{\max\limits_{j \in V(T)} \{y_j\}}$
      $w_T \leftarrow a_T^{1-\nu} \cdot s_T^{\delta\nu}$
    end for
    for all $T \in \mathcal{T}$ **do**
      $p_T \leftarrow \frac{w_T}{\sum\limits_{T \in \mathcal{T}} w_T}$
    end for
    for $i = 1, \ldots, k$ **do**
      randomly select T from $\mathcal{T}$ where $P(T \mid \mathcal{T}) = p_T$
      $\mathbf{x} \sim \mathcal{U}(T)$
      $\mathbf{X} \leftarrow \mathbf{X} \cup \mathbf{x}$
      $\mathbf{y} \leftarrow \mathbf{y} \cup f(\mathbf{x})$
    end for
    $\mathcal{T} \leftarrow DelaunayTriangulate(\mathbf{X})$
    $\tau \leftarrow \tau + \frac{1}{n}$
  end for
  for all $T \in \mathcal{T}$ **do**
    $\hat{w}_T \leftarrow \frac{1}{|V(T)|} \sum\limits_{j \in V(T)} y_j$
  end for
  $arms \leftarrow$ incenter of top z best triangles
  $\mathbf{x}^* \leftarrow UCB(arms)$
  return $\mathbf{x}^*$
end function

4.3 Selecting the Optimal Point

The previous phase will have spent a large portion of the sampling budget in promising regions of $\mathcal{X}$ ensuring an accurate approximation of the function in those regions. What remains is to select a point as an approximate solution for the optimization problem.

To do so, DS sets the weights of the triangles produced in the previous phase to be the mean of the scores observed at the vertices of each triangle. Next, i triangles with the highest weights are chosen. Then UCB is run with these triangles as arms of a bandit. When UCB samples a triangle, the sample is taken at the triangle's incenter, and its value is a (stochastic) sample of the objective function at that point. The triangle that achieved the highest average reward during UCB is selected and the incenter is the action returned by DS.

Different weighting functions are used in the two phases because of the different purposes they serve. The first phase is exploratory, trying to find promising regions that are relatively small. For this purpose it makes sense to use an optimistic score for a region. The max value observed at any vertex of a triangle provides an optimistic estimate of the expected value of the objective function in the region of the triangle (similar to B-values in HOO (§ 3.1.2.)) The aim of the second phase is to identify the incenter of a region with the largest expected value. For this purpose it is appropriate to use the mean value of the scores observed at the vertices to weight the triangles as an estimate (less optimistic than the max value) of the true expected value of the function in the region within a triangle.

In our description of Delaunay sampling we have only considered the case where the domain of the objective function is 2-dimensional. However, the algorithm can easily be generalized for any metric space $\mathcal{X} \subseteq \mathbb{R}^d$ where $d \geq 2$, although at higher dimensions performance may deteriorate.

Figure 4.3 shows the heatmap of the function of a typical hammer state in curling. The parameters of this function are x_1 and x_2 which are the x- and y-axes respectively of the heatmap and the intensity of the color at any point (x_1,x_2) represents the expected value of the function at that point. Figures 4.4 through 4.5(b) illustrate the first 3 steps of the sampling phase of Delaunay sampling.

Delaunay sampling is initialized by taking 100 samples in a grid in the space over which the function is defined. The resulting scoring of the triangles is illustrated in Figure 4.4 where a darker color marks a region with a higher score. This results in several large darker regions that are scored higher than other regions in the space. The triangles in these regions will receive high weights and there is a higher probability of sampling in these regions in the next iteration.

The dots in Figure 4.5(a) are the new samples taken in the next iteration and the darker regions in Figure 4.4 did indeed receive a bulk of the new samples and the new triangulation

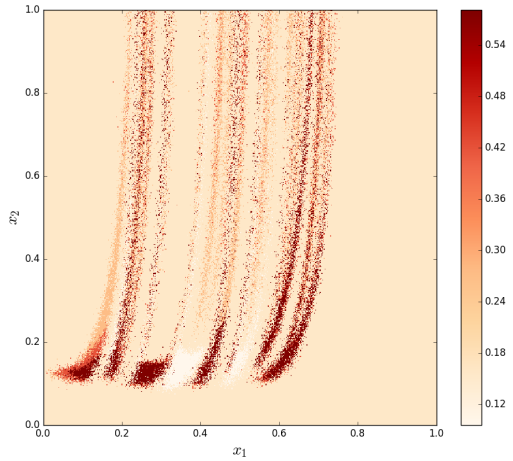


Figure 4.3: Heatmap illustrating the form of a non-convex 2-dimensional function

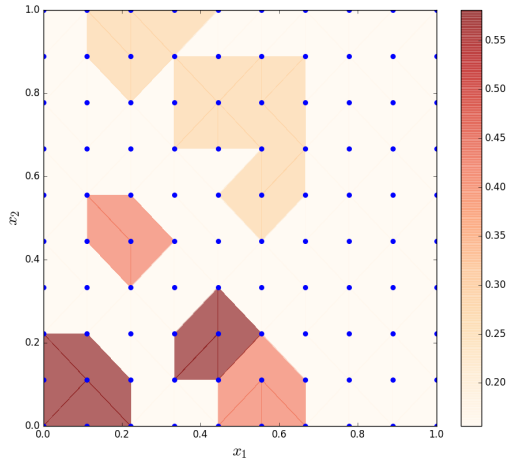
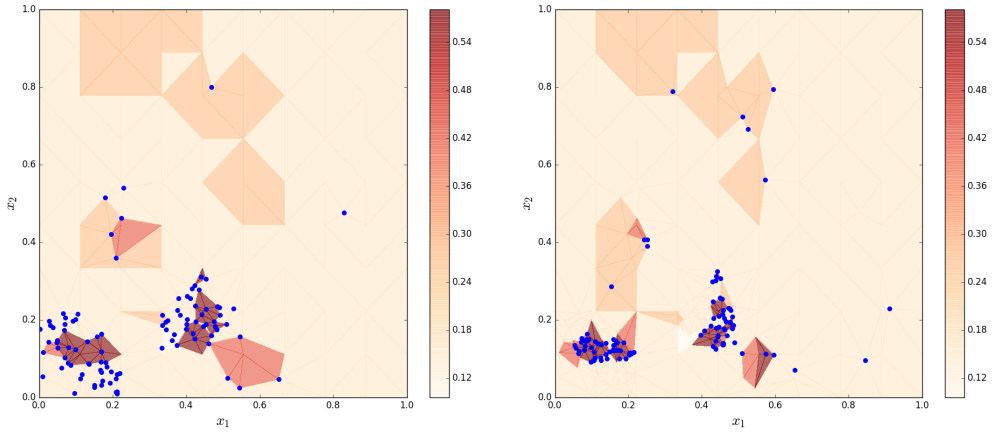


Figure 4.4: Initial step of DS.

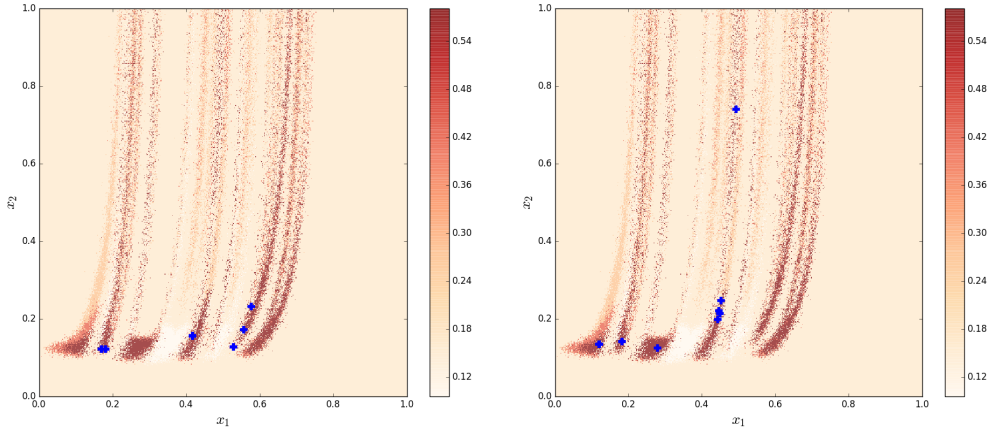
has discretized these regions further. The new triangulation now has smaller regions with high scores as the samples have refined the discretization within the promising regions. The refinement of these regions continues in step 3 of DS (shown in Figure 4.5(b)) and the function approximation becomes more informative than previously about the changes in the function values in these regions.

Figure 4.6 illustrates the selection phase of DS after having observed 900 samples over 9 iterations in the first phase. Figures 4.6(a) and 4.6(b) show two sets arms chosen during two independent runs of DS. Since there may be several triangles with the same weight or the weights may not be exactly the same over independent runs of DS, there is some semblance of randomness when selecting triangles for UCB.



(a) Step 2 of DS (b) Step 3 of DS

Figure 4.5: Sampling phase of DS.



(a) 1st run of DS (b) 2nd run of DS

Figure 4.6: Selection of arms for UCB

4.4 Preliminary Analysis

4.4.1 The Phases of DS

It is pertinent to understand the importance of each of the phases of Delaunay Sampling. The first phase focuses on efficiently sampling in the space to find promising action. The second phase uses UCB, which is used to select the next best point among the candidate points to evaluate at each iteration of the phase [Auer *et al.*, 2002].

To empirically evaluate the performance of these two phases separately we ran each phase independently on a set of 515 hammer states from data from the 2010 Winter Olympics men's and women's curling competition. For each of the test states, we gave the phases a sampling

Budget	500	1000	1500	2000	2500	3000
DS no UCB	-2.67	2.49	4.54	5.47	6.05	6.38
UCB	-4.35	2.69	4.80	6.21	6.24	7.46
DS	3.52	7.33	8.63	9.17	9.43	9.59

Table 4.1: Average Δ_{wp} of the phases of DS separately and together.

δ	0.001	0.01	0.1	1	10	100
Δ_{wp}	4.57	4.48	4.51	4.75	7.13	7.49

Table 4.2: The performance of DS with different values of δ.

budget of between 500 and 3000 samples. After a phase selected a shot 10 outcomes were sampled with the simulator, with the outcome's sample mean used as the resulting estimate of WP. This procedure was repeated 250 times for each test state. Table 4.1 shows the performances of both phases run independently in terms of Δ_{wp}, the difference between the WP of a team before and after taking an action. The table also shows the performance of DS (both phases together) when similarly run on these states. We can see that together both phases work better than they do independently. The full details of the experimental design, the experiments and results can be found in Chapter 5.

4.4.2 Parameter Sensitivity

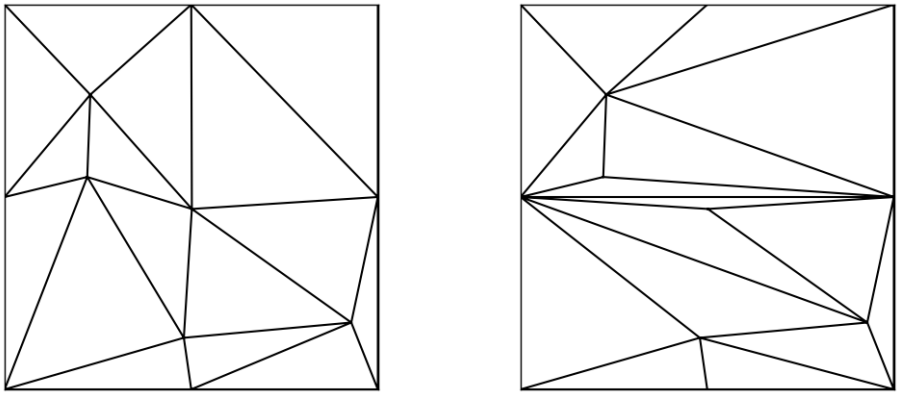
Table 4.2 shows the performance of DS at different values for the exploration parameter δ when run with 1500 samples. A set of 397 hammer shot states from the logs were used to find examine the performance of DS with different values for the parameter δ. These states were separate from the 515 states mentioned above. As can be seen, a value of $\delta \geq 10$ achieves optimal performance. Below that, DS does not concentrate sampling on the promising regions in the function space (in the domain of curling) to successfully find the optimal action.

4.5 Normalized Delaunay Sampling

In the normalized version of Delaunay sampling (NDS), the parameters of the objective function are scaled such that they are in the interval $[0, 1]$. Therefore, every sample point in the original space $\mathcal{X} \subseteq \mathbb{R}^2$ undergoes a transformation to be a point in the new, normalized space $\mathcal{X}' \subseteq [0, 1] \times [0, 1]$. That is, a point $\mathbf{x}_i \in \mathcal{X}$ is transformed by the following operation to give the corresponding point $\mathbf{x}'_i$ in the new space:

$$\mathbf{x}'_i = \boldsymbol{\alpha} \circ \mathbf{x}_i + \boldsymbol{\beta} \tag{4.4}$$

where $\boldsymbol{\alpha} = \left[\begin{smallmatrix}\alpha_1 \\ \alpha_2\end{smallmatrix}\right]$, the multiplicative scaling of the coordinates of $\mathbf{x}$, and $\boldsymbol{\beta} = \left[\begin{smallmatrix}\beta_1 \\ \beta_2\end{smallmatrix}\right]$, the translation of $\mathbf{x}$. The operation $\boldsymbol{\alpha} \circ \mathbf{x}$, is the *Hadamard product* or the elementwise product



(a) (b)

Figure 4.7: The effect of anisotropic scaling on Delaunay triangulation.

of the two vectors.

To understand how normalization affects the performance of DS, it is important to identify which steps of the algorithm normalization alters and which steps it does not. Delaunay sampling can be broken down into several key steps:

1. Sampling
2. Triangulation
3. Weighting
4. Resampling
5. Selection for UCB
6. Incenter sampling

Normalization involves *anisotropic scaling* of the original point space. This causes the triangles to be stretched in the dimension of one parameter (angle of the shot) and shrunk along the other dimension (velocity of the shot). As a result, the angles created between edges produced during Delaunay triangulation are changed. Ultimately, normalization changes the triangulation mesh that is produced when the point set in $\mathcal{X}$ is transformed to the space in $\mathcal{X}'$. Figure 4.7 illustrates how scaling can affect the mesh produced. Figure 4.7(a) shows the Delaunay triangulation of a point set. Figure 4.7(b) depicts the Delaunay triangulation of the point set from Figure 4.7(a) where all points are scaled by .5 on the x-axis and by 100 on the y-axis.

Anisotropic scaling of the triangles also changes the corresponding incenters of the triangle. That is, the incenter of a triangle T in $\mathcal{X}$ does not correspond to the incenter of T in $\mathcal{X}'$. For instance, Figure 4.8(a) illustrates a $\triangle ABC$ with incenter D. Figure 4.8(b) shows

$\triangle A'B'C'$ where the points A, B, C have undergone an anisotropic scaling to give points A', B', C'. However this triangle has incenter E which is not equivalent to the transformation of point D.

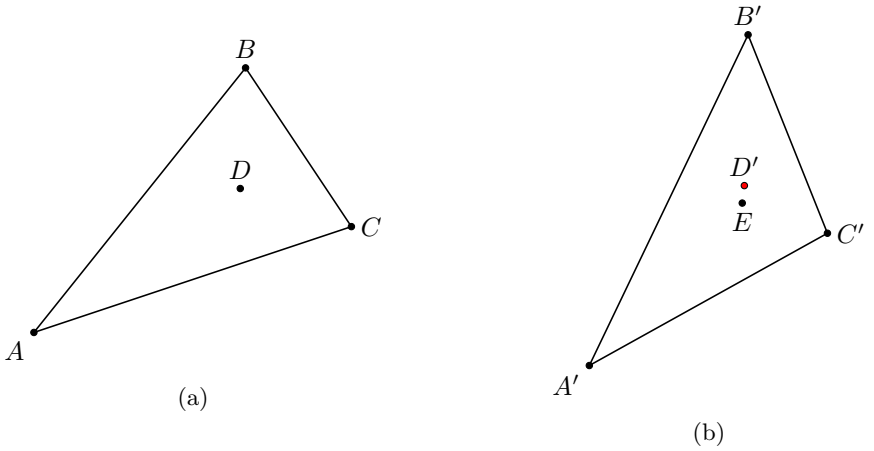


Figure 4.8: The effect of anisotropic scaling on the incenters of the triangles

However, normalization affects neither the selection of triangles for resampling or for UCB nor the uniform sampling of a point within a triangle.

The Effect of Normalization on Selection of Triangles for Resampling

Equation 4.2 shows how the weights for each triangle is calculated. The weight of a triangle T depends on its score and its area. The score of a triangle is invariant to normalization. The area of a triangle with coordinates $(x_1, y_2), (x_2, y_2), (x_3, y_3)$ is calculated by

$$a_T = \frac{x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)}{2}. \tag{4.5}$$

Normalization would result in the area with the transformed coordinates calculated by

$$\begin{aligned}
a'_T &= \frac{x'_1(y'_2 - y'_3) + x'_2(y'_3 - y'_1) + x'_3(y'_1 - y'_2)}{2} \\
&= \frac{(\alpha_1 x_1 + \beta_1)\alpha_2(y_2 - y_3) + (\alpha_1 x_2 + \beta_1)\alpha_2(y_3 - y_1) + (\alpha_1 x_3 + \beta_1)\alpha_2(y_1 - y_2)}{2} \\
&= \frac{\alpha_1\alpha_2 x_1(y_2 - y_3) + \beta_1\alpha_2(y_2 - y_3)}{2} + \frac{\alpha_1\alpha_2 x_2(y_3 - y_1) + \beta_1\alpha_2(y_3 - y_1)}{2} \\
&\quad + \frac{\alpha_1\alpha_2 x_3(y_1 - y_2) + \beta_1\alpha_2(y_1 - y_2)}{2} \\
&= \frac{\alpha_1\alpha_2 x_1(y_2 - y_3) + \alpha_1\alpha_2 x_2(y_3 - y_1) + \alpha_1\alpha_2 x_3(y_1 - y_2)}{2} \\
\therefore\ a'_T &= a_T\alpha_1\alpha_2.
\end{aligned} \tag{4.6}$$

The weight of a triangle in the normalized space can then be written in terms of the weight

in the original space as

$$w'_T = w_T \alpha_1 \alpha_2. \tag{4.7}$$

As a consequence, if the Delaunay triangulation of a set of points $\mathbf{X}$ in $\mathcal{X}$ is the same as the triangulation of the corresponding points $\mathbf{X}'$ in $\mathcal{X}'$, then during resampling each triangle T in $\mathbf{X}$ will have the same probability of being selected as each corresponding triangle T' in $\mathbf{X}'$. That is, normalization does not affect the selection of triangles if the triangulations in both spaces are equivalent. It follows that normalization does not affect the selection of triangles for the UCB phase either as all the mean weights of the triangles would be multiplied by the same constant.

The Effect of Normalization on Sampling Within a Triangle

A point $\mathbf{x}$ can be sampled uniformly within a triangle in space $\mathcal{X}$ with vertices $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$ using

$$\mathbf{x} = (1 - \sqrt{r_1})\mathbf{x}_1 + \sqrt{r_1}(1 - r_2)\mathbf{x}_2 + r_2\sqrt{r_1}\mathbf{x}_3 \tag{4.8}$$

where r_1 and r_2 are random numbers between [0, 1]. In the normalized space the new point can be written in terms of the new point in the original space by

$$\begin{aligned}
\mathbf{x}' &= (1 - \sqrt{r_1})\mathbf{x}'_1 + \sqrt{r_1}(1 - r_2)\mathbf{x}'_2 + r_2\sqrt{r_1}\mathbf{x}'_3 \\
&= (1 - \sqrt{r_1})(\boldsymbol{\alpha} \circ \mathbf{x}_1 + \boldsymbol{\beta}) + \sqrt{r_1}(1 - r_2)(\boldsymbol{\alpha} \circ \mathbf{x}_2 + \boldsymbol{\beta}) + r_2\sqrt{r_1}(\boldsymbol{\alpha} \circ \mathbf{x}_3 + \boldsymbol{\beta}) \\
&= (1 - \sqrt{r_1})(\boldsymbol{\alpha} \circ \mathbf{x}_1) + \sqrt{r_1}(1 - r_2)(\boldsymbol{\alpha} \circ \mathbf{x}_2) + r_2\sqrt{r_1}(\boldsymbol{\alpha} \circ \mathbf{x}_3) + \boldsymbol{\beta} \\
\therefore\ \mathbf{x}' &= \boldsymbol{\alpha} \circ \mathbf{x} + \boldsymbol{\beta}
\end{aligned} \tag{4.9}$$

The point sampled in $\mathcal{X}'$ is equivalent to the corresponding point that would be sampled in $\mathcal{X}$. Therefore, normalization does not affect how a point is sampled within a triangle (given that the triangulations are the same in the original space and the normalized space.)

Chapter 5

Experiments and Results

5.1 Experimental Design

The data used for our experiments was drawn from the hammer shot states captured by hand from the 2010 Winter Olympics men's and women's curling competition.[1] A set of 397 hammer shot states from these logs were used in the parameter sweeps mentioned below. The parameter sweep for Delaunay Sampling (DS) chose a value of 17 for σ. A separate set of 515 hammer shot states (the "test states") from these logs were used to evaluate the systems (including the humans).

For each of the test states, we gave DS a budget of between 500 and 3000 samples. This budget includes 100 samples for initializing the triangulation over each turn, 100 samples per iteration of the first phase, and 100 samples for the final UCB phase. After DS selected a shot, 10 outcomes were sampled from the simulator, with the outcome's sample mean used as the resulting estimate of WP. This procedure was repeated 250 times for each test state. The values we report in Table 5.1 are the average over the 2500 evaluations for each test state. DS was also tested with the values for angle and velocity normalized to values in the interval $[0, 1]$. These results are shown in the table under the label NDS. The normalization of the action parameters was also done for the other optimization algorithms (HOO, PSO, CMA and GPO).

When implementing HOO, for every state, the action space of the objective function for each turn was covered by a separate tree. At each iteration, the tree to be expanded was selected using UCB. We ran HOO for 250 trials over the test states using the same sampling budgets as DS. The parameters for HOO described by Bubeck et al. [2009b] were set by a parameter sweep to $\rho = \frac{1}{\sqrt{2}}$, $\nu = 2\sqrt{2}$ and UCB constant $C = 0.01$.

PSO was tested slightly differently. Since the particles move in a continuous space, having a discrete parameter ("turn") required us to run PSO on each turn separately. For each test state, PSO was run using one turn value and then the other with each run being

[1] The logs do not contain the actual shot played by the humans they only contained the states before and after the hammer shots were taken.

provided half the sampling budget. The best shots found with each turn were compared and the one with the higher average WP was evaluated one final time to compute its expected WP. This was performed 250 times for each test state. The values for the parameters for PSO were selected using a parameter sweep and were set to $c_1 = 1.5$, $c_2 = 1.5$, $w = 0.7$ and 50 particles.

We used the BIPOP-CMA-ES [Auger and Hansen, 2009] version of CMA-ES for our tests. The experimental setup was the same as for PSO. The parameters for the number of offspring and parents were, respectively, $\lambda = 4 + 3\log 2$ and $\mu = \lambda/2$ with 4 restarts. We set the initial standard deviation to $\sigma_0 = 0.25$ and normalized the action parameters θ and v to range between 0 and 1. The parameters for μ, λ, step-size control and the covariance matrix adaptation were set in accordance to the values recommended by Hansen [2016].

GPO was tested in a manner similar to PSO. However, since GPO is considerably slower we implemented it with only a budget size of 250, 500 and 1000 samples per turn with 50 restarts for GP regression and ran one trial of the algorithm at each budget size. In addition, at each budget size, we ran GPO only once and evaluated the shot chosen 2500 times. Although there are many choices for covariance functions, we chose the squared-exponential function as the GP kernel. We also selected GP-UCB as the acquisition function and set the noise variance $\sigma = 0.01$ using a parameter sweep.

KR-UCT was initialized with a uniform grid of 100 samples over each turn as the candidate set. At each iteration, the action with highest action value over both turns is selected. KR-UCT was run with budget size of 3000 samples. Due to the slow speed of KR-UCT, it was only run for one trial for each hammer shot and the shots selected after 500, 1000, 1500, 2000, 2500 and 3000 iterations were returned. Each of these shots were then executed 2500 times and the expected value was calculated. The value of $C = 0.1$ was chosen by Yee et al. [2016] for our application.

UCB was also tested to observe its performance. In this case, the actions for UCB were created in three different manners. In the first case (UCB+Uni), for each state the algorithm was run on each turn separately with each turn given half the sampling budget. 5% of the budget was then spent on creating a uniform grid of samples in the action space and the rest of the budget was used to run UCB. This was repeated 250 times. This is the preliminary testing described in §4.4.1. For UCB+Rand (second case), the same setup was used as UCB+Uni except this time the actions were created by uniform randomly sampling the space instead of using a grid. For the final case (UCB-Gen), the actions for UCB were created using domain knowledge. One of the key features our simulator includes is a set of shot generators which use expert human knowledge to analyze a given state and create a set of actions that an expert player would be most likely to recognize. The shots created using this expert knowledge were used to initialize UCB-Gen. UCB-Gen was run for each

Budget	500	1000	1500	2000	2500	3000
DS	3.52	7.33	8.63	9.17	9.43	9.59
NDS	8.22	9.58	9.84	9.92	9.98	9.96
HOO	-1.39	3.76	5.56	7.63	7.98	8.84
PSO	5.40	5.48	5.46	5.45	5.48	5.47
CMA	5.20	5.46	5.41	5.21	5.42	5.36
GPO	4.38	4.49	-	4.48	-	-
KR-UCT	8.05	8.79	8.94	9.11	8.85	8.92
UCB	-4.35	2.69	4.80	6.21	6.24	7.46
UCB+Gen	9.39	9.50	9.56	9.57	9.58	9.59
DS+Gen	6.57	8.53	9.36	9.61	9.75	9.81

Table 5.1: Average Δ_{wp} for algorithms initialized with uniform sample grid.

Budget	500	1000	1500	2000	2500	3000
DS	4.95	8.01	9.07	9.51	9.76	9.90
NDS	8.24	9.64	9.88	9.97	9.99	9.99
UCB	0.90	4.24	5.62	6.52	7.08	7.47
Rand	5.43	5.53	5.52	5.51	5.50	5.50

Table 5.2: Average Δ_{wp} for algorithms initialized with random samples.

test state with the given sampling budgets 250 times. (In addition, domain knowledge was also used in the initial sample set for DS and the results for these experiments are shown in Table 5.1 under DS+Gen.)

Finally, as a benchmark against which to compare the performance of all the algorithms, the experiments were also performed by selecting an action at random. For each state, each turn had received half the sampling budget, all of which went toward sampling randomly in the action space of each turn. The action with the highest observed value was selected and its expected value calculated. This procedure was performed 250 times for each state.

To determine if DS's average WP on the test states was statistically significantly different than the average WP of the humans or other systems tested, we ran a Wilcoxon signed-rank test (WSRT) with one pair of values for each of the 515 test states (one of the values was DS's WP on the state, the other was the WP of the system DS was being compared to). Unless otherwise noted, WSRT produced p-values sufficiently small to conclude with greater than 99% confidence that DS's superior average WP (for any specific sample budget) is not due to chance.

5.2 Analysis

In § 2.4, we described the function that we are trying to optimize using the various algorithms. While the algorithms are indeed optimizing over actions to find the action that achieves the greatest WP, we report the results of their performances in terms of gain in WP, Δ_{wp}. That is, for any state an optimization algorithm is run on, we record the WP of

the hammer team before taking an action. Then we find the WP of the team after taking the action selected by the algorithm. Δ_{wp} is the difference between these two values of WP.

Table 5.1 shows the performance in terms of average Δ_{wp} over all test states for each of the optimization methods in our study. The performance of **DS** increased as we increased the number of samples in its sampling budget. With a low sampling budget, DS had significantly worse performance than the other optimization algorithms, excluding HOO and UCB. It surpassed the performance of PSO, CMA and GPO with a budget of at least 1000 samples. Its performance, however, was consistently better when it was initialized with random samples instead of uniform samples. The random initialization increased the chances of DS finding good actions in the beginning of each trial.

NDS performed better than DS and significantly better than all other algorithms over all sampling budgets.

Budget	100	150	200	250	300	350	400	450	500
NDS	-2.86	4.50	6.21	6.99	7.46	7.81	8.07	8.24	8.40

Table 5.3: NDS with smaller sampling budgets.

Since, NDS exceeded the performance of other algorithms by a large margin with even the smallest sampling budget, NDS was tested with even fewer samples. In these experiments, NDS was given sampling budgets from 100 through 500. It was initialized by a 5×5 grid of samples. At each iteration of the algorithm, 50 new samples were observed. The UCB phase was given a budget of 50 samples (at a sampling budget of 100, NDS did not refine the triangulation with new samples.) The results of these trials are shown below in Table 5.3.

HOO performed worse than DS for smaller sample budgets because it was slower to build its cover tree deep enough in the more promising parts of the action space to accurately locate the best shots. We believe this is largely due to the effects of the bias term on the value of Equation (3.2a). The bias term was scaled using a static UCB constant for all hammer shots. In some states, this may have resulted in much of the sampling budget being used exploring the space (where the constant C is set too high). In other states, this may have caused the samples to prematurely concentrate on a local optimal region (when C is too low). Choosing a different constant for each hammer shot may have helped the performance of HOO however it not a feasible task because it would require a parameter sweep for each state and you would have to pay the cost of those samples to have a fair comparison. However, higher sampling budgets drown out these adverse effects as its performance grows closer to DS's as the sample budget increases.

PSO's Δ_{wp} changed very little as its sample budget increased. This is due to PSO's tendency to converge to local optima of the function space. It may, however, converge to a global optimum if the random initial sampling places a particle on or near a global optimum.

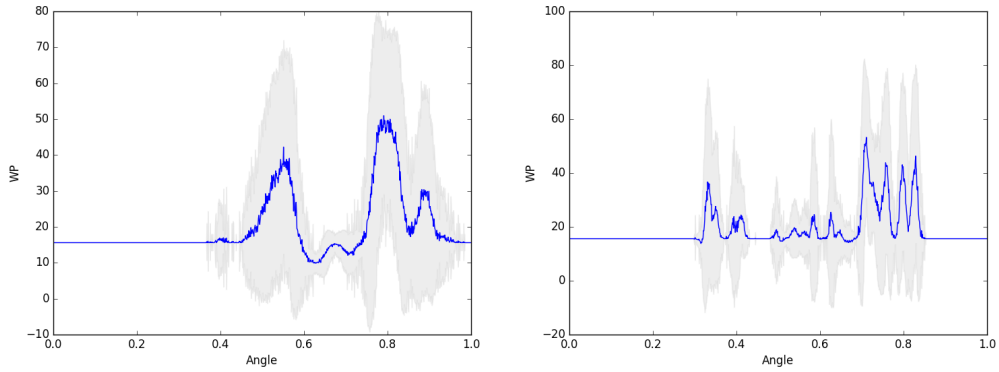
Number of Particles	10	25	50	75	100
Δ_{wp}	4.99	5.39	5.51	5.51	5.43

Table 5.4: Average Δ_{wp} of PSO after 20 iterations with different number of particles.

Increasing the number of particles (at the cost of decreased number of iterations of PSO given a set sampling budget) would increase the chances of PSO finding a global optimum but this reaches a limit in performance increase at 50 particles. Table 5.4 shows how the performance of PSO changes with an increase in the number of particles after 20 iterations of PSO. **CMA-ES** also suffers from convergence to suboptimal local optima.

GPO performs considerably worse than DS when DS is given a sampling budget of at least 1000 samples. One of the main contributing factors to the poorer performance of GPO is its noise parameter. The noise parameter σ provides an estimate of the variance of the distribution of the rewards observed at any point in the action space. The setting of this variance is vital in generating an accurate predictive model of the function and in determining points to sample. Since GPO seeks a global optimal action, one static value for σ over all states may result in excessive exploration of the action space (regions not near promising points sampled so far) depending on the state for which it is trying to find the best action. This is further complicated by the fact that the noise in our application is actually not on the function values but on the values of the parameters that are passed to the objective function (the parameters of the actions are perturbed before an observation.) While this ultimately results in some distribution of rewards with an expected value for each of the actions, these distributions are different for each action in any state (see next paragraph.) So for each state we have *heteroscedastic* noise which one value for σ fails to model. In addition, it is infeasible to set a separate σ for each action since an appropriate σ for each action is not possible without knowing the actual objective function.

To demonstrate this with an example we examined how the value of σ selected affects the performance in a single state. For the state chosen, we chose two set values for velocity to produce two different 1-dimensional functions for which we try to optimize the angle. Figure 5.1 illustrates the two functions f_A and f_B. A velocity $v = 0.1$ and $v = 0.3$ was set for f_A and f_B respectively and the WP achieved over the angles is plotted. The x-axis is the normalized range of angles and the y-axis is the WP achieved. The dark lines are the average WPs observed and the gray region depicts the pointwise variance of the distribution of the WP. GPO was used to find the angle that observes the highest WP. For each function, GPO was run with σ values from $[0.1, 0.01, 0.001, 0.0001, 0.00001]$ given sampling budgets of 15 and 30 samples with 50 trials at each σ. The average expected WP achieved over all trials was then calculated. Table 5.5 show the results of this experiment. As can be seen, looking at the σ value that yields the greatest WP for each functions, even in a single state



(a) f_A (b) f_B

Figure 5.1: Examples of 1-dimensional versions of our objective function.

σ	f_A		f_B	
	15	30	15	30
0.1	33.6	36.9	31.1	34.6
0.01	33.3	36.7	27.5	32.4
0.001	**35.1**	**38.4**	31.8	31.5
0.0001	32.2	37.6	**32.7**	**36.4**
0.00001	33.8	34.6	32.6	33.6

Table 5.5: Average Δ_{wp} achieved with different noise variance.

different values for σ are required to optimize the function at different regions. Despite these disadvantages, we believe that given a more generous sampling budget, it would eventually find better shots, but its running time would be prohibitive.

KR-UCT performs significantly better than vanilla DS for budget sizes of 500, 1000 and 1500. However, NDS significantly outperforms KR-UCT over all budget sizes. This might be due to KR-UCT's tendency to do local exploration (actions near previous actions). The algorithm was originally designed to use domain knowledge for candidate actions and improve upon those actions. Therefore in the optimization scenario we explore, any new shots the algorithm explores would be actions close to the initial uniform grid set of samples. As a result the performance of the algorithm relies entirely on the quality of the initial set of actions.

UCB-Uni and **UCB-Rand** both start off performing poorly at lower sample budgets but increase in performance as their budgets in increase with UCB-Rand performing significantly better than UCB-Uni over all budget sizes. **UCB-Gen** consistently performs well over all sampling budgets. DS only reaches similar performance with a budget of at least 2500 samples while NDS requires only 500 samples. NDS however exceeds the performance of UCB-Gen when given a budget of 1000 samples or more. This might be because UCB-Gen is restricted to selecting the best action solely among the actions produced using domain

knowledge. NDS, however, is not limited by these restrictions and may find newer, better actions. **DS-Gen** performs worse than UCB-Gen at lower sample budgets but exceeds it when given larger budgets. This indicates that while the expert knowledge from humans used in our simulator may help us to find good actions they may not be best actions.

Chapter 6

Comparing to Humans

In the previous chapter, the performance of Delaunay sampling was compared to only that of other algorithms and we learned that, for our application, the normalized DS algorithm outperforms the others when it comes to selecting actions for hammer shots. One of the main goals of this research is to explore the possibility of using AI agents to outperform humans. In this chapter, we examine how the strategies selected by NDS for hammer shots compare to the strategies Olympic-level humans chose.

Blanks	Humans	NDS					
		500	1000	1500	2000	2500	3000
All	4.82	8.22	9.58	9.84	9.92	9.98	9.96
Late	5.52	9.20	10.74	11.02	11.11	11.18	11.17

Table 6.1: Average Δ_{wp} for NDS and Humans

The performance for Olympic-level humans was calculated using the data from the 2010 Olympic's men's and women's games logs. There were a total of 912 states: 515 was used for testing and 397 were used for parameter sweeps. For each of the 515 hammer states on which NDS was tested, the WP that the human team had before the hammer shot was computed from the state information. The state information for the outcomes of the actions taken by humans were then used to compute the WP of the team after the shot was taken. The difference between these two values were the Δ_{wp} Olympic humans observed for each hammer state. Table 6.1 show these results.

The first row of results in the table show the performance of humans over all states. Olympic-level humans had an average Δ_{wp} of 4.822%. NDS outperforms humans significantly (WSRT p-value < 0.01) with a sampling budget of at least 200 samples ($\Delta_{wp} = 6.21\%$, see Table 5.3). An interesting thing to note is that in practice, humans play with one general convention: most teams choose to blank the early ends of the game. To see how this affects the average Δ_{wp} of humans, we also include their performance by removing all the early blanked ends from the data. The second row of results reflect these changes for humans

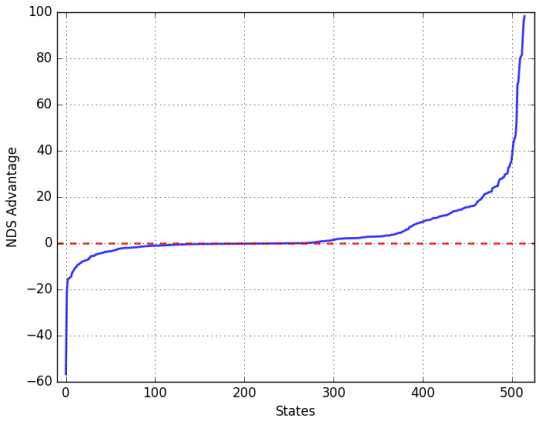


Figure 6.1: NDS advantage over humans for each state.

and NDS. Without the early blanked ends, it can be seen that the average Δ_{wp} for humans increased to 5.52%. Similarly, the performance of NDS also increased.

Figure 6.1 plots the differences between the Δ_{wp} achieved by actions selected by NDS and humans for each state (which we will call *NDS advantage*). The x-axis in the plot is the set of the 515 test states (in order of increasing difference in performance) and the y-axis is the NDS advantage. The blue, solid line is the plotted NDS advantage. The red, dashed line marks the points of zero difference. Wherever the NDS advantage is negative, humans achieved better performance and points where NDS advantage is positive are states where NDS performed better. It can be seen on the graph, for approximately 20% of the states, humans performed worse than NDS (the proportion of the solid line below the zero mark). On approximately half the states, NDS performed better than humans.

In the following two sections, we will examine more closely several states where NDS outperforms humans and vice versa. For each case, we provide the diagram of the state with shot trajectories, **NDS advantage** over humans (the difference between their Δ_{wp} values), the **the intended shot of the human team**, **the shot chosen by NDS** and a brief analysis. We also provide, in terms of WP, the **expected difference** of the action chosen by NDS over humans in cases where the shots they chose are different. The data logs from the actual games contain the type of shot that the human team intended to take for every hammer state. Then using the heatmaps, for each of the cases we studied, we manually identified the parameters for the human team's intended shot that achieved the best expected WP. Then we executed the action with those parameters using the simulator for 2500 times and calculated the expected Δ_{wp} for the intended human shot. We then provide the difference between this expected value and the expected value for the action chosen by NDS. In the diagrams, trajectories depicted as dashed lines are the human shots, solid lines are the NDS shots. *Out turns* are trajectories curving upwards and *in turns* are

trajectories curving downwards. Red rocks (darker rocks in grayscale) in the state pertinent to the hammer shot selection are labelled as r_i and yellow rocks (lighter shaded rocks) are similarly labelled as y_i. For the sake of simplicity, all hammer teams are throwing red rocks in these examples.

6.1 Cases NDS Exhibits Improvement

6.1.1 Case 1

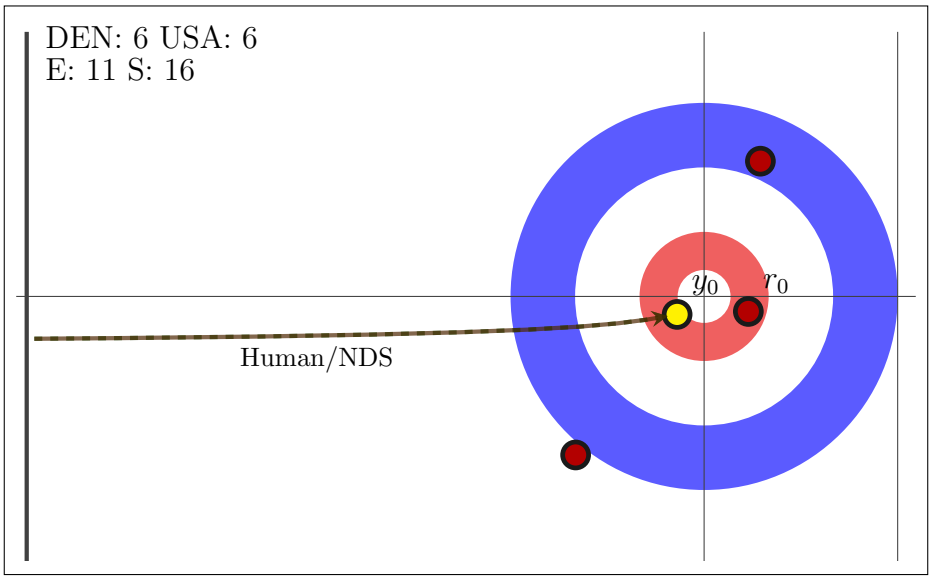


Figure 6.2: NDS v Humans Case 1

NDS Advantage: 98.28%[1]

Expected Difference: -

Human Shot/Turn: Clearing/Out

NDS Shot/Turn: Clearing/Out

The first case (Fig 6.2) that we examine, is the case where NDS shows the most improvement. It is the last shot of the 11th end with the scores tied. The yellow team is currently sitting at winning 1 point for the end to win the game, with y_0 being closest to the button. The red team intended to use their shot to knock out y_0 from play to allow them to score at least 3 points (depending on where the shot rock ends.) However, instead they ended up hitting the y_0 into r_0, knocking their own rock out instead with y_0 being the closest to the button scoring 1 point for the yellow team. NDS actually chose the same shot as humans

[1]It is important to note that these differences are not exactly 100% because the shots NDS chooses have a chance of blanking the end (scoring 0).

but executed it with the intended result a majority of the time during repeated trials. In this case, the humans were just "unlucky" in their shot execution.

6.1.2 Case 2

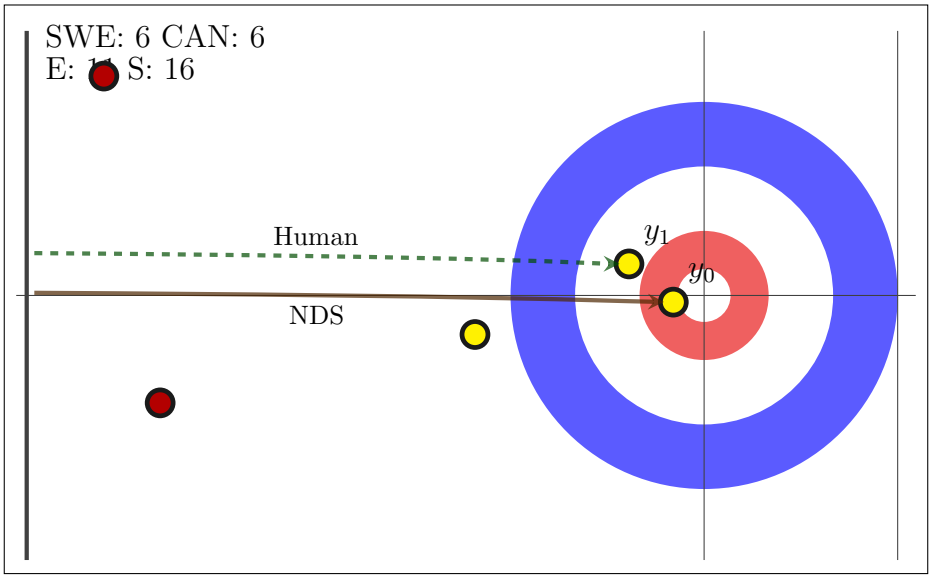


Figure 6.3: NDS v Humans Case 2

NDS Advantage: 96.22%

Expected Difference: 22.3%

Human Shot/Turn: Double Takeout/In

NDS Shot/Turn: Takeout/In

The second case (Fig 6.3) shows a situation of the last shot of the 11th end with scores tied. The yellow team has two rocks y_0 and y_1 in the house and are currently sitting on two points. The humans decided to hit y_1 with their shot rock into y_0 with the intent of knocking both rocks out of play to score 1 point to win the game. However, they ended up hitting y_1 straight on and only knocking that rock out resulting in a loss of 1 point for the end. NDS decided to only try to take out y_0 instead. This shot has a 22.3% improvement on the action selected by humans. This is probably a more robust shot because taking out one rock is more likely to be successful than taking out two rocks. Furthermore, even though the gap to hit y_0 is not a large width of space, takeouts are much straighter in trajectory than other shots so it is not too difficult a shot.

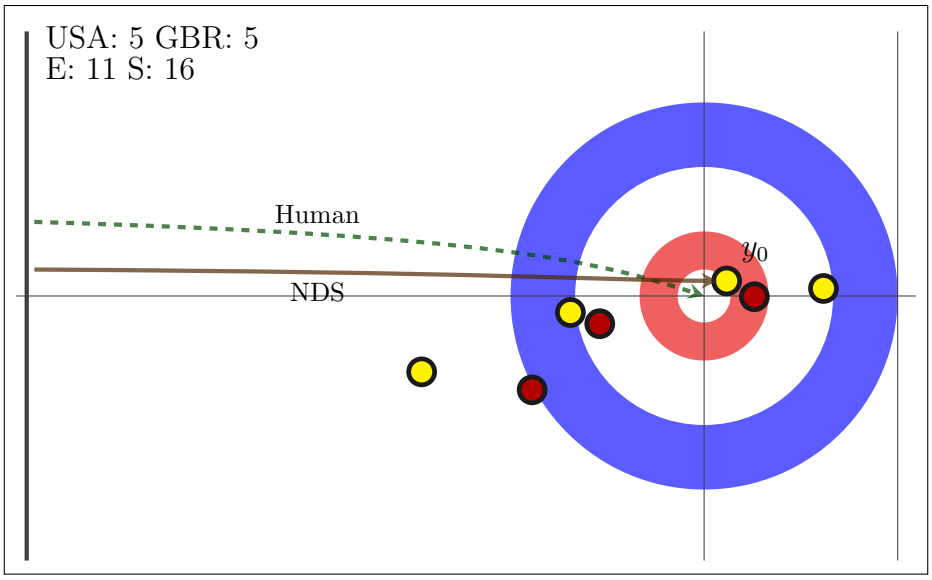


Figure 6.4: NDS v Humans Case 3

6.1.3 Case 3

NDS Advantage: 89.6%

Expected Difference: 1.7%

Human Shot/Turn: Draw/In

NDS Shot/Turn: Takeout/In

Fig 6.4 illustrates the third case NDS chooses a better shot than humans. It is the last shot of the 11th end with the scores tied. The yellow team has one rock y_0 in position to score a point to win the game with a red rock sitting just behind it. The humans decided to draw in closer to the button right in front of y_0. However, the human player missed this shot completely. They ended up shooting a little too strongly (since they wanted to use y_0 for backing) and did not have enough curl for the rock to stop in the button. NDS decided to take out y_0. Similar to the previous case, a takeout is probably a more robust shot (this shot has and expected value of 96.7% while the draw has an expected value of 95.0%) because it will be straighter and the shooter would not have to be as rigid about the shot velocity with a takeout as with a draw. However, both shots have high expected values and the human team failed to execute it.

6.1.4 Case 4

NDS Advantage: 81.52%

Expected Difference: -

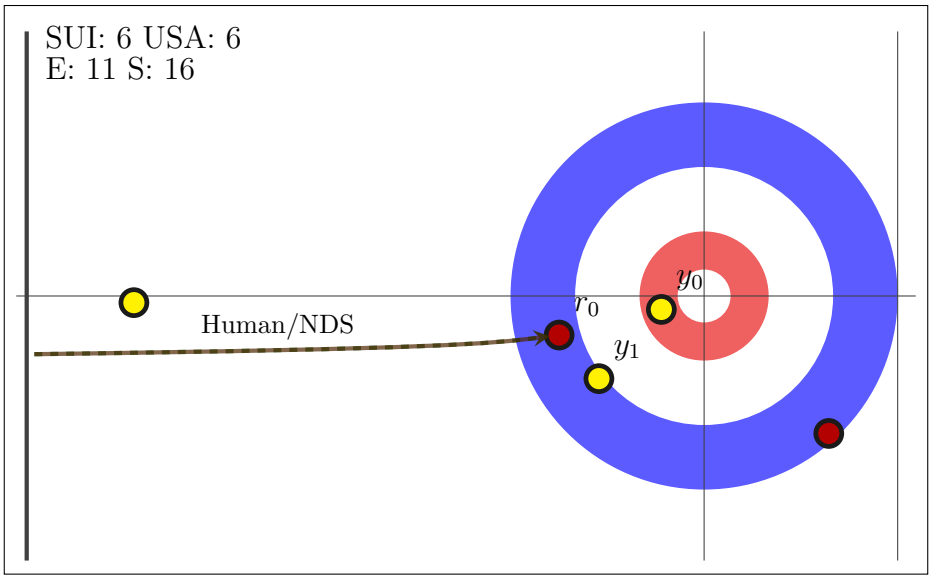


Figure 6.5: NSD v Humans Case 4

Human Shot/Turn: Promotion Takeout/Out

NDS Shot/Turn: Promotion Takeout/Out

Fig 6.5 is another case where NDS chooses the same shot the humans did. This state is the hammer shot of the 11th end of a game and the scores are tied. In this case, both teams have two rocks each in the house. However, rocks y_0 and y_1 are the closest to the button, so the yellow team is currently sitting to gain 2 points and win the game. The hammer team decided to *promote* their rock r_0 into y_0 to remove it from the house with the hope that r_0 would end up being the closest rock to the button. NDS chose to execute the exact same shot. However, during execution, r_0 indeed ended up hitting y_0 out of the house but r_0 then travelled further resulting in y_1 becoming the closest rock to the button. As a result, the human team with the hammer lost a point to the other team. This marks another case, where both humans and NDS chose the same but humans suffered bad luck during execution. It is important to note that NDS chose to ignore drawing in to the button in this case as well suggesting that draw shots are probably harder to execute.

6.1.5 Case 5

NDS Advantage: 80.77%

Expected Difference: -16.48

Human Shot/Turn: Draw/Out

NDS Shot/Turn: Promotion Takeout/Out

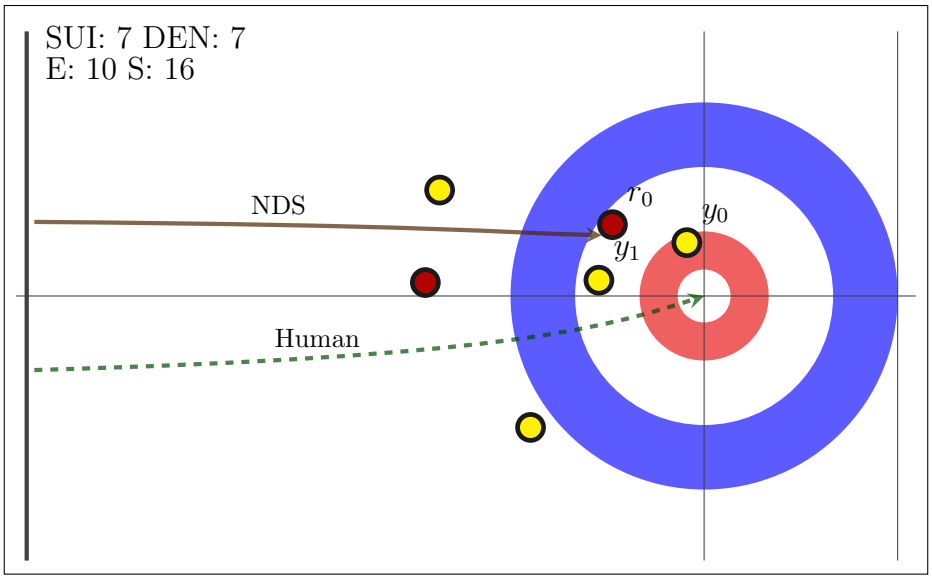


Figure 6.6: NDS v Humans Case 5

Fig 6.6 is the final example we will look at where NDS outperformed humans in our experiments. It is the hammer shot for the 10th end of the game and scores are tied. In this case there are three rocks in the house: two yellow rocks y_0, y_1 and one red rock r_1. The yellow team is set to gain two points with y_0, y_1. The hammer team decided to draw to the button with an out turn to score 1 point. However they failed to execute it and lost 2 points. This is a straightforward shot in that there are no rocks in the way (the one yellow rock at the bottom is too far to be an obstacle.) However, it still requires the shooter to control the shot's velocity to not be too high or too low. The intended shot selected by humans has an expected value of 86.52%. NDS chose a shot that is more complex than a draw. It chose to take out both y_0 and y_1. This requires the shot bounce between r_0 and y_0 and then move on to hit y_1. This is a case where NDS chose a shot that has a lower expected value (70.04%) than the intended shot selected by humans but still did better (humans were probably unlucky during execution.) NDS consistently chose the double takeout on repeated trials.

6.2 Cases NDS Exhibited Deterioration

6.2.1 Case 6

NDS Advantage: -56.73%

Expected Difference: -

Human Shot/Turn: Promotion Takeout/Out

NDS Shot/Turn: Promotion Takeout/Out

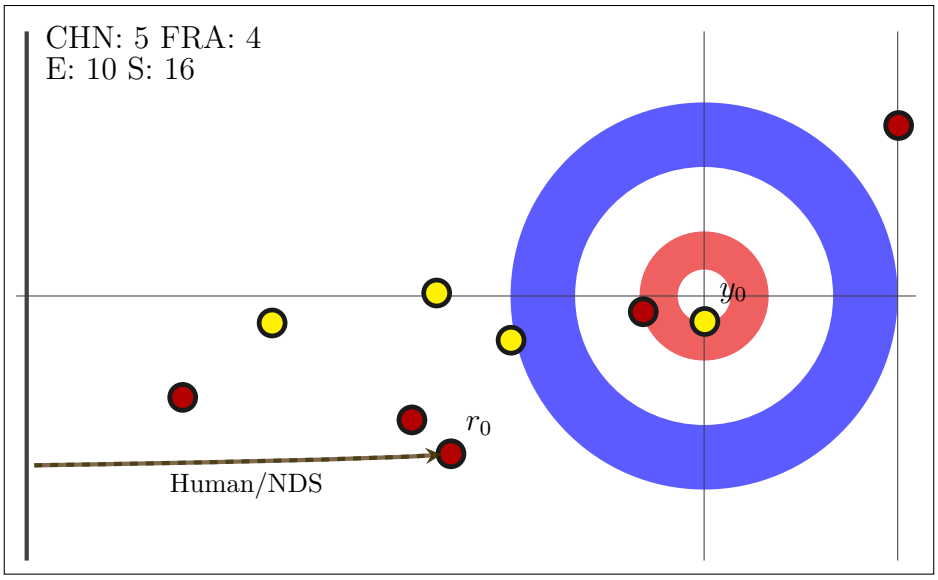


Figure 6.7: NDS v Humans Case 6

Case 6 (Fig 6.7) is the first example we will look at where NDS does worse than humans. It is the final shot of the 10th end and the red team is down by 1 point. The yellow team currently has a rock y_0 closest to the button and has a potential score of 1 point for the end. The red team needs 2 points to win the game and 1 point to blank the end and move to an extra end. In this case humans chose to hit their own rock r_0 and promote that to take out y_0. They executed the shot perfectly and scored 2 points. NDS found the same shot to be the best possible action to take. Unfortunately this is a very difficult shot to execute perfectly (it has an expected value of 38.5%). A draw in to the button seems like an easier shot to take but it would at most score 1 point. This would result in a tied score and another end would have to be played, but in this end the other team would possess the hammer and have an advantage. The promotion takeout is the only viable shot that allows the hammer team to win. In this situation, the human team successfully executed a difficult shot.

6.2.2 Case 7

NDS Advantage: -21.85%

Expected Difference: -

Human Shot/Turn: Promotion Takeout/Out

NDS Shot/Turn: Promotion Takeout/Out

Case 7 (Fig 6.8) is similar to the previous case. It is the final shot of the 7th end of a game and the red is leading by 1 point. The yellow team currently has a rock y_0 closest to the

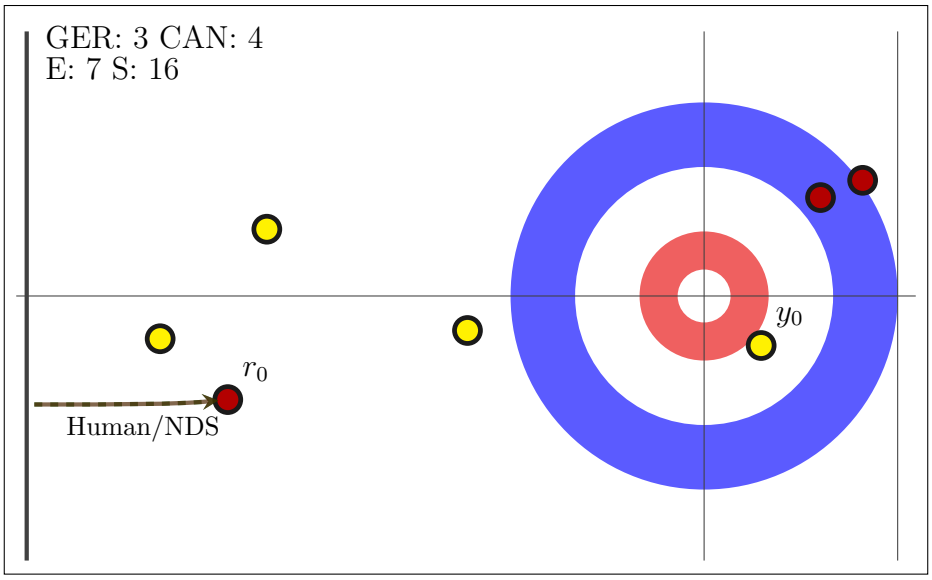


Figure 6.8: NDS v Humans Case 7

house and has a potential score of 1 point for the end. Again humans chose to hit their own rock r_0 and promote that to take out y_0. They executed the shot perfectly to score 3 points. NDS found the same shot to be the best action to take. Just like the previous example, humans executed a shot that was not easy (expected value of 63.0%).

6.2.3 Case 8

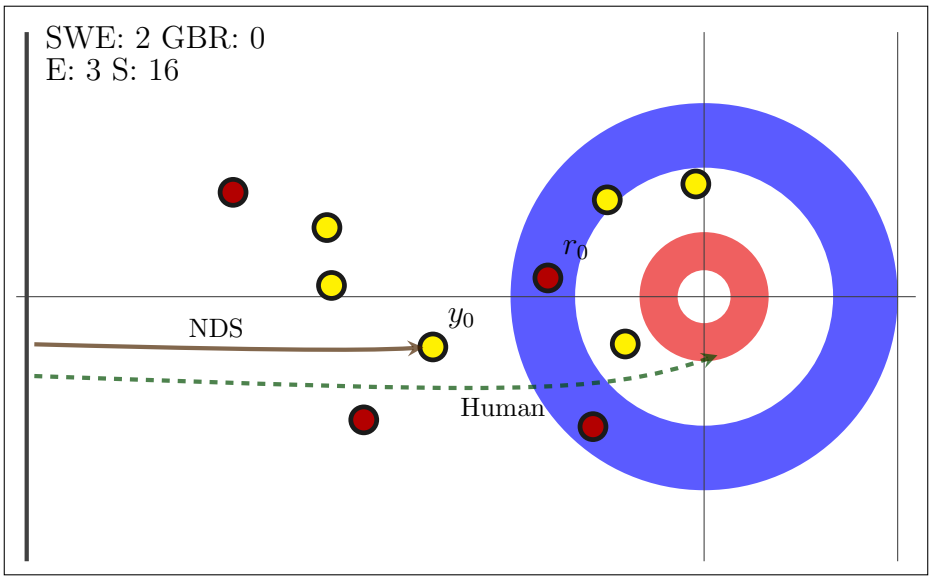


Figure 6.9: NDS v Humans Case 8

NDS Advantage: -15.48%

Expected Difference: -1.1%

Human Shot/Turn: Draw/Out

NDS Shot/Turn: Promotion/Out

Case 8 (Fig 6.9) illustrates a state where there are 5 rocks in play. It is the last shot of the 3rd end and red is down by 2 points. The yellow team currently has three rocks closest to the house and has a potential score of 3 points for the end. The red team has 2 rocks in the outer ring. To avoid losing points, the human team decided to draw to the button along the path shown. NDS however chose a shot that would raise y_0 into r_0 to promote it closer to the button. The human shot requires precision to navigate the shot rock between 4 other rocks to reach the house. This shot has an expected value of 5.94%. Humans managed to execute the shot to score 1 point. The shot chosen by NDS is only slightly less robust than the shot chosen by humans with an expected value of 5.83%.

6.2.4 Case 9

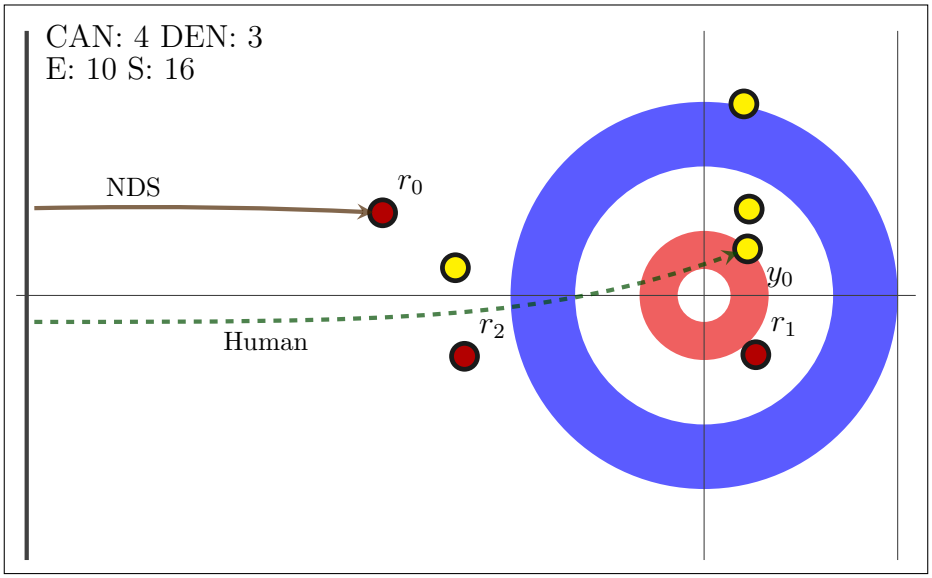


Figure 6.10: NDS v Humans Case 9

NDS Advantage: -15.31%

Expected Difference: 39.7%

Human Shot/Turn: Raise/Out

NDS Shot/Turn: Promotion/Out

In Case 9 (Fig 6.10) there are 3 yellow rocks and 1 red rock in the house. It is the final shot of the 10th end and the hammer team (red) is down by 1 point. The hammer team needs 1 point to tie and move on to extra ends or 2 points to win. Rock y_0 is in position to potentially score 1 point for the yellow team. The hammer team decided to tap y_0 so that r_1 or the shot rock (with the intent that both) is the closest rock to the button. As shown in the diagram, the shot's trajectory passes between two rocks in front of the house which demands a higher level of precision from the shooter for the shot to be executed with the intended outcome. The alternate shot consistently chosen by NDS is to raise r_0. Here the shot rock collides with r_0 and deflects it to the button. This shot is more robust - it has an expected value of 60.97% while the human team's intended shot has an expected value of 21.2%. Depending on the angle of the shot, r_0 can be knocked into the house or the shot rock can glance off of r_0 and move into the house itself.

From the examples that we examined we see that in most states where NDS performed better than humans, it was because they found an action with a higher expected value or the same shot as humans but humans were unlucky. However, in one case (Case 5), NDS selected a shot less robust than the humans did (nonetheless, humans failed to execute their intended shot). Usually, in cases where humans did better, they were successful in executing a difficult shot (actions with low expected WP). In these cases, NDS selected similar shots and in one case they found a better shot (this action was still a difficult shot to execute). In one state, NDS selected a slightly worse action but both actions selected by NDS and humans had very low expected values. Overall for the rest of the states, there are many cases where NDS performed as well as humans (the expected difference in WP was close to 0.)

6.3 Caveats

There were a substantial number of test states in which the curlers and NDS chose the same shot but the curlers mis-executed the shot and NDS, on average, did not. This highlights one of the limitations of this comparison — we were not able to repeatedly sample human execution of a shot to obtain its expected value. This works both ways, of course: just as the curlers may have been unlucky and mis-executed a shot they would make 80% of the time, there may have been shots where they got lucky and perfectly executed a shot they would make only 20% of the time.

A related limitation is that our execution model may be miscalibrated, making NDS more (or less) accurate at shot execution than the Olympic teams. The model was calibrated to Olympic-level curlers to the best of our ability. Likewise, our physics simulation is not a perfect replica of the true physics of curling. So even if our execution model was perfect and exactly the same shot was executed by NDS and by the curlers, the distribution of outcomes

might be different because our model fails to model the physics of varying ice conditions, physical differences between the rocks, etc.

A final limitation of this comparison to Olympic performance is that the data from the Olympics was logged by hand, so the positions of the rocks, as recorded in the logs and used by NDS in our study, might not be exactly the positions faced by the curlers during the Olympic games. Small differences in rock positions can substantially affect the distribution of a shot's outcomes. As with all the limitations we have discussed, this could work in favour of NDS but it could equally well work against it.

Chapter 7

Related Work

In this chapter, we look at a number of previous works that have been done in similar problem settings to that on which we focused in this thesis. We also briefly explore the approaches that each of these works took to address the problem.

7.1 Optimization Problems

7.1.1 Lipschitz Optimization

There are a slew of other approaches to global optimization that we have not applied to our application. One of the earlier approaches to global optimization is known as **Lipschitz optimization** [Danilin and Piyavskii, 1967; Shubert, 1972]. Algorithms in this family aim to find the optimal point of a function by assuming that the slopes of a function f in a d-dimensional space $\mathcal{X} \subseteq \mathbb{R}^d$ are bounded. The most common of these assumptions is the *Lipschitz continuity* [Eriksson *et al.*, 2013] of f given as

$$\forall \mathbf{x}_1, \mathbf{x}_2 \in \mathcal{X} : |f(\mathbf{x}_1) - f(\mathbf{x}_2)| \leq b\|\mathbf{x}_1 - \mathbf{x}_2\| \tag{7.1}$$

where b is known as the *Lipschitz constant*. Lipschitz continuity is the assumption that the rate at which a function can change is strictly bounded. Lipschitz optimization algorithms use this assumption to compute an upper bound over the function and iteratively choose points with the highest upper bound to evaluate. While algorithms exist that can find optimal points of function using this approach in tractable time [Danilin and Piyavskii, 1967; Shubert, 1972; Mladineo, 1986], Lipschitz optimization algorithms have one discouraging drawback: they work on the assumption that the Lipschitz constant is known. Practical considerations need to be taken when using these algorithms in real-world cases such as when the objective function to be optimized is a complex physics simulator (i.e., our application).

To relax this reliance on a known Lipschitz constant, the **DIviding RECTangles** or DIRECT algorithm seeks a global optimum by considering all possible values $0 < b < \infty$ of

the Lipschitz constant [Jones *et al.*, 1993]. DIRECT is an iterative algorithm where the function space is repeatedly partitioned into hyperrectangles and the function evaluated at the center points of these hyperrectangles. The observed values are then used to calculate upper bounds on the estimates of the function values at these points. The hyperrectangle that contains the point with the highest upper bound with respect to some value of b is chosen to be partitioned. Originally designed to be used for deterministic functions, a modification of the DIRECT algorithm exists for optimization of noisy functions [Deng and Ferris, 2007]. In this version, the center point of each hyperrectangle is not evaluated just once. They are instead evaluated multiple times. However, this results in an increased amount of overall function evaluations. To combat this, at each iteration the set of evaluations already performed is used to create a posterior distribution over function values to determine the appropriate number of repeated evaluations at any center point. This posterior distribution also provides further information about the variance at any point improving the accuracy of determining potentially optimal regions to partition and the final point selection.

7.1.2 Other Optimization Approaches

Other optimization techniques exist that do not explicitly look at the local structure of the function using Lipschitz continuity. **STAGE** [Boyan and Moore, 2000] is an algorithm that uses local optimization techniques as heuristics to guide the search for global optima. The performance of a local optimization technique is reliant on the sample points that are used to initialize the technique. If the random initial samples are evaluated in suboptimal regions of the function space, it is unlikely that the performance of the technique is better than suboptimal. In order to increase the likelihood of finding a good approximation of the optimal value, local optimization techniques are repeated a number of times with random restarts. STAGE seeks to find a global optimum of an objective by changing the trajectory of the local search at each restart. Let us assume, that $\dot{\mathbf{x}}$ is the initial "state" of a local search technique π. A state constitutes the starting point and other properties of a local search. Using the result of this search on the objective function f, STAGE builds another evaluation function $F^{\pi}(\dot{\mathbf{x}})$. This evaluation function is an approximation of the optimum value found by technique π from any given state $\dot{\mathbf{x}}$ in f. After each local search, $F^{\pi}(\dot{\mathbf{x}})$ is approximated using polynomial regression using the results from all local searches. The algorithm then optimizes $F^{\pi}(\dot{\mathbf{x}})$ to determine the starting state of the next local search. This algorithm has demonstrated good performance in several optimization domains including the design of VLSI circuitry [Boyan and Moore, 2000].

7.2 Resource Allocation Problems

7.2.1 Budgeted Learning

A problem similar in nature to our action selection setting is resource allocation for medical diagnosis, for example, how do we select which diagnostic tests are best for detecting cancers in a tissue balancing the cost of the tests and their effectiveness. This general problem is called **budgeted learning**. In the traditional learning setting, a learner is given data samples with which it learns an optimal model. Budgeted learning tasks [Madani *et al.*, 2004], are problem settings where a learner does not begin with any data about the samples but can retrieve data for a cost. The objective is then, given a set of resources (a budget) with which to collect data, to efficiently allocate the budget to retrieve the data needed about any instance to learn the optimal model. For instance, selecting the best subset of diagnostic tests among all possible tests to diagnose cancers of a type of tissue. A variant of this task called **active model selection** constitutes determining the best model from a given set of models that most accurately evaluates the given data (i.e., which individual test most accurately diagnoses cancer in a set of tissues). For a simplified example of the problem, we can imagine a learner is given a set of indistinguishable coins with varying unknown head probabilities. The learner can sequentially flip coins to observe the outcomes. After a specified number of flips, the learner has to decide which coin has the highest expectation of landing heads. The budgeted active model selection task is congruent to the problem of finding the best action given a set of actions, which is what we focus on. Madani et al. [2004] investigate this problem and analyze the performance of various techniques (i.e., round-robin sampling, Gittins) to complete the task. They also show that the simple biased robin algorithm is more effective than other algorithms in practice at addressing budgeted active learning problems where the costs for sampling every point is identical. The biased robin works similarly to round robin: each action is sampled iteratively. However unlike round-robin, in the biased robin, whenever an action yields a desired outcome it is repeatedly sampled until it yields an undesired outcome before sampling the next action (i.e., if a coin flip results in a heads, keep flipping that particular coin till it lands tails.)

7.2.2 Pure Exploration in Multi-Armed Bandits

Another way we can look at the resource allocation problem from above is as a **multi-armed bandit problem with pure exploration** [Bubeck *et al.*, 2009a; Audibert and Bubeck, 2010]. In the standard multi-armed bandit setting we described earlier (§3.1), the objective is to find the sequence of actions, where each action is an arm of the bandit, that maximize the cumulative reward. In the pure exploration problem each action has an associated cost for sampling. The goal is then to identify the single best action with efficient use of the

budget for sampling actions. For instance, in real world settings, each action may require a considerable amount of CPU time and we want to use the CPU time effectively when improving our expectation of the actions. Audibert and Bubeck [2010] provide a simple modification to the UCB formula (Eq. 3.1) to adapt it to the exploration setting. The result is the UCB-E action selection strategy where at each stage N of the algorithm an agent would play an arm i among a set of k arms according to

$$\underset{i \in \{1,2,\ldots,k\}}{\arg\max} \left[\bar{r}_i + \sqrt{\frac{a}{N}} \right] \tag{7.2}$$

where $\bar{r}_i$ is the average reward observed by arm i after N samples and a is tunable parameter. The authors show that in practice, this algorithm works effectively and they prove that it has at most logarithmic cumulative *regret* (the cost for choosing a suboptimal action over the optimal action at each stage). However, both the pure exploration bandit algorithm and active learning techniques suffer from the same limitations preventing them from being useful to our application: they only work with a finite set of actions while our problem setting consists of an infinite set of possible actions. Furthermore, when these techniques are given a finite set of candidate actions, they are limited to those and cannot identify new actions.

7.3 Curling and Billiards

Previous work has been done on developing strategies for curling in [Yamamoto *et al.*, 2015]. Work in billiards, a game which is similar to curling with continuous actions and states along with stochastic outcomes, has received some attention in the literature [Archibald *et al.*, 2009; Smith, 2007; Smith, 2006]. However, in all of this work, the authors use domain knowledge to create a finite set of discrete actions. This makes it possible for search to be employed without addressing the challenges of optimization in continuous settings. This work sought to forego domain knowledge as much as possible, in favor of a more domain-independent approach.

Chapter 8

Conclusion

8.1 Summary

This thesis studied optimal decision making for selecting actions in settings where the actions have stochastic outcomes. Curling was presented as an ideal domain for this study. This thesis described the research challenges that arise in this particular setting. This work then showed how this setting could be reduced to a low-dimensional optimization problem with continuous, stochastic actions. An algorithm based on Delaunay triangulation was adapted to be used for this setting and compared to existing methods for optimization. Then finally this research compared strategies developed by an artificial intelligence agent against those of Olympic-level humans in the domain of curling.

The optimization problem modelled reflects finding the last shot of an end in curling called the hammer shot. This was the focal point of this thesis and various existing approaches were surveyed for solving this problem. The new algorithm designed, called Delaunay Sampling, was shown to perform well, significantly better than existing algorithms, finding good robust actions without any domain-specific knowledge.

All the algorithms studied and NDS were tested on actual hammer shot situations that transpired during 2010 men's and women's Olympic curling games. Data logged from the games provided a set of 515 hammer states on which the algorithms were tested. The actions examined by the algorithms were evaluated by a curling physics simulator calibrated using over 30000 games played between 2011 and 2014. These tests showed that DS and the normalized version of DS performed better than existing algorithms. One of the main objectives of this research was to explore the possibility of using AI agents to beat human strategies for curling (or stochastic action optimization in general). Testing using Olympic data also allowed strategies selected by DS to be compared to those of humans and this work has shown that our algorithm, with a few caveats, does outperform human strategies.

This research has produced several important results. One of the key results is that even in a more simplified setting, decision making for selecting actions when there is uncertainty

in the action outcomes is a non-trivial task. Although our research has shown some algorithms, notably DS and normalized DS, have exhibited good performance in our setting, it is difficult to assume that comparable performance can be had in other settings or whether they can even be scaled to problems in higher dimensions. Another result from the experimental evaluations has shown that even when ignoring domain knowledge in our setting, AI algorithms can be used to improve human decision making strategies and incorporating domain knowledge can help to increase the efficiency of these algorithms. It follows that this is also an indication that there are limitations to humans' abilities to judge the expectation of stochastic actions.

8.2 Limitations

One of the key limitations stems from the use of a physics simulator for our experimental evaluations. Our simulator was based on the physics of real-world curling which is not comprehensively understood therefore our simulator is not a perfect replica of the true physics of curling. The execution model of the simulator adds stochasticity to actions by attempting to accurately model the execution errors due to human inaccuracy and unpredictable ice conditions. Although calibrated using actual Olympic games, this model is not perfect. While these do not affect the results gleaned from the comparisons between the various algorithms that has been tested in this thesis, it does however, affect the certainty of our comparisons of AI algorithms with the performance of humans. A lack of repeated samples of actions chosen by humans also add some limitations to our analysis.

Another limitation of this research is due to where stochasticity lies in our particular optimization setting. In conventional optimization problems, stochasticity is introduced as a Gaussian noise added to function evaluations. However, in ours, the noise is added to parameters of the function instead. While this does have an equivalent noise model on function evaluations, it is impossible to compute this model using the former during action optimization. This adds a challenge when investigating approaches in our setting that explicitly require knowledge of noise on the function observations.

Simplification of the curling problem introduces some limitations. The first simplification is how turns are modelled for shots in curling. Instead of representing a turn using a continuous variable with a value for the angular velocity on the rock being shot, only the direction of the turn is taken into consideration. While the angular velocity on a rock has but a small effect on the trajectory of a shot, it is still a variable that adds uncertainty to shot execution. In our simulator, this is ignored. The second simplification is that sweeping is not explicitly modelled as a variable for taking a shot. Since for the most part, sweeping is used to correct for execution error, it is incorporated into the execution model of simulator. However, in certain situations in curling, there are actions that can only be taken using

sweeping. By not explicitly modelling sweeping separately, we remove any possibility that these actions are recognized by the AI agents.

8.3 Final Word

Optimal decision making in situations where the actions have uncertainty in their outcomes is a difficult task. In order to find good solutions to problems of this nature, we must develop algorithms that are robust to complex, unknown noise models. This thesis has shown that curling is a real-world scenario where this non-trivial setting exists and has explored various existing approaches and a novel approach to solving this problem with reasonable success. Finally, while the setting we have tested our approaches on was more simplistic than most real-world situations, they show promise in being applied to various larger problems.

Bibliography

[Archibald *et al.*, 2009] Christopher Archibald, Alon Altman, and Yoav Shoham. Analysis of a winning computational billiards player. In *IJCAI*, volume 9, pages 1377–1382. Citeseer, 2009.

[Audibert and Bubeck, 2010] Jean-Yves Audibert and Sébastien Bubeck. Best arm identification in multi-armed bandits. In *COLT-23th Conference on Learning Theory-2010*, pages 13–p, 2010.

[Auer *et al.*, 2002] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

[Auger and Hansen, 2009] Anne Auger and Nikolaus Hansen. Benchmarking the (1+1)-CMA-ES on the BBOB-2009 function testbed. In *Genetic and Evolutionary Computation Conference, GECCO 2009, Proceedings, Montreal, Québec, Canada, July 8-12, 2009, Companion Material*, pages 2459–2466, 2009.

[Bäck *et al.*, 1991] Thomas Bäck, Frank Hoffmeister, and Hans-Paul Schwefel. A survey of evolution strategies. In *Proceedings of the 4th International Conference on Genetic Algorithms, San Diego, CA, USA, July 1991*, pages 2–9, 1991.

[Boyan and Moore, 2000] Justin Boyan and Andrew W Moore. Learning evaluation functions to improve optimization by local search. *Journal of Machine Learning Research*, 1(Nov):77–112, 2000.

[Boyd and Vandenberghe, 2004] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[Browne *et al.*, 2012] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, Simon Colton, et al. A survey of monte carlo tree search methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43, 2012.

[Bubeck *et al.*, 2009a] Sébastien Bubeck, Rémi Munos, and Gilles Stoltz. Pure exploration in multi-armed bandits problems. In *International conference on Algorithmic learning theory*, pages 23–37. Springer, 2009.

[Bubeck *et al.*, 2009b] Sébastien Bubeck, Gilles Stoltz, Csaba Szepesvári, and Rémi Munos. Online optimization in x-armed bandits. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *Advances in Neural Information Processing Systems 21*, pages 201–208. Curran Associates, Inc., 2009.

[Danilin and Piyavskii, 1967] Yu M Danilin and SA Piyavskii. An algorithm for finding the absolute minimum. *Theory of Optimal decisions*, 2:25–37, 1967.

[De Berg *et al.*, 2000] Mark De Berg, Marc Van Kreveld, Mark Overmars, and Otfried Cheong Schwarzkopf. Computational geometry. In *Computational geometry*, pages 1–17. Springer, 2000.

[Deng and Ferris, 2007] Geng Deng and Michael C Ferris. Extension of the direct optimization algorithm for noisy functions. In *2007 Winter Simulation Conference*, pages 497–504. IEEE, 2007.

[Denny, 1998] Mark Denny. Curling rock dynamics. *Canadian journal of physics*, 76(4):295–304, 1998.

[Eriksson *et al.*, 2013] Kenneth Eriksson, Donald Estep, and Claes Johnson. *Applied Mathematics: Body and Soul: Volume 1: Derivatives and Geometry in IR3*. Springer Science & Business Media, 2013.

[Hansen and Ostermeier, 1996] Nikolaus Hansen and Andreas Ostermeier. Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation. In *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on*, pages 312–317. IEEE, 1996.

[Hansen, 2016] Nikolaus Hansen. The CMA evolution strategy: A tutorial. *arXiv:1604.00772*, 2016.

[Jensen and Shegelski, 2004] ET Jensen and Mark RA Shegelski. The motion of curling rocks: experimental investigation and semi-phenomenological description. *Canadian journal of physics*, 82(10):791–809, 2004.

[Jones *et al.*, 1993] Donald R Jones, Cary D Perttunen, and Bruce E Stuckman. Lipschitzian optimization without the lipschitz constant. *Journal of Optimization Theory and Applications*, 79(1):157–181, 1993.

[Kennedy, 2011] James Kennedy. Particle swarm optimization. In *Encyclopedia of machine learning*, pages 760–766. Springer, 2011.

[Kleinberg, 2004] Robert D Kleinberg. Nearly tight bounds for the continuum-armed bandit problem. In *Advances in Neural Information Processing Systems*, pages 697–704, 2004.

[Kocsis and Szepesvári, 2006] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Machine Learning: ECML 2006*, pages 282–293. Springer, 2006.

[Lee and Schachter, 1980] Der-Tsai Lee and Bruce J Schachter. Two algorithms for constructing a delaunay triangulation. *International Journal of Computer & Information Sciences*, 9(3):219–242, 1980.

[Lizotte *et al.*, 2007] Daniel J Lizotte, Tao Wang, Michael H Bowling, and Dale Schuurmans. Automatic gait optimization with gaussian process regression. In *IJCAI*, volume 7, pages 944–949, 2007.

[Lozowski *et al.*, 2015] Edward P Lozowski, Krzysztof Szilder, Sean Maw, Alexis Morris, Louis Poirier, Berni Kleiner, et al. Towards a first principles model of curling ice friction and curling stone dynamics. In *The Twenty-fifth International Offshore and Polar Engineering Conference*. International Society of Offshore and Polar Engineers, 2015.

[Madani *et al.*, 2004] Omid Madani, Daniel J Lizotte, and Russell Greiner. Active model selection. In *Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pages 357–365. AUAI Press, 2004.

[Mladineo, 1986] Regina Hunter Mladineo. An algorithm for finding the global maximum of a multimodal, multivariate function. *Mathematical Programming*, 34(2):188–200, 1986.

[Nadaraya, 1964] Elizbar A Nadaraya. On estimating regression. *Theory of Probability & Its Applications*, 9(1):141–142, 1964.

[Nyberg *et al.*, 2012] Harald Nyberg, Sture Hogmark, and Staffan Jacobson. Calculated trajectories of curling stones sliding under asymmetrical friction. In *Nordtrib 2012, 15th Nordic Symposium on Tribology, 12-15 June 2012, Trondheim, Norway*, 2012.

[Nyberg *et al.*, 2013] Harald Nyberg, Sara Alfredson, Sture Hogmark, and Staffan Jacobson. The asymmetrical friction mechanism that puts the curl in the curling stone. *Wear*, 301(1):583–589, 2013.

[Rasmussen, 2006] Carl Edward Rasmussen. Gaussian processes for machine learning. 2006.

[Shewchuk, 2002] Jonathan Richard Shewchuk. Delaunay refinement algorithms for triangular mesh generation. *Computational geometry*, 22(1):21–74, 2002.

[Shi and Eberhart, 1998] Yuhui Shi and Russell Eberhart. A modified particle swarm optimizer. In *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on*, pages 69–73. IEEE, 1998.

[Shubert, 1972] Bruno O Shubert. A sequential method seeking the global maximum of a function. *SIAM Journal on Numerical Analysis*, 9(3):379–388, 1972.

[Smith, 2006] Michael Smith. Running the table: An ai for computer billiards. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, volume 21, page 994. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2006.

[Smith, 2007] Michael Smith. Pickpocket: A computer billiards shark. *Artificial Intelligence*, 171(16):1069–1091, 2007.

[Snoek *et al.*, 2012] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[Surovik and Scheeres, 2015] David Allen Surovik and Daniel J Scheeres. Heuristic search and receding-horizon planning in complex spacecraft orbit domains. In *Eighth Annual Symposium on Combinatorial Search*, 2015.

[Watson, 1964] Geoffrey S Watson. Smooth regression analysis. *Sankhyā: The Indian Journal of Statistics, Series A*, pages 359–372, 1964.

[Yamamoto *et al.*, 2015] Masahito Yamamoto, Shu Kato, and Hiroyuki Iizuka. Digital curling strategy based on game tree search. In *Computational Intelligence and Games (CIG), 2015 IEEE Conference on*, pages 474–480. IEEE, 2015.

[Yee *et al.*, 2016] Timothy Yee, Viliam Lisy, and Michael Bowling. Monte carlo tree search in continuous action spaces with execution uncertainty. In *IJCAI*, 2016.