Learning to Learn From Simulation

Using simulations to expedite learning on robots

Akshara Rai
Robotics Institute, Carnegie Mellon University

Thesis Committee
Christopher G. Atkeson, Chair
Hartmut Geyer
Oliver Kroemer
Stefan Schaal, MPI Tübingen and USC

Submitted on October, 2017 for partial fulfillment of the Degree of Doctor of Philosophy

Abstract

Robot controllers, including locomotion controllers, often consist of expert-designed heuristics. These heuristics can be hard to tune, particularly in higher dimensions. It is typical to use simulation to tune or learn these parameters and test on hardware. However, controllers learned in simulation often don't transfer to hardware due to model mismatch. This necessitates controller optimization directly on hardware. Experiments on walking robots can be expensive, due to the time involved and fear of damage to the robot. This has led to a recent interest in adapting data-efficient learning techniques to robotics. One popular method is Bayesian Optimization, a sample-efficient black-box optimization scheme. But the performance of Bayesian Optimization typically degrades in problems with higher dimensionality, including dimensionality of the scale seen in bipedal locomotion. We aim to overcome this problem by incorporating prior knowledge to reduce the number of dimensions in a meaningful way, with a focus on bipedal locomotion. We propose two ways of doing this, hand-designed features based on knowledge of human walking and using neural networks to extract this information automatically. Our hand-designed features project the initial controller space to a 1-dimensional space, and show promise in simulation and on hardware. On the other hand, the automatically learned features can be of varying dimensions, and also lead to improvement on traditional Bayesian Optimization methods and perform competitively to our hand-designed features in simulation. Our hardware experiments are evaluated on the ATRIAS robot, while simulation experiments are done for two robots - ATRIAS and a 7-link biped model. Our results show that these feature transforms capture important aspects of walking and accelerate learning on hardware and perturbed simulation, as compared to traditional Bayesian Optimization and other optimization methods.

Acknowledgements

The work in this proposal was done in close association with Rika Antonova. I thank her for her continuous help, motivation and involvement in this work.

Contents

CHAPTER 1

Introduction

Robot controllers, and locomotion controllers in particular, can consist of many expert-designed heuristics, for example feedback control of the Center of Mass (CoM) position and velocity, feedback control of robot joints, and designing reference trajectories and desired position. State of the art work in walking robots featuring such heuristics includes [1], [2] and [3]. These heuristics consist of sets of inter-dependent parameters, which can be hard to tune, especially in higher dimensions. In such cases, it is an art, rather than a science, to tune these parameters. This complexity motivates methods for learning parameters automatically. A simple approach is to learn in simulation and deploy on hardware. However, due to commonly encountered differences between simulation and hardware, such as modelling errors, performance in simulation often does not transfer to hardware. On the other hand, directly learning on hardware can require a prohibitively large number of experiments, making it nearly impossible to learn these controllers using traditional methods. This has led to a surge in interest in data-efficient learning techniques for robotics.

One popular data-efficient method for learning controller parameters is Bayesian Optimization (BO). BO is a sample-efficient gradient-free black-box optimization method that has been applied to a wide range of robotics problems. For example, [4], [5], [6] try to learn parameters directly on hardware using BO. However, the performance of BO degrades in high dimensions (see [7] for a related discussion), even for dimensionalities commonly encountered in locomotion controllers. We aim to overcome this problem by incorporating domain knowledge into BO.

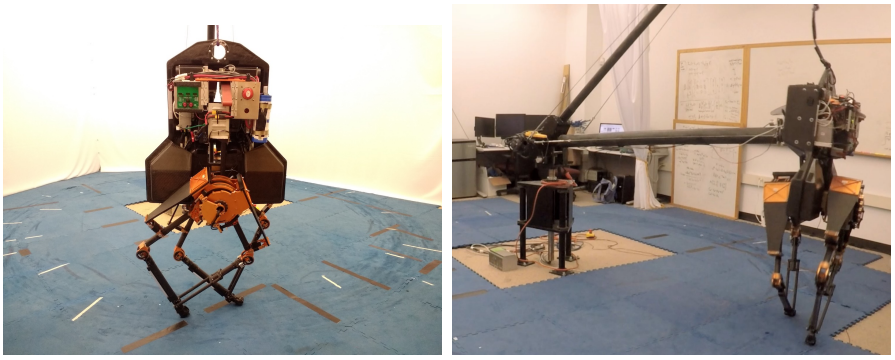


Figure 1.1: Our testbed is CMU's ATRIAS robot.

One way of incorporating domain knowledge could be to learn approximate models of the system from simulation, before moving to hardware. The idea of using simulation performance to speed up optimization on hardware has been explored before. A common approach is to learn controllers in simulation, and use this as a starting point on hardware. A domain expert would then typically have to fine-tune parameters on hardware. [8] learns parameters using a collection of simulations that help account for model uncertainty. [9] iteratively learns both the model and controller parameters using the differences between simulated behaviors and observed hardware experiments. [5] uses evaluations from simulation as a noisy prior for the optimization on hardware. [6] pre-selects high performing controllers from simulation and search among them on hardware.

1.1 Adding prior knowledge to optimization

We propose incorporating domain knowledge by building informed feature transforms that project the higher dimensional controller parameter space to an easy to optimize lower dimensional space of features. These can be expert-designed or learned from data, as described below:

1.1.1 Expert-designed feature transforms

We use human-walking inspired features to develop a feature transform for bipedal walking that roughly groups controllers based on their walking features in short simulations. The objective of such a transformation is that some behavioural cues have a higher probability of transferring between simulation and hardware than exact cost or controller performance. For example, a controller that falls instantly

in simulation by tilting its torso, would have a small chance of walking on hardware. On the other hand, a controller that walks efficiently in simulation might have a higher chance of walking on hardware, albeit with a different cost. Hence, it seems reasonable to roughly group parameters based on the behaviour they produce in simulation, and use this as a distance metric to distinguish between them. Despite being motivated by human walking features, these features also generalize to other robot morphologies and controllers, like the ATRIAS biped.

1.1.2 Data-driven feature transforms

Using extensive expert knowledge, as suggested above, has several advantages. It helps us learn and design features fast as well as provides transparency as to what the learning approach is prioritizing. However, it does raise concerns about problems for which such knowledge might not be available or as easily implementable. With this in mind, **we also develop a method to construct an informed metric automatically, without relying heavily on domain experts**. We propose to learn a distance metric with a neural network, utilizing data obtained from a high-fidelity simulator. This involves first running short simulations of a locomotion controller on a large grid of control parameters and recording the behavior of each set of parameters. The neural network then learns a mapping between input controller parameters and simulation output/behavior. We propose two ways of defining the target to be learned by the network. The first approach is based on the cost function that is to be optimized with BO on hardware, or a perturbed simulator. The second is cost-agnostic: learning to reconstruct a summary of robot trajectories obtained from simulation. This provides a useful re-parameterization: controller parameters that produce similar walking trajectory summaries are closer in this re-parameterized space.

1.2 Accounting for mismatch between hardware and simulation

An important question that arises when using simulation to guide hardware searches is: how can we learn and take into account the differences between simulation and hardware? For example [10] adapts the target task based on

the mismatch between the expected and actual performance. [5] develops an automatic way of transitioning between simulation and hardware based on data collected so far. **We propose to learn a mismatch-map that represents the mismatch between behaviour encountered in simulation and hardware.** This allows us to build a dynamic error map over our parameter space, based on data. We start by trusting all simulation points with predicted mismatch of 0, and as we sample points on hardware, we update our estimate of error on each simulation sample. The advantage of this is that we can have different error values for different regions of parameter space, as some parts of the parameter space might transfer well between hardware and simulation, while others might not. Using this trust-map, we can further improve sample-efficiency in perturbed simulation experiments. This is a promising result that needs to be tested on hardware.

1.3 Summary of completed goals and expected contributions

In work done so far, we present evaluations of our proposed methods on the ATRIAS biped robot (Figure 1.1), ATRIAS simulation, and a 7-link biped simulations. We evaluate our feature transforms on three different controllers of increasing dimensionality. We start with a 5 dimensional feedback-based reactively stepping controller, and increase its dimensionality to 9. Then we take a 50-dimensional highly non-linear neuromuscular controller and optimize its parameters. We successfully optimize parameters for a 5-dimensional and 9-dimensional controller on the ATRIAS hardware in less than 10 trials, which proves to be challenging for traditional BO. In simulation, we find the neural network based features and the hand-designed features to be very effective at optimizing all three controllers. All our experiments so far are either on a boom (in the case of ATRIAS) on in a 2 dimensional simulation.

Our results show that this feature transform extracts useful information from simulations, and leads to an effective transfer of knowledge to hardware. This motivates future work for using our approach on hardware for the 50-dimensional controller. Another important direction to explore is a walking controller for a 3-dimensional robot. This would need a feature transform with features about

3-dimensional walking.

While, Bayesian Optimization benefits from feature transforms and informed distance metrics, we would also like to explore other methods that can improve with knowledge from simulation. For example, we would like to explore using information about the cost function landscape from simulation to learn natural-gradients, used for example in, [11], instead of approximating from hardware data. Another approach that could benefit is developing a memory of learning rates, as described in [12]. The memory of learning rates can be learned from simulation and used to predict learning rates on hardware. These suggestions assume that the function landscape in simulation and hardware match, which might not always be the case. In such situations, it would be useful to use the mismatch-map to learn afresh where the dynamics in simulation and hardware don't match.

The proposed goals and contributions of this thesis are as follows:

1. Develop a feature transforms for 3-dimensional bipedal locomotion and test them on controllers in simulation and hardware
2. Develop a principled way of propagating mismatch between simulation and hardware by building mismatch maps, which can also be used by other learning methods.
3. Explore other learning approaches in literature that could benefit from prior knowledge from simulation.

CHAPTER 2

Background

In this Chapter, we cover some literature on Bayesian Optimization for robotics, using simulation information to learn faster on hardware and learning the mismatch between the dynamics in simulation and hardware. We also give a short introduction of other policy search methods in literature, which might benefit from simulation information.

2.1 Dynamic Programming and Reinforcement Learning

In Reinforcement Learning, a controller (agent) interacts with a process (environment) through a action, which changes the state of the process and generates a reward (feedback from the environment) (Figure 2.1). The controller receives the new state and the reward, and the whole process repeats. The behaviour of the controller is determined by a policy that maps the process state to an action. The process follows a stochastic or deterministic dynamics model which determines how the state changes as a result of actions.

The goal is to find an optimal policy that maximizes the (expected) return, consisting of the (expected) cumulative reward over the course of interaction, called the value function. This framework can be used to address problems from a variety of fields, e.g., automatic control, artificial intelligence, operations research, and economics.

We assume a discrete-time process and denote the current time step by k.

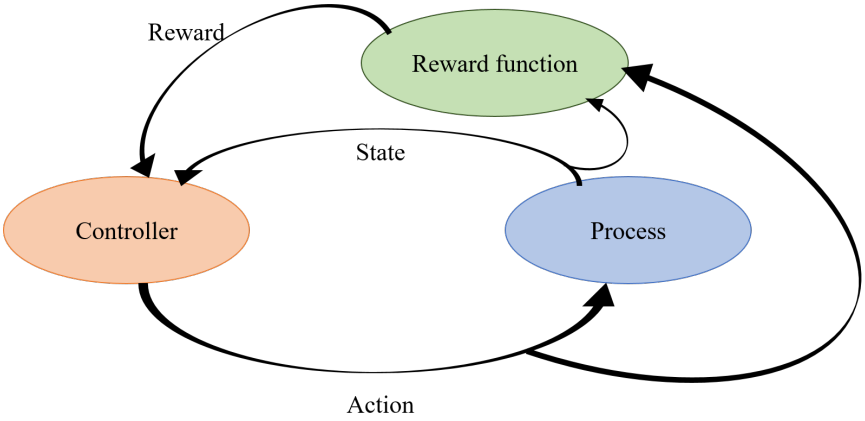


Figure 2.1: The interaction between the controller, process and reward function in dynamic programming and reinforcement learning.

The transition model p is given by a probability distribution conditioned on the current state and current action. We assume a Markov decision process, where the next state only depends on the previous state and not states before that.

$$x_{k+1} \sim p(x_{k+1}|x_k, u_k) \tag{2.1}$$

where $u_k \in \mathbb{R}^M$ denotes the current action, and $x_k, x_{k+1} \in \mathbb{R}^N$ denote the current and the next state respectively. We furthermore assume that actions are generated by a policy

$$u_k \sim \pi_\theta(u_k|x_k) \tag{2.2}$$

which is modeled as a probability distribution conditioned on the current state. This lets us naturally include exploratory actions in the policy. The policy is parameterized by some policy parameters $\theta \in \mathbb{R}^K$. Over one episode, the sequence of states and actions forms a trajectory, denoted by $\xi = [x_{0:H}, u_{0:H}]$ where H denotes the horizon, which can be infinite. At each instant of time, the learning system receives a reward (or cost for minimization) denoted by $r(x_k, u_k) \in \mathbb{R}$.

The objective function of policy search algorithms is the expected return of a trajectory, as a function of the policy parameters θ. We assume no-discounting in all our formulations, but discounting factors that weigh future costs lower than the current cost could be added.

$$J(\theta) = \mathbb{E}[\sum_{k=0}^{H} r_k] \tag{2.3}$$

In general, for each considered policy π_θ , a state-value function $V_\pi(x,k)$, and a state-action value function (Q-function) $Q_\pi(x,u,k)$ can be defined as

$$V_\pi(x,k) = \mathbb{E}[\sum_{i=k}^{H} r_i | x_k = x] \tag{2.4}$$

$$Q_\pi(x,u,k) = \mathbb{E}[\sum_{i=k}^{H} r_i | x_k = x, u_k = u] \tag{2.5}$$

Note, that we can define the expected return in terms of the state-value function by

$$J(\theta) = \int_x p(x_0) V_\pi(x_0, 0) dx_0 \tag{2.6}$$

where $p(x_0)$ is the probability of x_0 being the initial state.

In the presence of a model, one of the widely used optimal control methods in robotics are dynamic programming algorithms, which can be considered a part of reinforcement learning. Popular dynamic programming methods include value iteration or policy iteration, which are iterative methods that are guaranteed to converge to the global optimum [13]. However, dynamic programming methods do not scale to higher dimensional problems, leading to a "curse of dimensionality". For example, value iteration converges to the true value function in polynomial time in the number of states and actions [14]. This raises concerns about generalizing to higher dimensions, and especially for continuous action and states. To deal with this approximate dynamic programming approaches have been suggested, for example, [15] randomly samples over actions over a continuous space to avoid discretizing actions. [16]breaks down the problem of controlling a 5-link biped into 4 simpler, lower-dimensional problems and solves each separately.

Alternatively, in the absence of a model, function approximators can be used to learn the value function and Q-function, resulting in feasible methods even for higher dimensions and continuous action states, example SARSA and Q-learning [17]. However, this can result in biases in the estimate of the value function, especially with limited data. Unfortunately, in the general case, since these methods depend on their previous estimate of the target to generate the next solutions, they have an inherent bias based on their initial estimate. Coming up with an appropriate function approximator is a non-trivial problem and can involve using prior knowledge that might not be available, especially for model-

free RL.

Policy search methods avoid the problem of approximating the value function or the Q-function by directly searching for an optimal control policy using optimization. The optimization criterion is the expected return – a weighted average of rewards from different initial conditions, resulting in a policy the maximizes reward from all initial states. For a general problem, the optimization criterion may be a non-differentiable function with multiple local optima. This means that global, gradient-free optimization techniques are more appropriate than local, gradient-based techniques. Some successful techniques include covariance-matrix adaptation [18], and cross-entropy optimization [19].

2.2 Reinforcement learning for bipedal robots

Reinforcement learning (RL) has been widely applied to robotics problems, including simulations and hardware of manipulation, locomotion, perception problems. The widespread use of RL makes it hard to review all works in robotics, hence we will focus largely on RL for bipedal locomotion.

Some of the applications of RL on bipedal robots include, [20], [21], [22] and [23]. [20] started with a passively stable 3d biped and added ankle actuation. With simplifications, like decoupling the roll and sagittal degrees of freedom, they approximate a value function and the control policy using linear function approximators. This system was shown to learn to walk in a very short time. [21] modulates the via-points of reference trajectories of hip and knee joints of a 5-link biped, by learning a value function, control policy and a Poincaré section of the robot. The policy-gradient method, aided by the Poincaré model, found robust walking solutions in 100 trials on a moderately-complex robot hardware. [22] uses a central-pattern generator (CPG) based controller, whose parameters are tuned online using an actor critic method. However, the robot had to be externally supported with a "walker" to aid stability. [23] uses a rhythmic dynamic movement primitive to learn an initial gait from locomotion and adapt the frequency of oscillation using RL on a 5-link biped.

Another popular approach employing RL or DP in robotics is using it to learn the Center of Mass (CoM) trajectory, and follow it using optimization based Inverse Dynamics. [24] use RL to reduce the energy consumption by learning a

variable-height CoM trajectory, with a evolving policy parametrization of their policy. The learned CoM trajectory exploits the passive dynamics of the COMAN robot and reduces the overall energy of walking better than the fixed policy parametrizations. [1] use an approximation of dynamic programming, called Differential Dynamic Programming to optimize the CoM trajectory using an inverted pendulum model with ankle torques. Combining this with reactive foot placement in [25] results in very a robust walking controller on the ATLAS humanoid robot. An application of approximate dynamic programming for whole-body control of bipedal robots was described in [16]. [16] breaks down the problem of controlling a 5-link biped into 4 simpler, lower-dimensional problems - sagittal stance, sagittal swing, coronal stance and yaw control. The coupled dynamics of the simpler models are considered as disturbances. The control policy generated by dynamic programming is robust to perturbations in forward and lateral directions, as well as generalize to different speeds. This was also implemented on the Sarcos humanoid and shown to reject small unknown disturbances while walking.

2.3 Bayesian Optimization for Robotics

In this section, we will provide a basic introduction of Bayesian optimization and Gaussian processes. This will be followed by a survey of BO for policy search in robotics and in particular, using BO with trajectory information.

2.3.1 Bayesian Optimization and Gaussian Processes

Bayesian Optimization (BO) is a framework for sample-efficient black-box and gradient free global search. It was introduced by [26] and originally was referred to as Efficient Global Optimization. Recent tutorials [27] and [28] provide a comprehensive overview. Here we will summarize the basic background for Bayesian Optimization to establish notation that will be used in the following sections.

The goal of BO is to find $\boldsymbol{x}^*$ that optimizes an objective function $f(\boldsymbol{x})$, while executing as few evaluations of f as possible. The class of functions is not restricted to continuous or smooth functions and hence BO can optimize non-differentiable functions. Little or no a-priori knowledge about f is necessary, but if available, prior knowledge can be incorporated (as we will discuss shortly). The optimization starts with a prior roughly capturing the prior uncertainty over the

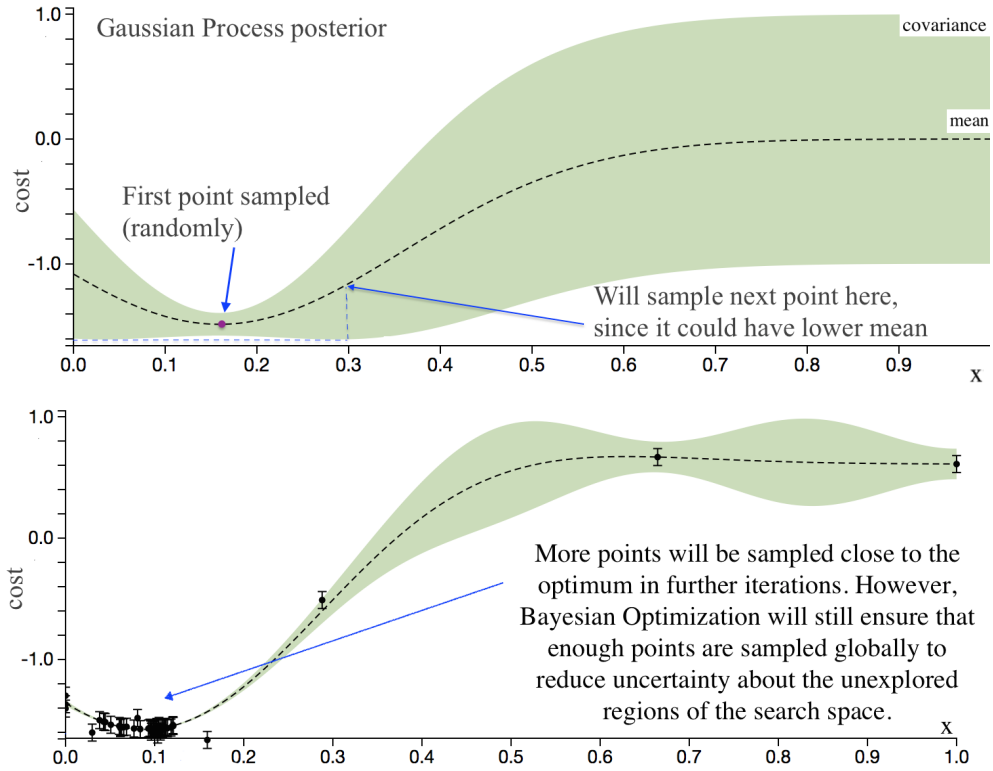


Figure 2.2: Bayesian Optimization posterior for a 1D function. Acquisition function computes the location of points to sample, taking into account both estimated mean and variance (uncertainty).

value of $f(\boldsymbol{x})$ for each $\boldsymbol{x}$ in the domain. If no prior knowledge is available, this prior can be uninformed and updated with data. As points are sampled, these are included into the estimate of $f(\boldsymbol{x})$, giving an approximate model based on data seen so far. An auxiliary optimization function, called the *acquisition function*, is used to sequentially select points $\boldsymbol{x}_n$, based on the current model of $f(\boldsymbol{x})$. $f(\boldsymbol{x}_n)$ is evaluated and added to the sampled points. The aim of the acquisition function is to automatically balance exploration and exploitation: select points for which the posterior estimate of the objective f is promising, while also decreasing the uncertainty about f. An example of BO is shown in Figure 2.2.

Expected Improvement (EI) function is commonly used as an acquisition function. EI selects $\boldsymbol{x}$ to maximize expected improvement over the value of the best

result obtained so far [29]. For minimization problems:

$$EI(\boldsymbol{x}) = \begin{cases} \big[f(\boldsymbol{x}^-) - \mu(\boldsymbol{x})\big]\Phi(Z) + \sigma(\boldsymbol{x})\phi(Z) & \text{if } \sigma(\boldsymbol{x}) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.7}$$

where $\boldsymbol{x}^- = \arg\min_{x_i \in x_{1:t}} f(\boldsymbol{x}_i), \quad Z = (f(\boldsymbol{x}^-) - \mu(\boldsymbol{x}))/\sigma(\boldsymbol{x})$.

$\boldsymbol{x}_i$ are points for which objective f has been evaluated so far, μ, σ are posterior mean, variance; ϕ, Φ are PDF and CDF of standard normal distribution. In all our experiments, we use EI as the acquisition function, as it doesn't have any hyperparameters to tune.

Another frequently used acquisition function is Upper Confidence Bound (UCB) or, for minimization problems, Lower Confidence Bound (LCB) [30]. LCB balances the mean μ and variance σ of the posterior by selecting points which minimize

$$LCB(\boldsymbol{x}) = \mu(\boldsymbol{x}) - \alpha\sigma(\boldsymbol{x}) \tag{2.8}$$

While there can be many ways of modelling $f(\boldsymbol{x})$, we describe a specific non-parametric way of describing $f(\boldsymbol{x})$ using a Gaussian Process (GP). A Gaussian Process is defined by a set of random variables that are jointly Gaussian. It can be completely specified by its mean function and covariance function.

$$f(\boldsymbol{x}) \sim \mathcal{GP}(\mu(\boldsymbol{x}), k(\boldsymbol{x}_i, \boldsymbol{x}_j)), \tag{2.9}$$

with mean function $\mu(\boldsymbol{x})$ and kernel $k(\boldsymbol{x}_i, \boldsymbol{x}_j))$.

$$\mu(\boldsymbol{x}) = \mathbb{E}[\boldsymbol{x}] \tag{2.10}$$

$$k(\boldsymbol{x}_i, \boldsymbol{x}_j)) = \mathbb{E}[(f(\boldsymbol{x}_i) - \mu(\boldsymbol{x}_i))(f(\boldsymbol{x}_j) - \mu(\boldsymbol{x}_j))] \tag{2.11}$$

Assuming Gaussian noise with variance σ^2_{noise} on our measurements y_i,

$$\boldsymbol{y}_i = f(\boldsymbol{x}_i) + \epsilon_{\mathcal{N}(0,\sigma^2_{noise})} \tag{2.12}$$

Gaussian Process conditioned on sampled points represents a posterior distribution for f. After evaluating f at points $x_1, ..., x_t$ the predictive posterior $P(f_{t+1}|\boldsymbol{x}_{1:t}, \boldsymbol{y}, \boldsymbol{x}_{t+1}) \sim \mathcal{N}\big(\mu_t(\boldsymbol{x}_{t+1}), cov_t(\boldsymbol{x}_{t+1})\big)$ can be computed in closed form

with mean and covariance:

$$\mu_t(\boldsymbol{x}_{t+1}) = \boldsymbol{k}^T[\boldsymbol{K} + \sigma^2_{noise}\boldsymbol{I}]^{-1}\boldsymbol{y} \tag{2.13}$$

$$cov_t(\boldsymbol{x}_{t+1}) = k(\boldsymbol{x}_{t+1}, \boldsymbol{x}_{t+1}) - \boldsymbol{k}^T[\boldsymbol{K} + \sigma^2_{noise}\boldsymbol{I}]^{-1}\boldsymbol{k}, \tag{2.14}$$

where $\boldsymbol{k} \in \mathbb{R}^t$, with $\boldsymbol{k}_i = k(\boldsymbol{x}_{t+1}, \boldsymbol{x}_i)$; $\boldsymbol{K} \in \mathbb{R}^{t\times t}$ with $\boldsymbol{K}_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$; $\boldsymbol{I}$ is an identity $\in \mathbb{R}^{t\times t}$, and $\boldsymbol{y}$ is a vector of values obtained after evaluating $f(\boldsymbol{x}_1), ..., f(\boldsymbol{x}_t)$. More details can be found in [31].

The prior mean function $\mu(\boldsymbol{x})$ can be set to 0 if no relevant domain-specific information is available. The kernel $k(\boldsymbol{x}_i, \boldsymbol{x}_j)$ encodes how similar f is expected to be for two inputs $\boldsymbol{x}_i, \boldsymbol{x}_j$. The value of $f(\boldsymbol{x}_i)$ has a significant influence on the posterior value of $f(\boldsymbol{x}_j)$, if $k(\boldsymbol{x}_i, \boldsymbol{x}_j)$ is large. The Squared Exponential (SE) kernel is a commonly used similarity metric:

$$k_{SE}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \sigma_k^2 \exp\Big(-\frac{1}{2\boldsymbol{l}^2}\|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2\Big), \tag{2.15}$$

where σ_k^2, $\boldsymbol{l}^2$ denote signal variance and a vector of length scales respectively; σ_k^2, $\boldsymbol{l}^2$ are called 'hyperparameters' in BO literature. It is possible to adjust these automatically during optimization to learn the overall signal variance and how quickly f varies in each input dimension. Squared Exponential kernel is a special case of a more general class of Matérn kernels [32].

2.3.2 Bayesian optimization for robotics

Bayesian optimization has been used in robotics, usually to tune parameters of controllers, cost functions or planners. The advantage of using a global search framework like BO is that globally optimal policies can be discovered. These may perform better than human hand-designed policies as well as locally optimal solutions. As shown in [33], the alternative of using Reinforcement Learning (RL) algorithms for policy search might not discover global optima reliably. [34] uses BO to optimize parameters of a neural network controller for a two-link pendulum on a cart. [5] solves the problem of balancing a pole on a real inverted pendulum, by optimizing the cost function for a LQR controller. Authors also propose a way of trading-off simulation and hardware experiments to optimally switch between hardware and simulation for experiments. [35] learns the policy for planning the

path of a simulated robot that minimizes the uncertainty in its environment.

The most prominant works using Bayesian Optimization for locomotion controllers include [36], [37], [6] and [38]. These works typically involve simpler to control robots or smaller bipeds. [36] uses AIBO quadrupeds, [37] uses snake robots, [6] uses hexapods and [39] uses a small biped. [37] optimizes 3 parameters for a open-loop parametrized controller that generate trajectories for a 16 degree of freedom snake robot. The generated solutions find fast moving gaits of about $5m/s$ between 10-40 trials, depending on the difficulty of the task. Such performance was hard to achieve with hand-tuned parameters. [36] uses BO to optimize open-loop gait parameters for a AIBO robot to maximize velocity and smoothness of movement. Authors find that the Bayesian optimization approach not only outperformed previous techniques, but used drastically fewer evaluations.

[39] uses BO for optimizing gaits of a 4 dimensional controller on a small biped o a boom. Authors report needing 30-40 samples for finding walking gaits for a finite-state-machine based controller. This controller consists of 4 threshold parameters that switch the states of a finite-state-machine cycling between stance and swing for each leg. Optimizing a higher-dimensional controller, which is needed for more complex robots and tasks, might prove to be even more challenging. The learning could be especially difficult if a significant number of the points/parameters sampled would lead to unstable gaits and falls. Such samples might result in eventual wear and breakage of the robot hardware (even if care is taken to prevent actual falls). Hence, there is a need to either limit search spaces to "safe" points, or bias the search towards such points.

[6] tabulates best performing points in simulation versus their average score on a behavioural metric for a hexapod robot. This metric then guides BO to quickly find behaviours that can compensate for damage of the robot. The search on hardware is conducted in behaviour space, and limited to pre-selected "successful" points from simulation. This helps make their search faster and safer on hardware. However, if an optimal point was not pre-selected, BO cannot sample it during optimization, losing global optimality guarantees. "Best points" are cost-specific, so the map needs to be re-generated for each cost.

Our proposed method generalizes to highly dynamic behaviours and discontinuous cost functions, while maintaining the global guarantees of BO. We also bias our search towards sampling points successful in simulation (but not exclusively),

leading to a sample-efficient and safer search.

2.3.3 Incorporating behavior information in Bayesian Optimization

Adding prior knowledge based on behavior of controllers is a promising avenue to speed up optimization. In higher dimensions, the performance of BO typically degrades, as discussed in [7]. Incorporating domain knowledge, in the form of behavior might speed up optimization in such cases.

Domain knowledge can be incorporated in the form of trajectory information. If we could query the resultant trajectory of each point, the problem of optimizing the cost would be trivial, as can be expected. To elaborate, consider a Markov Decision Process starting from a state x_0, consisting of states x and actions u, that follow a transition function $P(x'|x,u)$. Here P defines the probability of transitioning to state x' given current state x and taking action u. A typical reinforcement learning problem is characterized reward function $R(x,u,x')$ and a policy $P_\pi(u|x,\theta)$ that maps states to actions, given some parameters θ. Bayesian Optimization, in the classic setting, studies episodic RL which looks at rewards at the end of the execution, or the value of a trajectory. Consider a trajectory $\xi = (x_0, u_0, \cdots, x_T, u_T)$. The probability of sampling this trajectory is

$$P(\xi|\theta) = P(x_0)\prod_{i=0}^{T} P(x_{i+1}|x_i,u_i)P(u_i|x_i,\theta) \tag{2.16}$$

The cumulative reward of a trajectory is given as

$$R(\xi) = \sum_{i=0}^{T-1} R(x_{i+1},u_i,x_i) \tag{2.17}$$

The expected return of a parameter set θ then becomes

$$J(\theta) = \int R(\xi)P(\xi|\theta)d\xi \tag{2.18}$$

The expected return could itself be a vector consisting of different aspects of a trajectory, for example, distance walked, average speed, cost of transport, etc.

The overall cost is a combination of these:

$$f(\theta) = J(\theta)^T w \tag{2.19}$$

where $w \sim \mathcal{N}(0, \Sigma_p)$.

If we fit a GP to this linear setting, the mean and covariance functions become

$$E[f(\theta)] = J(\theta)^T w \tag{2.20}$$

$$E[f(\theta), f(\theta')] = J(\theta)^T \Sigma_p J(\theta) \tag{2.21}$$

If $J(\theta)$ is a scalar, we can determine the the fitness or cost of each controller parameter uniquely in the second trial. After sampling the first point randomly, the UCB acquisition for the posterior (similar to Equation 2.8) is

$$\begin{aligned}
\mu(\theta) &= J(\theta)^T w \\
\sigma(\theta) = k(x_2, x_2) - k(x_2, x_1) k(x_1, x_1)^{-1} k(x_1, x_2) &= 0 \\
UCB(\theta) &= \mu(\theta) + \alpha\sigma(\theta) \\
UCB(\theta) &= J(\theta)^T w \\
\theta_{next} &= \arg\max_{\theta} UCB(\theta)
\end{aligned}$$

$\sigma(\theta) = 0$ is an intuitive result, as for a deterministic linear function, once we know the weights w, there is no uncertainty about the posterior mean of any parameter θ. It is exactly equal to $J(\theta)^T w$. Optimizing the acquisition function is equivalent to optimizing the cost. Although it can be difficult to optimize such a function globally, there is no real advantage to using Bayesian optimization anymore. Optimizing the acquisition function can typically need a very large number of samples, which are assumed to be much cheaper than sampling a point in BO. However, in the current case, sampling points for optimizing the acquisition function is just as costly as sampling the actual cost. This is impossible for a system without unrolling almost every possible θ for a controller, which can be prohibitively large number of points. To overcome this, several alternative solutions to evaluating $J(\theta)$ for each point exist in literature.

[40] proposes a Behavior Based Kernel (BBK) that uses similarity of trajectories induced by the evaluated policies. A kernel constructed to use behavior-based similarity offered an improvement over a standard SE kernel. However, BBK re-

quired obtaining trajectory data every time kernel values $k(\boldsymbol{x}_i, \boldsymbol{x}_j)$ were evaluated. This would not be tractable when optimizing locomotion controllers, as it would need a full evaluation to determine the kernel distance (similar to the problem we described above). The authors suggest combining BBK with a model-based approach to overcome this difficulty by learning a transition model for the problem. But building a reliable model might need several data points, and building good models with few data points is an interesting research question in itself.

[6] builds a behavior map based on the simulation of controllers. Thus, they overcome the problem of building models of the robot from hardware data, but rely on behavior transfer between simulation and hardware. However, rather than using behavior as part of the kernel, like in [40], they build a look-up map between behavior and "successful" controllers. They then search in the low-dimensional behavior space, among the pre-selected controllers from simulation. Though this leads to a highly sample-efficient search and potentially safe samples, this method cannot search beyond the pre-selected points, and hence loses the global guarantees of BO.

[41] designs a kernel for LQR assuming a quadratic cost and a one-dimensional linear system model. For such a problem, they show improvements in cost within 3 trials. It is unclear how this approach would scale to higher dimensional problems and non-LQR controllers.

Our approach constructs an informed kernel that incorporates behavior-based similarity, in a manner that ensures $k(\boldsymbol{x}_i, \boldsymbol{x}_j)$ can be obtained efficiently when running BO. We pre-calculate locomotion-specific features on a high-fidelity simulator by running short simulations, enabling us to efficiently calculate the kernel distances during optimization. Our feature transform also biases the search towards high performing points in simulation, which have a higher chance of being stable on hardware than randomly selected points. In this manner, we try to incorporate behavior information from simulation, as also was proposed in [6], but maintain global guarantees of BO.

2.3.4 Learning with neural networks in robotics

Recently, there has been a surge in work using neural networks to control robots, though hardware results on locomotion robots are still to be seen. We do a very brief review of these methods, and try to understand ways in which incorporating

information from simulation might help these methods.

2.3.4.1 Deep Reinforcement Learning for locomotion

[42] formulates the problem of learning locomotion gaits as a actor-critic Reinforcement Learning with neural networks as function approximators for policy and value functions. The neural network takes as input terrain and robot states, parametrized leaps or steps as output. The input dimensionality is large (83D robot state and a 200D terrain state) and maps to a 29D control space, that operates at a high level, like leaps and steps for the robot, leading to about 570k parameter large network. The outputs from the policy network is fed to a finite state machine which is hand-designed to achieve the goal dictated by the neural network. While highly successful at traversing very uneven terrain, this network needs over 10 million data points to train the network. Such approaches, however, are not data-efficient enough to support learning optimal parameters for locomotion controllers on real hardware. So in our work we are interested in combining sample efficiency of an approach like Bayesian Optimization with the flexibility and scalability of deep neural networks.

[43] uses deep reinforcement learning to learn locomotion tasks for a planar biped, humanoid walker and a quadruped, training the neural network with increasingly difficult terrains. The network takes as input the terrain state and the robot state and directly outputs the torques for each joint. They develop a robust policy gradient method that uses a trust region constraint to restrict the policy update from changing too much, making the policy gradient robust to noise and hyperparameter selection. This results in highly dynamic and robust gaits in simulation, though it is unclear how such training can be carried out on hardware.

Using the idea that learning over increasingly difficult terrains leads to better learning rates, as described in [43], we would like to experiment with learning over multiple increasingly perturbed models. Similar to [8], this might result in robust policies that generalize to hardware. Another addition would be to use policy gradient approximations from simulation in combination with gradients observed on hardware to add simulation data to hardware data. While this might lead to faster training, it would be interesting to study how to trade off simulation policy gradients with hardware gradients.

2.3.4.2 Guided policy search and related methods

Guided policy search is a recently popularized method for learning policies for robotic tasks [44] using neural networks. It has been applied to learning policies from images for complex manipulation tasks, such as peg-in-hole [45], contact-rich tasks like opening doors [46], etc. These networks represent global policies that are initialized by locally optimal policies. On facing new situations, where the local policies fail, a new local policy is initialized and trained to minimize cost and stay close to the global policy. The global policy is now updated to match the local policy. This ensures an increasingly general global policy that stays close to the initial policy, minimizing the "forgetting" factor for neural networks.

While there hasn't been work on locomotion on hardware with guided policy search, initializing the network in simulation over perturbed models and implementing on hardware seems straightforward. To enable fast learning on hardware, it might be beneficial to aid the policy gradients from optimal gradient step-size from simulation. This can be done either using natural gradients [11] from simulation or a meta-function that predicts gradient step-size [12]. Aiding gradient descent based on information from simulation can help with sample-efficiency on hardware. This was originally proposed in [47] where exact evaluations were aided with approximate gradients to suggest local improvements. While the exact value of the value function might be different between simulation and hardware, the gradient direction and step-size might have higher probability of transferring. Since the learning rates of these methods are sensitive to these hyper parameters, learning these in simulation seems straight-forward and beneficial. This can be done in combination with a mismatch-map which predicts the likelihood of transfer between simulation and hardware. If we expect high mismatch, we can use hardware samples to learn the correct learning rate.

CHAPTER 3

Learning to learn

Learning to learn is growing increasingly popular in machine learning and robotics. It can refer to learning hyper-parameters for learning methods to overcome the need for hand-tuning. In our case, we are using simulations to extract information that will enhance the rate of learning on hardware.

In this chapter we describe two ways of adding data from simulation to hardware searches. We start by hand-designing a locomotion-specific feature transform which is inspired from the Determinants of Gait walking metric [48]. Next, we explain an automatic way of learning a feature transform from data using neural networks. Both these approaches aim at collecting data by running a lot of simulations ("Big simulation") and use it to extract behavioral information about the controller space. This information will later be used for hardware and perturbed-simulation experiments that demonstrate ultra-sample-efficiency at learning controllers.

3.1 Distance metrics from simulation

In this section, we describe our proposed bipedal locomotion specific feature transform, followed by a more general transform learned from data.

3.1.1 A bipedal locomotion specific transform

The proposed locomotion feature transform is a generalization of the Determinants of Gaits (DoG) used by physiotherapists to evaluate the quality of human

walking [48]. This transform is designed to generalize to a range of locomotion controllers and robot morphologies, unlike our previous work [49], which focused on human-like robots and controllers. We mainly get rid of some of the human-specific features that do not have a clear impact on the quality of walking, and change others to accommodate general bipedal walking controllers. Our new locomotion kernel looks at the following aspects :

1. M_1 **: Swing leg retraction** – We look at the swing leg trajectory in swing and if the maximum leg retraction is more than a threshold, we set $M_1 = 1$. Otherwise, $M_1 = 0$. Higher swing leg retraction is better for disturbance rejection in the presence of obstacles, etc. Hence, a controller is less likely to fall if $M_1 = 1$.

2. M_2 **: Center of Mass height** – We look at the Center of Mass (CoM) height at the start and end of each step. If the CoM height stays about the same (change is below a threshold), we set $M_2 = 1$. Otherwise, $M_2 = 0$. This metric checks that the robot is not falling across steps, but allows changes in CoM height within a step. Human-like walking typically exhibits a oscillatory CoM movement within a step, while with a linear inverted pendulum controller, the CoM height stays constant through the step. Thus, looking at just the CoM height at the start and end of a step includes both these situations and makes sure that the controller isn't falling. Thus, $M_2 = 1$ controllers are likely to walk.

3. M_3 **: Trunk lean** – We compare the mean trunk lean at the start and end of a step, and if the average lean is about the same (change is below a threshold), we set $M_3 = 1$. Otherwise, $M_3 = 0$. This ensures that the trunk is not changing orientation between steps, but allows the lean to change within a step. Again human-like walking typically exhibits a oscillatory torso movement within a step, while typical walking controllers maintain a constant torso lean. Thus, looking at just the torso lean at the start and end of a step includes both these situations and makes sure that the controller isn't falling. Thus, $M_3 = 1$ controllers are likely to walk.

4. M_4 **: Average walking speed** – We evaluate the average speed of a controller per step and set $M_4 = v_{avg}$. Unlike the other metrics, M_4 is not binary and helps distinguish between controllers that satisfy all conditions

of M_{1-3}. In general, faster walking controllers could be better, though that depends on the desired walking speed. M_4 helps distinguish controllers that score the same of M_{1-3} but lead to different behaviors in simulation.

The step metrics M_{1-4} are collected per step i and summed over the total number of steps N.

$$score^i = \sum_{j=1}^{4} M_j^i \tag{3.1}$$

$$score_{DoG} = \sum_{i}^{N} score^i \tag{3.2}$$

In general, a higher score implies better performance of a controller for M_{1-3}. If M_{1-3} are 0 for a particular controller, it is likely to fall. On the other hand if M_{1-3} are 1 for a controller, it is likely to walk. However, the score doesn't have a fixed relationship to a particular cost function. The cost depends on the specific desired behavior/outcome.

Controllers that chatter (step very fast, with step time less than $100ms$) can have a large number of steps before falling. Since this could lead to a misleadingly high score, the DoG score is scaled by the fraction of time the simulation walked before falling. If the simulation terminated at time t_{sim} and the desired time for simulation was t_{max}, the final DoG score ϕ becomes:

$$\phi = score_{DoG} \cdot \frac{t_{sim}}{t_{max}} \tag{3.3}$$

ϕ defines a reparameterization of a point from the original space of controller parameters into a 1-dimensional space. We use ϕ to define a 1-dimensional kernel that utilizes the Determinants of Gait scores. The functional form of this kernel is the same as Squared Exponential kernel. However, instead of Euclidean distances of points in the original space, we use distances between the DoG scores of the points:

$$k(\boldsymbol{x}_i, \boldsymbol{x}_j) \rightarrow k(\phi(\boldsymbol{x}_i), \phi(\boldsymbol{x}_j)) \tag{3.4}$$

$$k_{DoG}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \sigma_k^2 \exp\Big(-\frac{1}{2\boldsymbol{l}^2} \|\phi(\boldsymbol{x}_i) - \phi(\boldsymbol{x}_j)\|^2 \Big), \tag{3.5}$$

where hyperparameters σ_k^2, $\boldsymbol{l}^2$ are signal variance and length scale respectively.

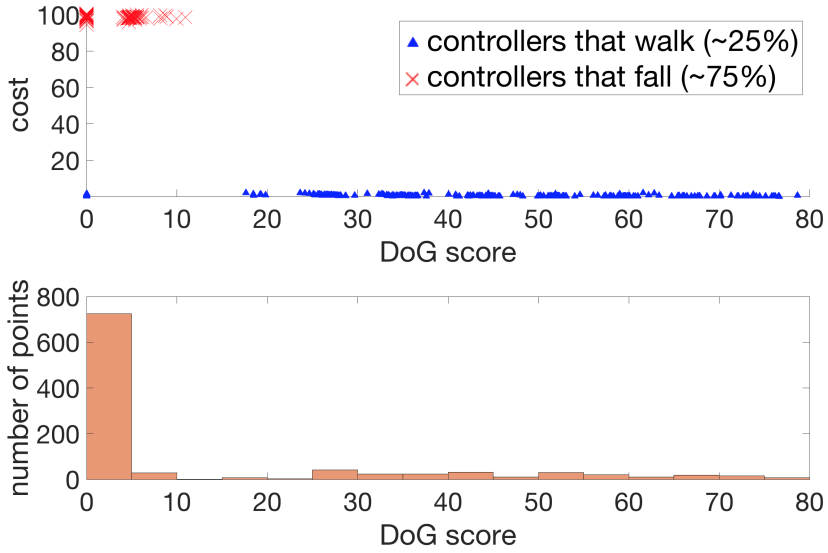


Figure 3.1: DoG score vs cost for 1000 randomly selected controller parameters (controller & cost from Sections 4.1.2.2,5.1.3). Lower DoG scores usually lead to higher costs & falling. A few points that step very fast (chatter) don't fall in simulation, so can have low cost and low DoG score. But such points are very likely to fall on hardware.

We refer to k_{DoG} as 'DoG-based kernel' in the following sections. To speed up calculation of kernel distances during optimization, we pre-calculate ϕ for a large grid of points in simulation. We run short simulations of each point/controller, evaluate ϕ and store it in a large look-up table.

The DoG score helps cluster controllers based on their behaviour in simulation. The hope is that behavioral cues like the ones described in metrics M_{1-4} have a higher chance of transferring between simulation and hardware than costs. On hardware, once we have evaluated a controller with a particular value of ϕ, we expect controllers with similar values of ϕ to have a similar cost. This roughly splits the cost function landscape, separating points that can potentially walk, and those that cannot, as shown in Figure 3.1. Suppose we sample an unstable point with a low ϕ score and obtain a high cost on hardware (we can also seed our search with such points). We can then be fairly certain of other unstable points doing poorly as well. As a result, we can focus on potentially promising points to sample – making the search more sample efficient and biased towards sampling safe points.

Note that in Equation 3.1, metrics M_{1-4} are weighed equally when summing up. This over-condensed version is very efficient at finding walking points as

it groups all non-walking points as one and separates them from all walking points. However, it is not good for distinguishing between walking points. In other words, if we are only searching for reasonably good walking points, and do not care about optimizing the quality of walking of each point, the 1 dimensional kernel is ideal for quickly reaching such a point. However, if we care about the quality of walking, for example, we would like to walk at a particular speed, or minimize metabolic energy, the 1 dimensional kernel isn't ideal for distinguishing between such points. A small but useful addition could be to learn the weights for a 4-dimensional feature transform $[M_1, M_2, M_3, M_4]$ using Automatic Relevance Determination (ARD) [31] before summing up. This would enable us to weigh the 4 metrics depending on their importance for a particular task, controller or robot. Another addition would be to use a 4-dimensional kernel, which might be less sample-efficient, but more robust to mismatch between simulation and hardware, as well as help optimize particular cost functions better.

3.1.2 A feature transform from data

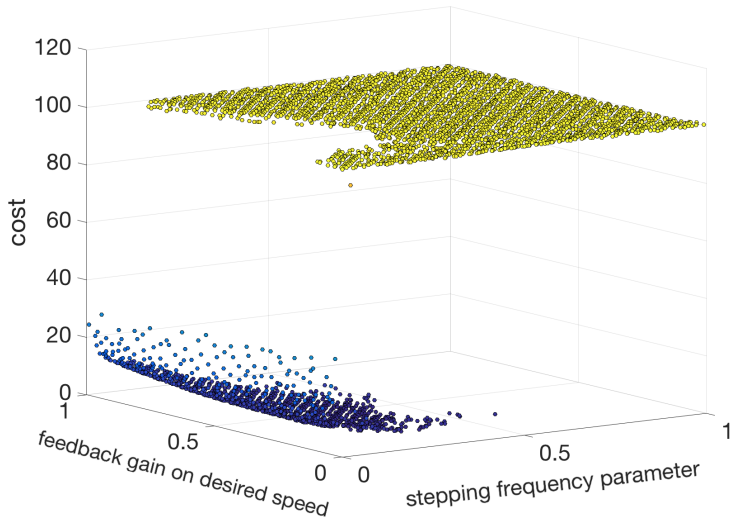
While hand-designed feature transforms can be very data-efficient, as well as shed light on what the optimization is prioritizing, it isn't viable for situations where such information is not available. In this section, we automatically learn an informed transform for optimizing bipedal locomotion controllers without extensive domain-expertise. We start by running short simulations in a high-fidelity simulator for the locomotion controller of interest. We run simulations for a large grid of parameter sets and record the resulting costs and simulation trajectories. Costs obtained during short simulations serve as approximate indicators of the quality of the controller parameters for longer simulations, though they can diverge later. Our idea is to use the behaviors obtained from simulation to generate an informed distance metric for the Bayesian Optimization kernel.

3.1.2.1 Regression with Implicitly Asymmetric Loss

We consider a cost function focused on matching the desired walking speed and heavily penalizing falls:

$$cost_{sim} = \begin{cases} 100 - x_{fall}, \text{if fall} \\ 10 \cdot ||v_{tgt} - \boldsymbol{v}_{actual}||^2, \text{if walk} \end{cases} \tag{3.6}$$

Figure 3.2: 2D slice of the cost landscape.

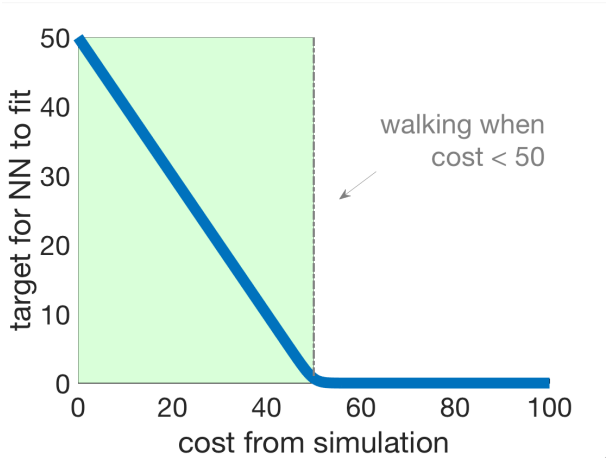


where x_{fall} is the distance travelled before falling, v_{tgt} is the target velocity and $\boldsymbol{v}_{actual}$ is the vector containing actual velocities of the robot. This kind of cost function is of interest because it helps easily distinguish points that walk from points that fall. Similar costs have been considered in prior work when optimizing locomotion controllers [49, 50].

Figure 3.2 shows a scatter plot of applying cost from Equation 3.6 to simulations of a 5-dimensional controller for the ATRIAS robot as introduced in Section 4.1.2.2. For visualization we only vary the 2 most sensitive dimensions, leading to a 2-dimensional subspace of the parameter space. We pick a well-performing set of parameters (in 5D), then vary the first two dimensions to obtain a 2D subspace. The two parameters we vary are the stepping frequency of the controller and the feedback gain on the desired speed. As seen in the figure, less than a quarter of this subspace contains points corresponding to low-cost simulation results (blue points on the scatter plot).

The challenge comes from the fact that the boundary between the well-performing (blue) and poorly performing (yellow) parameters is discontinuous. This is a typical landscape for bipedal systems, where a controller that makes the robot fall is much worse than one than walks, and the boundary is extremely sharp. While there can be variations to how costs are structured among stable walking points – efficiency vs speed vs distance covered, parameters that fall are much worse. Fitting such cost function with regression could be difficult. Learn-

Figure 3.3: Cost transformation.



ing to reconstruct the boundary exactly using the training set might result in overfitting and poor performance on the test set. Trying to fix this by applying regularization is likely to result in high loss and uncertainty about the points close to the boundary. This is particularly problematic if poorly performing points lie close to some of the most promising regions of the parameter space, which is the case for the setting we consider. We propose to use a transformation of the cost as the target for the supervised learning. Our approach is to train a deep neural network to reconstruct a reflected shifted softplus function of the cost:

$$score_{NN} = \zeta\big(cost_{walk} - cost_{sim}(\boldsymbol{x})\big) \tag{3.7}$$

Here ζ is a softplus function: $\zeta(a) = ln\big(1+e^{a}\big)$, $cost_{walk}$ is the average cost for the parameter sets that walk during short simulations, $cost_{sim}(\boldsymbol{x})$ is the cost computed by the simulator for vector $\boldsymbol{x}$ of controller parameter values (Eq. 3.6). Using this transformation yields a "score" function such that parameter sets which produce poor results in simulation are mapped to values close to zero. With this, the differences in scores of the poorly performing parameter sets become small or zero. In contrast, the differences in scores of the parameter sets yielding potentially promising results remain proportional to the difference in the corresponding costs. Figure 3.3 gives a visualization of this transformation.

Cost transformation in Equation 3.7 serves to essentially create an asymmetric loss for neural network training. This asymmetric loss is minimized when the promising (low-cost, high-score) points are reconstructed correctly. For the poorly

performing (high-cost, low-score) points, it only matters that the output of the neural network is close to zero. Such asymmetric loss can be interpreted as implementing a hybrid of regression and "soft" classification. The regression aspect aims to fit the promising points which correspond to parameters yielding walking behaviors. The "soft" classification causes an increase in the loss only if a poorly performing point is "mis-classified" as well-performing (output of the neural network is not close to zero). Conveniently, no actual change of the problem formulation is made in the implementation – the neural network training simply proceeds to solve a regression problem.

The asymmetry in the loss is essential for training to reconstruct noisy costs with sharp discontinuities. Such cost functions are frequently used in optimization of locomotion controllers, with the aim to sharply penalize falling. While there can be variations to how costs are structured among stable walking points - efficiency vs speed vs distance covered, parameters that fall are much worse. This yields cost landscapes with sharp discontinuities. This challenge is addressed by learning to reconstruct the transformation of the cost (as discussed above) in combination with using a L1 loss, instead of the usual L2 loss when training the neural network. With this, errors in reconstructing points on the boundary contribute only linearly to the overall loss. This helps achieve a better fit of the stable parts of the parameter space, instead of focusing on the boundary.

The resultant transform $\phi(\boldsymbol{x})$ from this cost based transform is:

$$\phi(\boldsymbol{x}) = score_{NN}(\boldsymbol{x}) \tag{3.8}$$

We utilize the reconstructed transformed costs to define *asymNN* kernel for Bayesian Optimization:

$$k(\boldsymbol{x}_i, \boldsymbol{x}_j) \rightarrow k(\phi(\boldsymbol{x}_i), \phi(\boldsymbol{x}_j)) \tag{3.9}$$

$$k_{asymNN}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \sigma_k^2 \exp\Big(-\frac{1}{2\boldsymbol{l}^2} \|\phi(\boldsymbol{x}_i) - \phi(\boldsymbol{x}_j)\|^2 \Big), \tag{3.10}$$

with hyperparameters $\sigma_k, \boldsymbol{l}$. The proposed approach is able to clearly separate the unpromising part of the parameter space. Under the resulting distance metric poorly performing sets of parameters are close together and far from well-performing ones. A smaller priority is given to high-precision reconstruction of the poorly performing region of the parameter space. This is desirable, since it

has been observed that usually simulators are optimistic compared to the real word [51]. This is indeed the case with the locomotion simulators we consider: if a set of controller parameters causes falling or instability in simulation, it is very unlikely that the same controller parameters would result in successful walking when applied to real hardware. With that, distinguishing all the various ways in which poorly performing points fail is not useful. Using *asymNN* kernel helps rapidly separate bad regions of the parameter space from the regions that might contain promising points.

3.1.2.2 Reconstructing Cost-agnostic Trajectory Summaries

While learning a distance metric from the cost could be effective for a wide variety of problems, frequently there is a need for a cost-agnostic approach. Such cases arise when the data from the simulator is computationally expensive to collect and we need to change the cost function, and if training the network takes time. Different tasks might call for slightly different cost functions. For example, high energy consumption could be penalized if energy use needs to be restricted; robustness of the walk could be emphasized if only stable walking is acceptable; if achieving the desired speed is the most important factor – then the cost might instead only reflect how well the desired speed is maintained. For such cases we propose to train a neural network to reconstruct summaries of trajectories that are cost-agnostic, then utilize these trajectory summaries for constructing kernel distance metric.

In most cases, a summary of trajectory information contains all the pertinent information. Hence, we focus on collecting the following aspects of the simulated trajectories: walking time (time before falling), energy used during walking, position of the torso, angle of the torso, coordinates of the center of mass at the end of the short simulation runs. These summaries of simulated trajectories are collected for a range of controller parameters and comprise the training set for the neural network to fit (input: $\boldsymbol{x}$ – a set of control parameters; output: $\boldsymbol{traj_x}$ – the corresponding trajectory summary obtained from simulation). The outputs of the (trained) neural network offer the reconstructed/approximate trajectory summaries: $score_{trajNN}(\boldsymbol{x}) = \widehat{\boldsymbol{traj}}_{\boldsymbol{x}}$, where $\boldsymbol{x}$ is the input controller parameters, and $\widehat{\boldsymbol{traj}}_{\boldsymbol{x}}$ is the corresponding reconstructed trajectory summary. These are then

used to define a distance metric for the kernel for Bayesian Optimization:

$$\phi(\boldsymbol{x}) = score_{trajNN}(\boldsymbol{x}) \tag{3.11}$$

$$k_{trajNN}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \sigma_k^2 \exp\Big(-\frac{1}{2\boldsymbol{l}^2} \|\phi(\boldsymbol{x}_i) - \phi(\boldsymbol{x}_j)\|^2 \Big) \tag{3.12}$$

with hyperparameters $\sigma_k, \boldsymbol{l}$.

The general concept of utilizing trajectory data to improve sample efficiency of BO has been proposed before, for example in [40]. However, prior work assumed obtaining trajectory data is possible every time kernel values $k(\boldsymbol{x}_i, \boldsymbol{x}_j)$ need to be evaluated. This is not the case in our setting. Trajectory summaries are initially obtained via costly high-fidelity simulations, and it would be infeasible to compute trajectory information via simulation during BO. So the BO would be limited to sampling from a very large pre-simulated grid of points. Hence, our approach is to train a neural network to learn reconstructing trajectory summaries first. Running a forward pass of the neural network is a relatively inexpensive operation, hence reconstructed/approximate trajectory summaries can be quickly obtained during BO whenever $k_{trajNN}(\boldsymbol{x}_i, \boldsymbol{x}_j)$ needs to be computed. Note that the trajectory information we extract is generic and can be applied to other problems without requiring in-depth domain knowledge. When applying this approach to a new domain, the strategy would be to include trajectory information used to compute cost functions that are of interest/relevance in the domain. For example, for a manipulator, the coordinates of end-effector(s) could be recorded at relevant points. In contrast, information extracted in [6, 49] are more domain specific.

In addition to offering a cost-agnostic approach, our trajectory-based kernel retains more information about important aspects of simulated trajectories than a cost-based kernel. We observe that this re-parameterization could be more effective than a cost-based kernel in cases when the kernel does not retain enough information to effectively optimize the acquisition function used in Bayesian Optimization, due to over-condensing the characteristics of the simulation. In particular, cost-based kernel offers most sample-efficient results the case of using lower-dimensional controller and the case of using a smooth cost with a higher-dimensional controller - basically easier to optimize problems. The trajectory-based kernel also significantly outperforms standard Bayesian Optimization, even when the optimization uses challenging discontinuous costs.

3.2 Modelling mismatch between simulation and hardware

While all the generalized transform ϕ proposed in the previous section can successfully characterize the quality of a gait, large mismatch between a simulator and real-world hardware could still present a challenge. Some controller parameters could yield good gait characteristics in a short simulation, but perform poorly during a longer trial on hardware or simulation. While this issue did not arise during our hardware experiments with the controller described in Section 4.1.2.2, we anticipate that with a different and higher-dimensional controller such mismatch could become an issue. Hence for our experiments with 50 dimensional virtual neuromuscular controller we explore learning the mismatch and adjusting the kernel accordingly.

We build a separate model for the mismatch between simulation and hardware $g(\boldsymbol{x}) \sim \mathcal{GP}(0, k_{SE}(\boldsymbol{x}_i, \boldsymbol{x}_j))$, starting with a prior mismatch of zero. For each controller $\boldsymbol{x}_i$ explored during BO, we observe the difference between its DoG score in simulation and on hardware: $d_{\boldsymbol{x_i}} = \phi_{sim}(\boldsymbol{x_i}) - \phi_{hw}(\boldsymbol{x_i})$. This difference $d_{\boldsymbol{x_i}}$ becomes a "training point" for the GP that is used to model the mismatch. The posterior mean g_* is then computed using standard Gaussian Process regression. This allows us to predict simulation-hardware mismatch for the whole space of controller parameters. Now, we would like to add this predicted mismatch information to the optimization of the controller.

In our first attempt, we expand the DoG-based kernel to have one more dimension. The re-parameterization becomes:

$$\boldsymbol{\phi}^{adj}_{\boldsymbol{x_i}} = \big[\phi(\boldsymbol{x_i}), g_*(\boldsymbol{x_i})\big] \tag{3.13}$$

$$k_{adj}(\boldsymbol{x_i}, \boldsymbol{x_j}) = \sigma_k^2 \exp\Big(-\frac{1}{2\boldsymbol{l}^2} ||\boldsymbol{\phi}^{adj}_{\boldsymbol{x_i}} - \boldsymbol{\phi}^{adj}_{\boldsymbol{x_j}}||^2 \Big) \tag{3.14}$$

This amounts to a product of two kernels – a kernel based on distances in feature transform ϕ and a kernel based on the distance is expected mismatch at the two

points.

$$k_{adj}(\boldsymbol{x_i}, \boldsymbol{x_j}) = \sigma_k^2 \exp\Big(-\frac{1}{2\boldsymbol{l}^2} || \big[\phi(\boldsymbol{x_i}), g_*(\boldsymbol{x_i})\big] - \big[\phi(\boldsymbol{x_j}), g_*(\boldsymbol{x_j})\big] ||^2 \Big) \tag{3.15}$$

$$= \sigma_k^2 \exp\Big(-\frac{1}{2l_1^2}\big(\phi(\boldsymbol{x_i}) - \phi(\boldsymbol{x_j})\big)^2 - \frac{1}{2l_2^2}\big(g_*(\boldsymbol{x_i}) - g_*(\boldsymbol{x_j})\big)^2 \Big) \tag{3.16}$$

$$= \sigma_k^2 \exp\Big(-\frac{1}{2l_1^2}\big(\phi(\boldsymbol{x_i}) - \phi(\boldsymbol{x_j})\big)^2 \Big) \cdot \sigma_k^2 \exp\Big(-\frac{1}{2l_2^2}\big(g_*(\boldsymbol{x_i}) - g_*(\boldsymbol{x_j})\big)^2 \Big) \tag{3.17}$$

Suppose we evaluate controller $\boldsymbol{x_i}$, and in simulation we get a walking behavior (high DoG score), but on hardware the controller falls. For the next few evaluations, BO with DoG-based kernel would associate high simulation-based DoG scores with bad performance. In contrast, $k_{DoG_{adj}}$ takes into account the high mismatch (high DoG score in simulation, low on hardware). Consequently, $k_{DoG_{adj}}(\boldsymbol{x_i}, \boldsymbol{x_j})$ would be highest only for $\boldsymbol{x_j}$s that have both similar simulation-based DoG scores and similar estimated mismatch. Hence, points with high simulation-based DoG scores and low predicted mismatch would still be 'far away' from the failed $\boldsymbol{x_i}$. They would be sampled if uncertainty about their costs is high or their mean posterior is promising.

We will call this adjusted kernel with hardware and simulation mismatch the adjusted DoG-based kernel in the future. We present preliminary results of this adjusted DoG-based kernel in Section 5.1.4. This formulation is similar in spirit to other approaches, such as [5]. However, while previous work "mistrusts" all simulation data, our formulation lets us fit a dynamic mismatch function from data. This lets us trust the simulation in some regions, while mistrust it in others.

There are several interesting questions that should be explored relating to mismatch modelling.

[40] proposed modelling the mismatch between a learned model and the real system by a second "residual" GP. They predict the expected cost at a new point using a convex combination of two GPs: $(1-\beta)\eta_1(\boldsymbol{x}) + \beta\eta_2(\boldsymbol{x})$.

$$\eta_1(\boldsymbol{x}) \sim \mathcal{GP}(0, k(\boldsymbol{x}, \boldsymbol{x}')) \tag{3.18}$$

$$\eta_2(\boldsymbol{x}) \sim \mathcal{GP}(m(\boldsymbol{x}, \boldsymbol{x}_i, ..., \boldsymbol{x}_n), k(\boldsymbol{x}, \boldsymbol{x}')) \tag{3.19}$$

β can be learned from data. η_1 models the residual between the predicted model and the actual data, initialized with a 0 prior. η_2 is a model learned over simulation points $m(\boldsymbol{x}, \boldsymbol{x}_i, ..., \boldsymbol{x}_n)$ which might be an inaccurate representation of the

actual system. The resultant GP has a mean which is a convex combination of the two GP means, and the same variance as the chosed k. The difference between our approach and the literature is that we are using the mismatch in the kernel, rather than in the mean. The reason for this is the following: [40] work uses simulation to model the mean of their Gaussian Process (as a prior), and keep the covariance the same. On the other hand, majority of the information from simulation in our approach is added to the kernel. So it is the kernel estimate that should be updated from hardware points, not just the mean. In the future, if we incorporate points from the simulation in our prior, it would be useful to add the modelled mismatch to the mean as well as the covariance.

Multiple sources of information can be added to Gaussian Processes [52]. For example, if we again model a residual over a GP fit in simulation, the true cost is a combination of two GPs:

$$\eta(\boldsymbol{x}) = \eta_1(\boldsymbol{x}) + \eta_2(\boldsymbol{x}) \tag{3.20}$$

where

$$\eta_1(\boldsymbol{x}) \sim \mathcal{GP}(0, k_{err}(\boldsymbol{x}, \boldsymbol{x}')) \tag{3.21}$$
$$\eta_2(\boldsymbol{x}) \sim \mathcal{GP}(\mu_{sim}(\boldsymbol{x}), k_{sim}(\boldsymbol{x}, \boldsymbol{x}')) \tag{3.22}$$

Here $\eta(x)$ is the true cost of the controller on hardware, $\eta_2(x)$ is a prediction from simulation, and $\eta_1(x)$ is the noise distribution, which is the mismatch between simulation and hardware. Points sampled from hardware do not have this mismatch, and hence are samples from the true GP $\eta(x)$. However points from simulation are samples from η_2 and their corresponding residual η_1 needs to be estimated.

[52] extend the parameter vector θ by an additional binary variable δ, which indicates whether the cost is evaluated in simulation ($\delta = 0$) or on the physical system ($\delta = 1$). Based on the extended parameter $a = (\boldsymbol{x}, \delta)$, we can model the cost by adapting the kernel of η to

$$k(\boldsymbol{x}, \boldsymbol{x'}) = k_{sim}(\boldsymbol{x}, \boldsymbol{x'}) + \delta\delta' k_{err}(\boldsymbol{x}, \boldsymbol{x'}) \tag{3.23}$$

Figure 3.4 shows a toy-example of using the approach from [52] in Bayesian

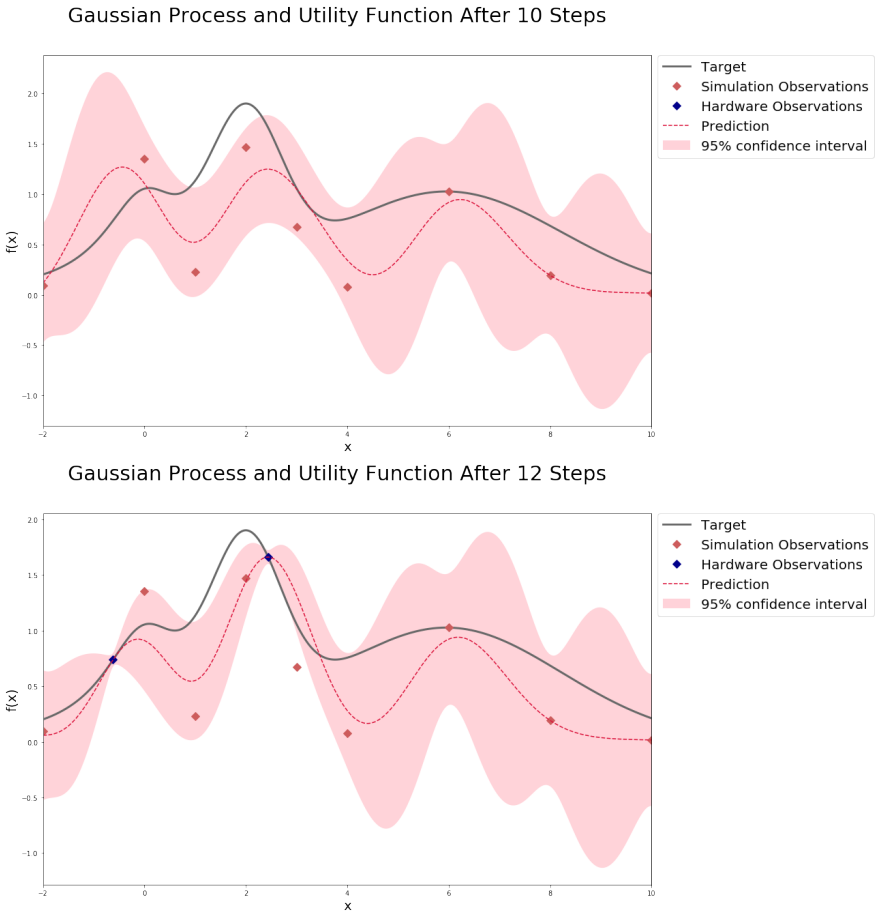


Figure 3.4: Bayesian optimization with 10 points from simulation and uniform uncertainty (top). The next 2 points are added from hardware, not sampling the optimum in two trials. There is too much uncertainty in the not promising regions, leading to a need to sample there on hardware.

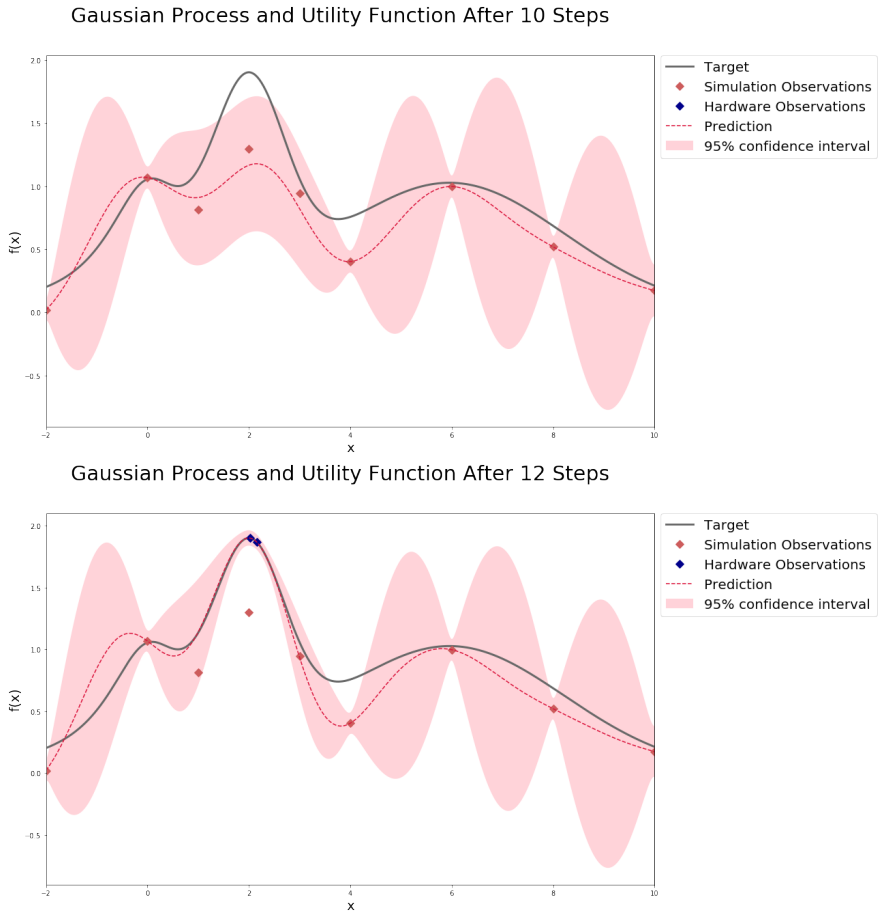


Figure 3.5: Bayesian optimization with 10 points from simulation and non-uniform uncertainty (top). The next 2 points are added from hardware, sampling the optimum. Since the uncertainty on the "bad" regions is already quite small, there isn't a need to explore there.

Optimization. 10 points are sampled from simulation with uniform uncertainty everywhere (top). 2 points are added from hardware with low uncertainty (bottom). Since there is still uncertainty in the potentially low performing regions, BO wastes a sample on unpromising region. Figure 3.5 shows an example with non-uniform noise in the simulation space.

We can be fairly certain of the potentially poorly performing points, even if the exact value of the cost might be different. This makes the BO not sample any unpromising points in the unpromising region on hardware. Hence it samples very close to optimum in as few as 2 samples. This toy-example motivates the need for variable uncertainty over the controller space to enhance sample-efficiency. We plan to model this with the mismatch map.

[5] develop an automatic way of trading off data from simulation and hardware based on the expected improvement of each source, normalized by the cost of sampling simulation vs hardware. If we modelled the mismatch from data, and incorporated it into the kernel in a similar manner, our resultant adjusted DoG-based kernel would become :

$$k_{adj}(\boldsymbol{x_i}, \boldsymbol{x_j}) = \sigma_k^2 \exp\Big(-\frac{1}{2l_k^2}\big(\phi_(\boldsymbol{x_i}) - \phi_(\boldsymbol{x_j})\big)^2\Big) + \sigma_m^2 \exp\Big(-\frac{1}{2l_m^2}\big(g_*(\boldsymbol{x_i}) - g_*(\boldsymbol{x_j})\big)^2\Big) \tag{3.24}$$

Instead of adding the mismatch to the feature transform ϕ, resulting in a product kernel as in Equation 3.15, we could add the distance between the mismatch at two points to their kernel distances. We would like to experiment with the various ways of modelling mismatch in literature and apply it to our problem.

While existing methods in literature talk about variable mismatch maps for different regions of space, to the best of our knowledge, there isn't any experiments on a real system. Since these maps can already need a lot of data to accurately model the inaccuracy if simulation, we are doubtful that they will succeed out of the box. To overcome this problem, we would also like to explore if characteristics of simulation can be used to create a prior mismatch. For example, controllers with higher impacts might have a high chance of falling on hardware even if they walk in simulation. Similarly, controllers that venture close to joint limits might be likely to fail on hardware, even if successful in simulation. If such data can be collected on a lower-dimensional, safer to use controller and generalized to higher dimensional controllers, we do not need to run higher dimensional controllers on hardware to collect this prior. However, the map would have to be learned on hardware to build a correct estimate of the mismatch for the current controller and system. The mismatch between robot and hardware can be extremely dynamic, changing over days, for example due to sensor mis-calibration, or unobserved environmental factors. Thus, it is crucial to learn the mismatch online, as running controllers.

Apart from adding these mismatched features to the mismatch map, we could also add them to the feature transform directly. As these are characteristics that seem important for distinguishing parameter performance on hardware, it might be beneficial to add them to the DoG-based kernel. Or we could use them for both - prior for mismatch, and as part of the kernel. While the kernel and mismatch prior are problem specific, building a map for mismatch between simulation and

hardware is general. So, we will first experiment with the mismatch map.

Another avenue to explore would be to compare the different ways of measuring mismatch between simulation and hardware. Some of the possibilities include, feature transforms, kinematic trajectories, dynamic trajectories, cost of simulation, etc.

3.3 Proposed work

3.3.1 A feature transform for 3-dimensional walking controllers

We would like to extend both the hand-designed feature transforms and the data-driven feature transforms to 3-dimensional systems. So far, all the features involve only 2-dimensional features of human walking. The original determinants of gaits look at 3-dimensional features of walking. Some that might be relevant to bipedal robot walking, and easily incorporated in our framework are:

- **Oscillatory Center of Mass between footsteps:** The CoM oscillates between the two feet per step, realized by a relative adduction of the hip. This results in reduction of vertical CoM displacement during a step.
- **Torso yaw and roll:** The torso roll and yaw are coupled and oscillatory, caused by a periodic pelvic tilt during stance. This helps maintain the CoM height increasing too much during stance, hence helping with energy conservation.

Encorporating these features into the feature transform would help generalizing to 3-dimensional walking.

3.3.2 Refine the mismatch-map formulation to suit sample-efficiency

We would also like to experiment with the mismatch formulations existing in literature, and refine them to suit our needs. Pre-initializing the map from knowledge from simulation can prove to be helpful. Using one or more sources to measure mismatch can also help get rid of noise in measurements. We would also like to

explore the benefits/disadvantages of using the several formulations provided in literature.

CHAPTER 4

Robots and controllers evaluated

In this chapter, we describe our completed and proposed experiments to validate our approach. Since this thesis is focused on using simulation to speed up hardware experiments, robot experiments form an important part of this proposal. Our hardware experiments so far have been on the ATRIAS robot on a boom. We present results on a feedback based reactively stepping controller, optimizing 5 and 9 parameters in 2 sets of experiments on hardware. Our simulation experiments have been on two different robot morphologies – a 7-link biped model in 2 dimensions, and an ATRIAS simulation on the boom. We present experiments on two different controllers, optimizing different number of parameters in each. We first present a feedback based reactively stepping controller – optimizing 5 parameters first, followed by optimizing 9 parameters. Then we present a neuromuscular walking controller – on a 7-link biped optimizing 16 parameters, and then on ATRIAS optimizing 50 parameters.

We also propose experiments on a 3-dimensional controller and feature transform on ATRIAS in the future. We are also working on a 3-dimensional controller for a humanoid robot like Sarcos, and propose simulation and hardware experiments for that too.

4.1 Completed Experiments

We start by highlighting our completed experiments for the domain-specific feature transform and the data-driven feature transform in simulation and on the ATRIAS robot. We also include experiments on a 7-link biped in simulation. The tested controllers are enumerated below:

1. A feedback-based reactively stepping controller - optimizing 5 and 9 parameters of the controller

2. A neuromuscular controller for humanoid robots - optimizing 16 parameters of the controller

3. A virtual neuromuscular controller for ATRIAS robot - optimizing 50 parameters of the controller

We detail our robots in the next subsection, followed by the controllers.

4.1.1 Robot morphologies tested

4.1.1.1 The ATRIAS Robot

Our test platform is CMU's ATRIAS robot (Figure 1.1), a human sized bipedal robot [53]. The ATRIAS robot was designed so that the inertial properties of the Center of Mass of ATRIAS matched that of humans. The robot weights about $64kg$, with most of its mass concentrated around the trunk. The torso is located about $0.19m$ above the pelvis, and its rotational inertia is about $2.2kgm^2$. The legs are 4-segment carbon-fiber linkages driven with a point foot, making the legs very light and enabling fast swing movements. The legs are actuated by 2 Series Elastic Actuators (SEAs) in the sagittal plane and a DC motor in the lateral plane. The SEAs consist of a fiberglass leaf spring attached to a geared DC motor on one end and the leg load on the other. Each spring is equipped with a load-side and motor-side encoder, which given the spring stiffness, can be used to estimate joint torques on each joint. Although ATRIAS is capable of 3D walking, in this work, we focus on planar movements around a boom.

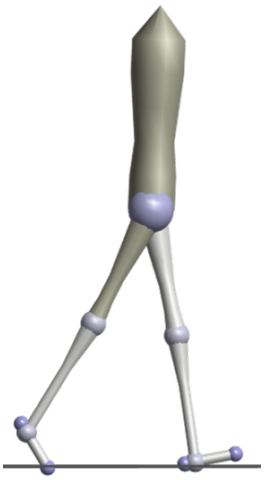


Figure 4.1: The 7-link planar biped model

4.1.1.2 A 7-link biped model

We also experiment with a 7-link planar human-like robot model as shown in Figure 4.1. It has two legs - composed of foot, shin and thigh and a hip. There are 6 actuators at the ankle, knee and hip joints in each leg with infinite allowed torque. This model is used as a simple 2-dimensional approximation to complex humanoid robots like Sarcos. In the future, we would like to experiment with more physically true models of such robots.

4.1.2 Controllers tested

4.1.2.1 Neuromuscular model for humanoid robots

We use neuromuscular model policies, as introduced in [54], as our controller for a 7-link planar human-like model. These policies use approximate models of muscle dynamics and human-inspired reflex pathways to generate joint torques, producing gaits that are similar to human walking in stance. [55] designed reflex laws for swing that enabled target foot-placement and leg clearance, by analyzing the double pendulum dynamics of the human leg. Integrating this swing control with the previous reflex control enables the model to overcome disturbances in the range of up to ± 10 cm [50].

Neuromuscular Stance Control

In stance, each leg is actuated by 7 Hill-type muscles [56], consisting of the

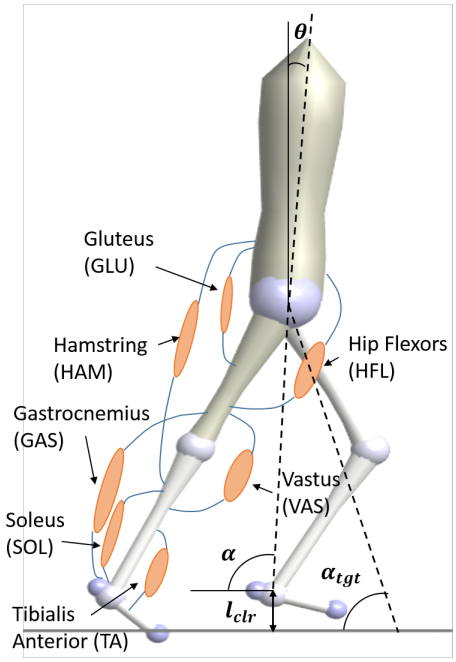


Figure 4.2: The neuromuscular model. The muscles and swing parameters are highlighted.

soleus (SOL), gastrocnemius (GAS), vastus (VAS), hamstring (HAM), tibialis anterior (TA), hip flexors (HFL) and gluteus (GLU), illustrated in Figure 4.2. Together, these muscles produce torques about the hip, knee and ankle. The muscle force F is a non-linear function of the muscle state s^m and stimulus S^m, which when multiplied by the moment arm $r(\theta_i)$ gives the resultant torque on joint i:

$$\tau_i^m = F(S^m, s^m) r(\theta_i),$$

where τ_i^m is the torque applied by muscle m on joint i and θ_i is the joint angle.

Most of the muscle reflexes in stance are positive length or force feedback on the muscle stimulus. In general, the stimulus $S^m(t)$ for muscle m is a function of the time delayed length or force signal P^m times a feedback gain K^m:

$$S^m(t) = S_0^m + K^m \cdot P^m(t - \Delta t),$$

where S_0^m is the pre-stimulus, K^m is the feedback gain and P^m is the time-delayed feedback signal of length or force. Some muscles can be co-activated and have multiple feedback signals from more than one muscle. The feedback gains

K^m described above are a subset of the parameters that we aim to tune in our optimization. The details of these feedback pathways can be found in [50].

This feedback structure generates compliant leg behaviour and prevents the knee from overextending in stance. To balance the trunk, feedback on the torso angle is added to the GLU stimulus:

$$S^{GLU}_{torso}(t) = K^{stance}_p(\theta_{des} - \theta) - K^{stance}_d\dot{\theta},$$

where K^{stance}_p is the position gain on the torso angle θ and θ_{des} is the desired angle. K^{stance}_d is the velocity gain and $\dot{\theta}$ is the angular velocity. Specifically, here are the stance parameters we optimize over, and their roles in the neuromuscular model:

- K^{GAS} : Positive force feedback gain on GAS
- K^{GLU} : Positive force feedback gain on GLU
- K^{HAM} : Positive force feedback gain on HAM
- K^{SOL} : Positive force feedback gain on SOL
- K^{TA}_{SOL} : Negative force feedback from SOL on TA
- K^{TA} : Positive length feedback on TA
- K^{VAS} : Positive force feedback on VAS
- K^{stance}_p : Position gain on feedback on torso angle
- K^{stance}_d : Velocity gain on feedback on torso velocity
- K^{GLU}_{mix} : Gain for mixing force feedback and feedback on angle for GLU

Swing Leg Placement Control

The swing control is controlled by three main components – target leg angle, leg clearance and hip control. Target leg angle is a direct result of the foot placement strategy which is a function of the velocity of the center of mass (CoM) v, and the as distance between the stance leg the CoM, and presented in [57]:

$$\alpha_{tgt} = \alpha_0 + C_d d + C_v v,$$

where α_{tgt} is the target leg angle, α_0 is the nominal leg angle, α_0, C_d and C_v are parameters optimized by our control.

Leg clearance is a function of the desired leg retraction during swing. The knee is actively flexed until the leg reaches the desired leg clearance height, l_{clr} and then held at this height, until the leg reaches a threshold leg angle. At this point, the knee is extended and allowed to reach the target leg angle α_{tgt}. Details of this control can be found in [55]. As was noted in [50], and observed in our experiments, the control is relatively insensitive to the individual gains of the set-up in swing. It is sufficient to control the higher level parameters such as the desired leg clearance and target leg angle.

The third part of the control involves maintaining the desired leg angle α_{tgt} by applying a hip torque τ^{α}_{hip}:

$$\tau^{\alpha}_{hip} = K_p^{swing}(\alpha_{tgt} - \alpha) - K_d^{swing}(\dot{\alpha}),$$

where K_p^{swing} is the position gain on the leg angle, K_d^{swing} is the velocity gain, α is the leg angle and $\dot{\alpha}$ is the leg angular velocity (see Figure 4.2).

More concisely, the swing parameters that we focus on in our optimization are the following:

1. K_p^{swing} : Position gain on feedback on leg angle
2. K_d^{swing} : Velocity gain on feedback on leg velocity
3. α_0 : Nominal leg angle
4. C_d : Gain on the horizontal distance between the stance foot and CoM
5. C_v : Gain on the horizontal velocity of the CoM
6. l_{clr} : Desired leg clearance

Though originally developed for explaining human neural control pathways, these controllers have recently been applied to robots and prosthetics, for example in [58] and [59]. As demonstrated in [50], these models are indeed capable of generating a variety of locomotion behaviours for a humanoid model - for example, walking on flat, rough ground, turning, running, walking upstairs and on ramps. However, a full study of using these models to control biped robots still needs to be done. Whether these models will transfer well to robots with significantly different dynamics and inertial properties than humans needs to be explored. It is difficult to transfer these models to robots because of a large number of interdependent gains that need to be tuned. Typically, this is done using Covariance Matrix

Adaptation Evolutionary Strategy (CMA-ES) [18], an evolutionary algorithm for difficult non-linear non-convex black-box optimization problems. Even though CMA-ES is useful for optimizing non-convex problems in high dimensions, it is not sample efficient and depends on the initial starting point. An optimization for 16 neuromuscular parameters takes 400 generations, around a day on a standard i7 processor and about 5,000 trials, as reported in [49].

The large number of trials make it impossible to implement CMA-ES on a real robot. This is a shortcoming because often we find that after training the policies in simulation, they do not transfer well to the real robot, due to differences between simulation and real hardware. We aim to overcome this problem by using Bayesian Optimization with informed feature transforms for optimizing controllers.

4.1.2.2 Feedback based reactive stepping policy

We design a parametrized controller for controlling the CoM height, torso angle and the swing leg by commanding desired ground reaction forces and swing foot landing location.

$$F_x = K_{pt}(\theta_{des} - \theta) + K_{dt}(\dot{\theta}_{des} - \dot{\theta}) \tag{4.1}$$

$$F_z = K_{pz}(z_{des} - z) + K_{dz}(\dot{z}_{des} - \dot{z}) \tag{4.2}$$

$$x_p = k(v - v_{tgt}) + C \cdot d + 0.5 \cdot v \cdot T \tag{4.3}$$

Here, F_x is the desired horizontal ground reaction force (GRF), K_{pt} is the proportional gain on the torso angle θ and K_{dt} is the derivative gain on the torso angular velocity $\dot{\theta}$. θ_{des} and $\dot{\theta}_{des}$ are the desired torso lean and desired torso angular velocity. F_z is the desired vertical GRF, K_{pz} is the proportional gain on the CoM height z and K_{dz} is the derivative gain on the CoM vertical velocity $\dot{z}$. z_{des} and $\dot{z}_{des}$ are the desired CoM height and desired CoM vertical velocity. Both $\dot{\theta}_{des}$ and $\dot{z}_{des}$ are always set to 0. x_p is the desired foot landing location for the end of swing; v is the horizontal CoM velocity, k is the feedback gain that regulates v towards the target velocity v_{tgt}. These quantities are highlighed in Figure 4.3. C is a constant and d is the distance between the stance leg and the CoM; T is the swing time and the term $0.5 \cdot v \cdot T$ is a feedforward term similar to a Raibert hopping policy [60].

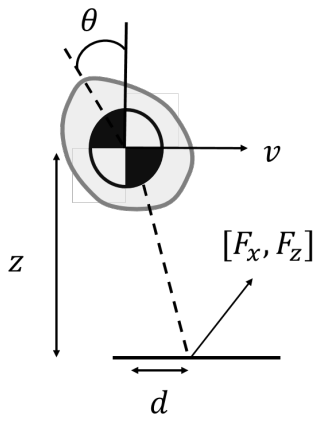


Figure 4.3: Illustration of the Center of Mass model and variables used for the feedback-based stepping policy.

This parametrization results in desired ground reaction forces (GRFs) in stance and a desired foot landing position in swing. Note that there is no desired CoM trajectory in this controller. We only command a desired CoM velocity, regulated through the swing foot placement. In stance, the desired GRFs regulate a desired CoM height and torso angle. These are then sent to the ATRIAS inverse dynamics model that generates desired motor torques (τ_f, τ_b) that realize the GRFs.

$$M\ddot{q} + h = S\tau + J^T F \tag{4.4}$$

where M is the mass matrix, $\ddot{q}$ are the joint accelerations, h are the gravitational terms, S is a selection matrix, τ are resultant joint torques, J is the contact Jacobian and $F = [F_x, F_z]$ are the desired forces generated in 4.1. There are no Coriolis terms due to the assumption of massless legs. Details can be found in [61]. These desired motor torques are then sent to a low level motor velocity-based feedback loop that generates the desired torques in the robot SEAs (Fig 4.4).

In swing, we continually re-generate a 5th order spline that starts from the current position and velocity of the swing leg, x_{sw} and $\dot{x}_{sw}$ and terminates at the desired foot position x_{fp}, with ground speed matching (swing leg is at rest with respect to the ground). The desired initial and final point of this spline is continually regenerated based on the current estimate of the swing foot position and velocity, as well as the desired footstep and CoM velocity. This trajectory gives the next desired position and velocity of the swing leg, x^*_{sw}, $\dot{x}^*_{sw}$, which is

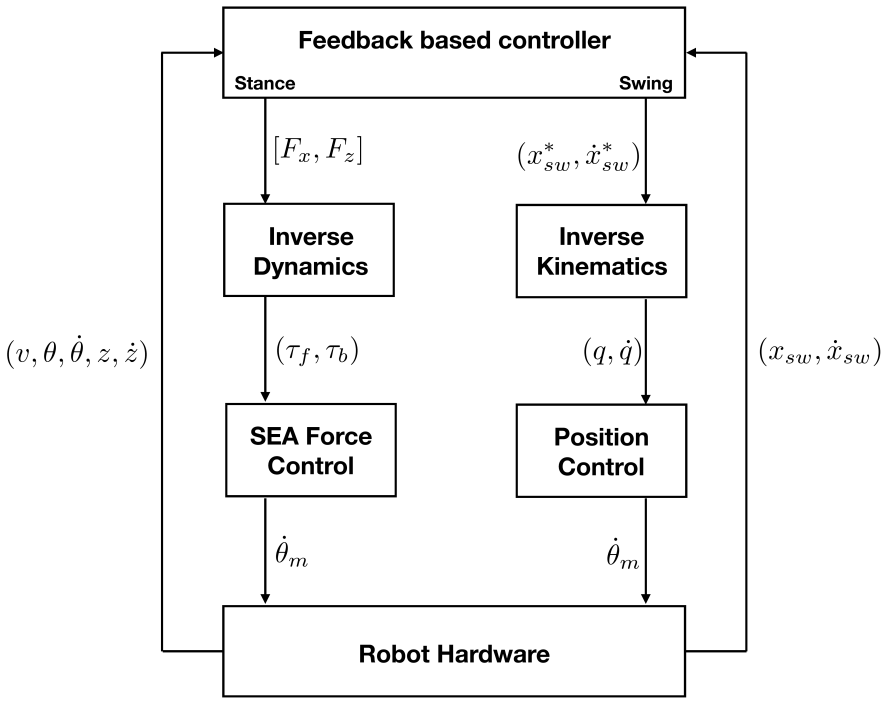


Figure 4.4: ATRIAS control flowchart with variables from Section 4.1.2.2

translated to desired joint positions and velocities using the robot kinematics. These are then position-controlled by sending a velocity command to the robot SEAs (Fig 4.4).

This controller assumes no double-stance, swing leg takes off as soon as stance is detected. This leads to a highly dynamic gait, as the contact polygon for ATRIAS in single stance is a point. It should be possible to extend this walking controller to include a double stance phase, but we leave that to future work. The controller also depends on the desired speed of walking (as this determines the next stepping location). This means that the "stability" of the controller depends not only on the parameters chosen, but also the desired target speed. We assume that the target speed is provided by the user and is constant in our experiments. It is possible to add the desired speed as an extra dimension to the controller and optimize controllers conditioned on this. But we leave that for future work.

1. 5 dimensional walking controller : In our first set of experiments, we optimized 5 parameters from the above described controller. These were $[K_{pt}, K_{dt}, k, C, T]$. The desired positions and velocities were hand tuned, and so was the feedback on z.

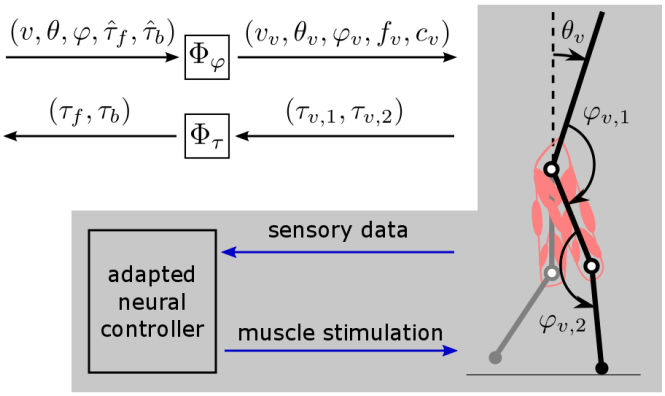


Figure 4.5: Virtual neuromuscular control. VNMC maps the robot's state, $(v, \theta, \varphi, \hat{\tau}_f, \hat{\tau}_b)$, to virtual measurements required to emulate a neuromuscular model, $(v_v, \theta_v, \varphi_v, f_v, c_v)$, where φ are joint angles, and f_v and c_v are force and contact data of the virtual leg. The virtual neuromuscular model (in the gray box) outputs virtual joint torques, $(\tau_{v,1}, \tau_{v,2})$, that are mapped to desired robot joint torques, (τ_f, τ_b), which are tracked by the SEA controller.

2. 9 dimensional walking controller : In our second set of experiments, we optimized 9 parameters of the above described controller. They were : $[K_{pt}, K_{dt}, \theta_{des}, K_{pz}, K_{dz}, z_{des}, k, C, T]$

4.1.2.3 Virtual Neuromuscular Controller for ATRIAS

We adapt a previously proposed virtual neuromuscular controller (VNMC) [62] adapted for ATRIAS. VNMC maps a neuromuscular model to the robot's topology and emulates it to generate desired motor torques, which is sent to the SEA controller (Figure 4.5). The emulated neuromuscular model, which is originally developed to study human locomotion, consists of primarily spinal reflexes, and with appropriate sets of control parameters, it generates diverse human locomotion behaviors [50], such as walk, run,turn, walk up stairs, etc. The VNMC control consists of 6 muscles on each leg - the hip flexors (HFL), the glutei (GLU), the hamstrings (HAM), the rectus femoris (RF), the vastii (VAS) and the biceps femoris (BFSH). Since ATRIAS doesn't have any ankles, these muscles are removed from the model. The feedbacks are similar to the ones described in 4.1.2.1, and details can be found in [50].

For this study, we adapt the previous VNMC [62] by removing some unnecessary biological components while preserving its basic functionalities. First, the new VNMC directly uses joint angular and angular velocity data instead of esti-

mating it from physiologically plausible sensory data, such as muscle fiber states, when applicable. Second, most of the neural transmission delays are removed, except the ones utilized by the controller.

The adapted VNMC consists of 50 control parameters including feedback gains for each muscle feedback in swing and stance, pre-stimulations for each muscle in swing and stance, high level leg placement gains and desired trunk inclination. When optimized using covariance matrix adaptation evolution strategy [18], it can control ATRIAS to walk on rough terrains with height changes of ± 20 cm in planar simulation. The original VNMC in [62] doesn't start from rest, which is impossible to achieve on a robot. To overcome this problem, for now, we start with a 5-dimensional walking controller that can start from rest. Once a desired speed if reached, we switch to the virtual neuromuscular control.

4.2 Proposed work

4.2.1 A 3 dimensional walking controller

We are working on a 3-dimensional walking controller for ATRIAS as well as a humanoid robot such as Sarcos. We would like to experiment with both model based and model-free 3-dimensional controllers.

Neuromuscular controller A 3-dimensional neuromuscular controller was proposed in [50] for human walking. We would like to adapt it for robot control, using techniques described in 4.1.2.3 and apply it to both ATRIAS and a humanoid robot, like Sarcos. For ATRIAS, we will have to do a virtual mapping of robot joints to a human model, similar to what was done in [62].

Reactive stepping controller [3] develops a 3-dimensional stepping policy for ATRIAS. We would like to incorporate this into our current controller on ATRIAS. Since there is no yaw control, we cannot expect to walk in a straight line, but ATRIAS is capable of walking stably in 3-dimensions [3]. With stance feedback on the CoM height, and torso roll and pitch, we can easily extend our current stance control to 3-dimensions. With a stabilizing swing foot placement, this controller should walk stably in 3-dimensions and track commanded desired velocities.

CHAPTER 5

Results and evaluation

We now present work completed so far on the controllers and feature transforms presented in the previous sections.

5.1 Using the DoG Transform

5.1.1 Hardware experiments with the 5-dimensional controller

The first set of hardware experiments were conducted on a 5-dimensional controller, described in Section 4.1.2.2 on the ATRIAS robot. The target speed profile for these experiments was $0.4m/s$ (15 steps)$-1.0m/s$ (15 steps)$-0.2m/s$ (15 steps)$-0m/s$ (5 steps). The total number of steps before the controller shut off were 50. The cost function that was optimized was:

$$cost = \begin{cases} 100 - x_{fall}, \text{if fall} \\ ||v_{avg} - v_{tgt}||, \text{if walk} \end{cases} \tag{5.1}$$

where x_{fall} is the distance covered before falling, v_{avg} is the average speed per step and v_{tgt} is the target velocity profile.

We sampled 100 random points on hardware and 10 of them walked for this profile. This means that random sampling has a 1/10 chance of sampling a good point. In simulation, 276 points out of 1000 randomly sampled points walked, implying a 1/4 success rate. This highlights the difference between hardware and

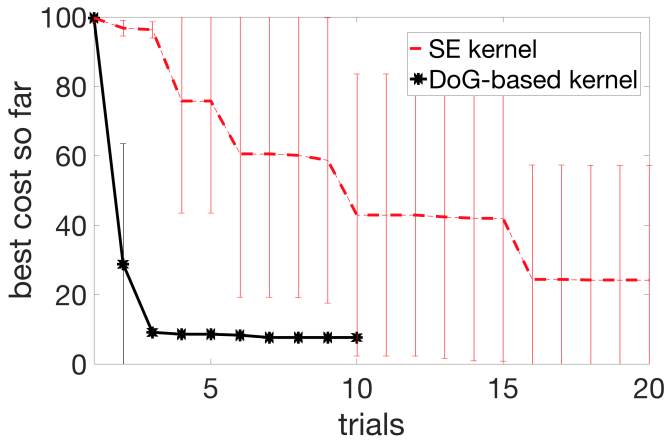


Figure 5.1: BO for 5 dimensional controller on ATRIAS robot hardware. BO with SE finds walking points in 4/5 runs in 20 trials. BO with DoG-based kernel finds walking points in 5/5 runs in 3 trials.

simulation, making this a tougher problem on hardware.

On hardware, we conducted 5 runs of each – BO with DoG-based kernel and BO with SE, 10 trials for DoG-based kernel per run, and 20 for SE kernel. In total, this led to 150 experiments on the robot (excluding the 100 random samples). We also experimented with using fixed vs automatically learned hyperparameters for both kernels. A simple choice of fixed hyperparameters worked well for DoG-based kernel, while for SE kernel it was better to learn these automatically. DoG scores were calculated on 20,000 points in simulation, by running $3.5s$ long simulations with target speed of $0.5m/s$.

BO with DoG-based kernel found walking points in 3 trials in 5/5 runs. BO with SE found walking points in 10 trials in 3/5 runs, and in 4/5 runs in 20 trials. These results can be seen in Figure 5.1.

5.1.2 Hardware experiments with the 9 dimensional controller

The second set of hardware experiments were conducted on a 9-dimensional controller, described in Section 4.1.2.2 on the ATRIAS robot. The target speed profile for these experiments was $0.4m/s$ (30 steps). The total number of steps before the controller shut off was 30. The cost optimized was the same cost as in Equation 5.1.

We sampled 100 random points on hardware and 3 of them walked for this

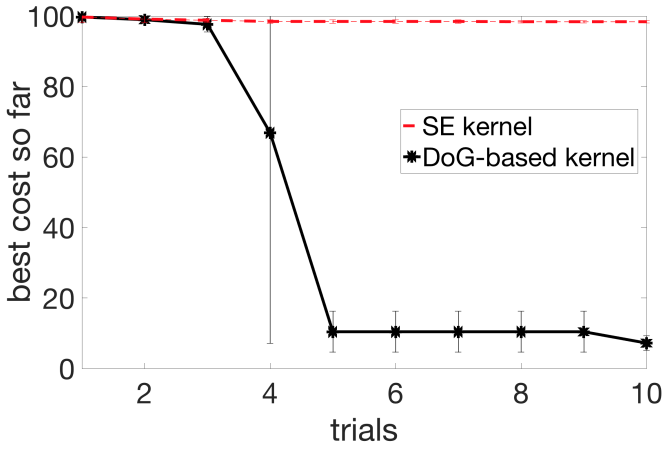


Figure 5.2: BO for 9 dimensional controller on ATRIAS robot hardware. BO with SE doesn't sample any walking points in 3 runs. BO with DoG-based kernel finds walking points in 5 trials in 3/3 runs.

speed profile. This means that random sampling has a 1/33 chance of sampling a good point. On the original speed profile from Section 5.1.1, the number of successful points out of 100 were 0, implying a less than 1% success rate. To keep the problem at hand reasonable, we used the simpler target speed profile. In comparison, the success rate in simulation is 8% for the tougher profile, implying a greater mismatch between hardware and simulation than the 5-dimensional controller.

For this setting, we conducted 3 runs of each BO with DoG-based kernel and BO with SE, 10 trials for DoG-based kernel per run, and 10 for SE. In total, this led to 60 experiments on the hardware (excluding the random sampling).

BO with DoG-based kernel found walking points in 5 trials in 3/3 runs. BO with SE did not find any walking points in 10 trials in all 3 runs. These results can be seen in Figure 5.2.

Based on these results, we concluded that BO with DoG-based kernel was indeed able to extract useful information from simulation and speed up learning on hardware. However, it took slightly longer to find points for the 9-dimensional controller and the quality of solution might have improved further if we sampled further. This is all as per expectation, as a higher dimensional controller should take longer to optimize, but eventually lead to a better solution than a smaller dimensional controller.

Figure 5.3: A time lapse of ATRIAS walking around the boom during a run of DoG-based kernel.

5.1.3 Simulation experiments with the 9-dimensional controller

To facilitate further experiments we used ATRIAS simulator [63] with modeling disturbances and different target speed profiles. For these experiments, we use the 9-dimensional controller described in 4.1.2.2 as this proved to be a more challenging setting for the ATRIAS hardware. Masses of the robot torso, legs, the boom, as well as inertia of the torso were perturbed randomly by up to 15% of their original values. This ensured a mismatch between the setting used to generate the kernel and the experimental setting for evaluating its performance, aimed at capturing the discrepancy between hardware and simulation. Note that the kernel was generated on the unperturbed setting, with parameters as described in Section 4.1.1.1 for a target speed of $0.5m/s$. The grid size for DoG scores was 100,000 points, and simulations were run for $5s$.

The cost used for these experiments was

$$cost = \begin{cases} 100 - x_{fall}, \text{if fall} \\ ||v_{avg} - v_{tgt}|| + c_{tr}, \text{if walk} \end{cases} \tag{5.2}$$

where x_{fall} is the distance covered before falling, v_{avg} is the average speed per step, v_{tgt} is the target velocity for that step, and c_{tr} captures the cost of transport - calculated by taking a sum of motor torques and normalizing them by a constant. Simulations for evaluating the cost were run for $30s$. Note the addition of the cost of transport in this cost, as compared to 5.1. c_{tr} needs more than 10 trials to be optimized significantly, and the current low-level motor controllers in Section 4.1.2.2 are not designed to reduce c_{tr}. Hence, its not considered in the hardware

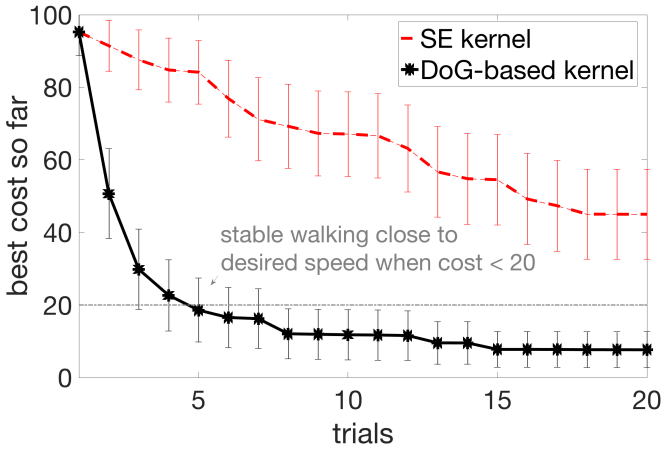


Figure 5.4: Further experiments using ATRIAS simulator: BO for "speed-up-down" target speed profile on robot model with mass and inertia differences (mean over 50 runs; 95% confidence intervals).

experiments. Figure 5.4 illustrates BO on a simulated model with mass and inertia differences. The target was to start walking at 0.4m/s, then speed up to 0.6m/s, then 1.0m/s, slow down to 0.6m/s, then walk at 0.2m/s. DoG-based kernel was collected using an unperturbed model with a target speed of 0.5m/s, and yet it performed very well on this more challenging setting, with the top speed two times of the speed on which the kernel was collected. After 20 trials, 96% of BO runs using the DoG-based kernel found a stable walking solution, compared to 56% of the runs using an SE kernel. The average cost of the walking solutions was also improved: lower by ≈30% when using DoG vs SE kernel.

These experiments suggest that DoG-based kernel is able to offer improvement for the settings different from the one used to generate it. This improvement is robust to both the deviations of the robot model/hardware parameters as well as desired walking speed profiles.

5.1.4 Simulation experiments with the 50-dimensional controller

Next set of experiments were using the VNMC 50-dimensional controller, as described in Section 4.1.2.3 on the ATRIAS simulation.

VNMC does not start from rest, and needs an initial velocity. In previous work this has been emulated by either giving simulations initial speeds, or by giving a push. These are either un-realizable or unreliable on hardware. To overcome this problem, we start the VNMC with a 5-dimensional walking controller (described

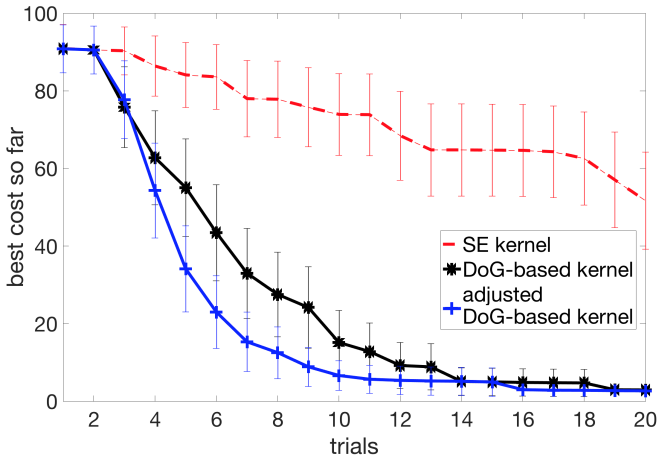


Figure 5.5: BO with Virtual Neuromuscular Controller.

in Section 4.1.2.2, parameters hand-tuned and fixed). Once the robot has taken 10 steps with this controller, the control is switched to VNMC.

To construct DoG-based kernel for this controller we collected 250,000 points from 7-second simulations. The DoG scores were computed after switching to the VNMC (so after first 10 steps). Searching in 50 dimensional space could be completely intractable if the search region is too large. Usually enough domain knowledge is available to confine the search to a reasonably manageable region. We tried to hand-tune an initial point that walked 3-4 steps before falling in simulation (this point still had a very high cost of ≈93). This point became the "center" of our search space, and we searched in a hyper-cube of size $[0.75, 1.25]$ in each dimension around this point. So, with initial point x_0, the search space was $[0.75 \cdot x_0, 1.25 \cdot x_0]$. With these boundaries, 4% of points sampled randomly were walking.

Figure 5.5 shows results of Bayesian Optimization with SE, DoG-based kernel and adjusted DoG-based kernel with mismatch information. During optimization simulations were run for 30 seconds. We used the same cost function as described in the previous section (equation 5.2).

In 50-dimensional control, the mismatch between long and short simulations becomes apparent. This raises concerns for how useful DoG-based kernel would be on hardware, since it tries to infer the quality of the controller parameters from short simulations. For the 5-dimensional and 9-dimensional controllers, the performance during short simulations usually predicted whether $30s$ simulations would be successful. That is, points that walk for 5s would usually walk for 30s.

However, this is not true for the 50-dimensional controller. Since this controller is capable of much richer behaviors, if a point is not in a limit cycle before the end of a short simulation, it can lead to a range of behaviours later. As a result, we noticed an improvement when using adjusted DoG-based kernel described in Section 3.2. While DoG is still very competitive and finds walking points in 100% of the runs by 20 trials, the adjusted DoG with mismatch has an advantage. It reaches the same optimum found by DoG faster.

The 50-dimensional controller has not been fully implemented to work on hardware yet due to lack of time. However, our experiments on other controllers so far seem promising and we are working towards a hardware implementation. To anticipate potential mismatch between simulation and hardware, we tested it on slightly perturbed initial conditions for the VNMC. The different conditions were aimed to replicate issues likely to be seen on hardware. The starting states for VNMC would differ slightly each time, since they would depend on the state of the robot after the 5-dimensional initiating controller has finished. Both DoG and adjusted DoG were robust to slight changes in initial condition.

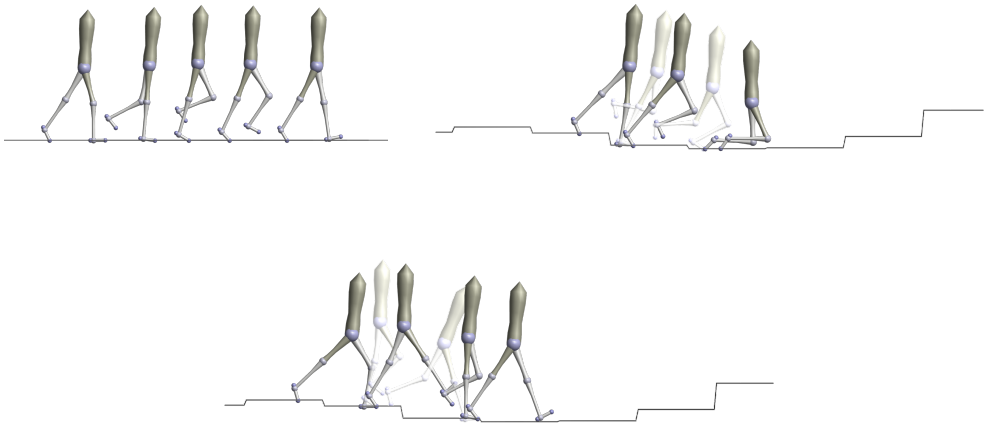
It would be interesting to study the effect of a mismatch map on performance on hardware. While essentially a mismatch map adds more parameters to learn, which would result in a slower optimization, if initialized properly with domain knowledge, it might help. The DoG-based kernel samples a few unstable points in the start of the 50-dimensional optimization, as these have high DoG scores on short simulations, but still might fall in longer simulations under disturbances. Since a lot of the 50-dimensional space is clustered around each DoG score, its an over-condensation of the space. A lot of potentially good points get rejected because one of the points falls. If we detect a mismatch at the high DoG point that fell, and effectively increase its distance to other high DoG points, we will help speed up the optimization.

We plan to test adjusted DoG-based kernel more in the future in this challenging setting that could be sensitive to simulation-hardware mismatch.

5.1.5 Simulation experiments on a 16-dimensional controller

To make sure that our feature transform generalizes to multiple robot morphologies, we also did experiments on the 7-link biped, described in Section 4.1.1.2.

Figure 5.6: Top row: a policy that generates successful walking on flat ground could fail on rough ground. Bottom row: optimization on rough ground finds policies that walk, even though pre-computation for DoG kernel is done using unperturbed model on flat ground.



We conduct our experiments on models with mass and inertial disturbances and on different ground profiles. After collecting the DoG scores, we perturb the mass of each link, inertia and center of mass location randomly by up to 15% of the original value. For mass/inertia we randomly pick a variable from a uniform distribution between $[-0.15,\ 0.15]\cdot M$, where M is the original mass/inertia of the segment. Similarly we change the location of the center of mass by $[-0.15,\ 0.15]\cdot L/2$, where L is the length of the link. These disturbances are different for each run of our algorithm, hence we test a wide range of possible modelling disturbances. For the ground profiles, we generate random ground height disturbances of upto $\pm 8cm$ per step.

To ensure that our approach can perform well across various cost functions, we conduct experiments on two different costs, constructed such that parameter sets achieving low cost also achieve stable and robust walking gaits. The first cost function varies smoothly over the parameter space:

$$cost = \frac{1}{1+t} + \frac{0.3}{1+d} + 0.01(v - v_{tgt}), \tag{5.3}$$

where t is seconds walked, d is the final CoM position, v is mean velocity and v_{tgt} is the desired walking velocity (from human data). This cost encourages walking further and for longer through the first two terms, and penalizes deviating from the target velocity with the last.

The second cost function is a slightly modified version of the cost used in [50] for experiments with CMA-ES. It penalizes policies that lead to falls in a non-smooth manner:

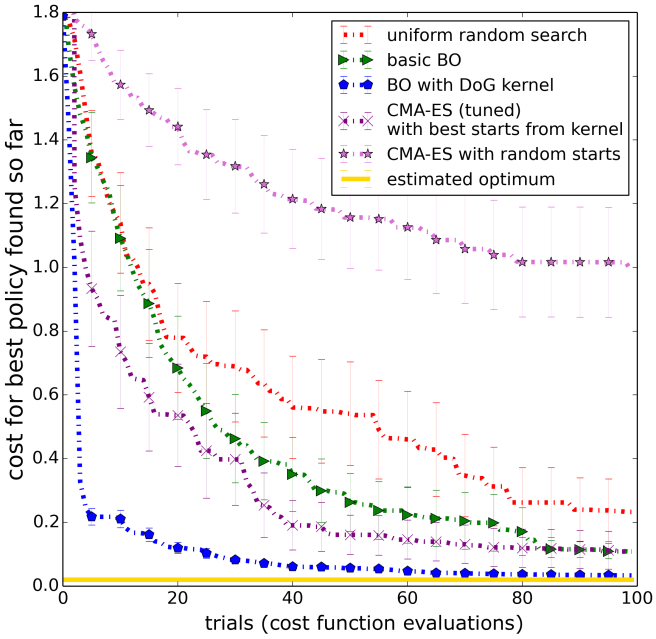
$$cost_{CMA} = \begin{cases} 300 - x_{fall}, \text{if fall} \\ 100||v_{avg} - v_{tgt}|| + c_{tr}, \text{if walk} \end{cases} \tag{5.4}$$

Here x_{fall} is the distance travelled before falling, v_{avg} is the average speed in simulation, v_{tgt} is the target speed and c_{tr} is the cost of transport calculated using muscle activations. The first term directly penalizes policies that result in a fall, inversely to the distance walked. If the model walks for the whole simulation time, the cost is lower, ensured by the constants, and encourages policies that result in lower cost of transport and walk at target velocity. Since we have the same set of gains for left and right legs, the steadiness cost of the original cost [50] was unimportant. So, we removed that term and focused on the first and second conditions in the optimization.

In the following sections we compare the performance of several baseline and state-of-the-art optimization algorithms in simulation. Motivated by the discussion in [39], we include the baseline of uniform random search. While this search is uninformed and not sample-efficient, it could (perhaps surprisingly) serve as a competitive baseline in non-convex high-dimensional optimization problems. Theoretically – it provides statistical guarantees of convergence, and practically – it can outperform informed algorithms as well as grid search on high-dimensional problems (see Section 2 in [39] for further discussion).

We also provide comparisons with CMA-ES [18] and Bayesian Optimization with a Squared Exponential kernel (basic BO). Since we were optimizing a non-convex function in a 16D space, it was not feasible to calculate the global minimum exactly. To estimate the global minimum for the costs we used, we ran CMA-ES (until convergence) and BO with our domain kernel (for 100 trials) for 50 runs without model disturbances on flat ground. When reporting results, we plot the best results found in this easier setting as the estimated optimum for

Figure 5.7: Experiments using DoG kernel on rough ground with model disturbances on the smooth cost over 50 runs. Basic BO line (green triangles) was obtained using BO with a Euclidean distance kernel. Estimated optimum (yellow line) was obtained on the unperturbed setting. BO with DoG-based kernel kernel (blue pentagons) used the DoG feature transform and was substantially more sample-efficient than the alternative approaches.



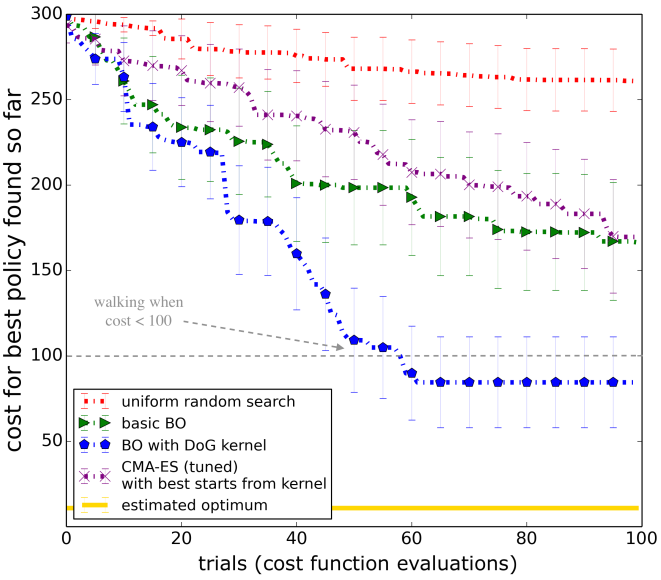
comparison.

All the experiments described below are done for 50 independent runs, each with a unique set of modeling disturbances and a different ground profile for rough ground walking. Each run consists of 100 trials or cost function evaluations, in which the optimization algorithm evaluates a parameter set for 100 seconds of simulation. Note that the disturbances and ground profiles remain constant across each run (100 trials).

5.1.5.1 Experiments on the smooth cost function

Figure 5.7 shows results of our experiments using the DoG-based kernel kernel on the smooth cost. Policies with costs of less than $\approx$0.15 corresponded to robot model walking on rough ground with $\pm 8cm$ disturbance. For BO with DoG-based kernel kernel, 25-30 cost function evaluations were sufficient to find points that corresponded to robot model walking on a randomly generated rough ground. This is in contrast to basic BO that did not find such results in under 100 trials.

Figure 5.8: Experiments using DoG-based kernel kernel on rough ground with model disturbances on the non-smooth cost over 50 runs. Policies with costs below 100 generate walking behaviour for 100 seconds in simulation. None of the optimization methods find optimal policies in all the runs and hence the mean cost is higher than the estimated optimum.



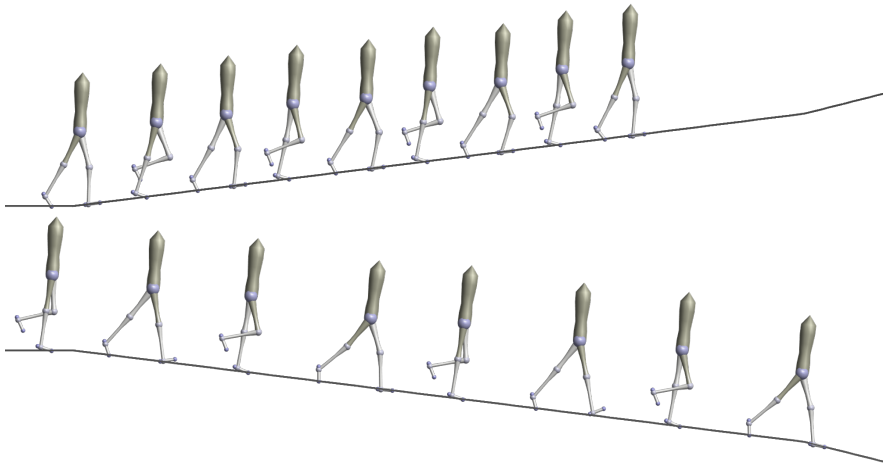
To let CMA-ES also benefit from the kernel, we started each run from one of the best 100 points for the DoG-based kernel kernel. After tuning the σ parameter of CMA-ES to make it exploit more around the starting point, we were able to find policies that resulted in walking on rough ground after 65-70 cost function evaluations on most runs. On the other hand, CMA-ES starting from a random initial point was not able to find walking policies in 100 evaluations.

These results suggest that DoG scores successfully captured useful information about the parameter space and were able to effectively focus BO and CMA-ES on the promising regions of the policy search space.

5.1.5.2 Experiments with the non-smooth cost

We observed good performance on the non-smooth cost function (Figure 5.8), though it was not as remarkable as the smooth cost. BO with kernel still outperformed all other methods by a margin, but this different cost seems to hurt BO and CMA-ES alike. Since this cost is discontinuous, there is a huge discrepancy between costs for parameters that walk and those that don't. If no walking policies are sampled, BO learns little about the domain and samples randomly, which

Figure 5.9: Optimized policies walking up and down a 12.5% ramp.



makes it difficult to find good parameters. Hence not all runs find a walking solution. BO was able to find successful walking in 74% of cases on rough ground with $\pm 6cm$ disturbance in less than 60 trials/evaluations. CMA-ES starting from a good kernel point was able to do it in 40% of runs.

This showed that our kernel was indeed independent of the cost function to an extent, and worked well on two very different costs. We believe that the slightly worse performance on the second cost is because of the cost structure, rather than a kernel limitation, as it still finds walking solutions for a significant portion of runs.

5.1.5.3 Experiments on different terrains

We also optimized on ramps – sloping upwards, as well as downwards. The ramp up and down ground slopes were gradually increased every $20m$, until the maximum slope was reached. The maximum slopes for going down and going up were 20% ($tan(\theta) = 0.2$). BO with DoG kernel was able to find parameters that walked for 100 seconds in 50% of cases in ramp up and 90% in ramp down. Example optimized policies walking up and down slope are shown in Figure 5.9.

We believe the reason we could not find walking policies on ramps in all runs, was that we are not optimizing the hip lean, which was noted to be crucial for this profile in [50]. Since we did not consider this variable when generating our 16 dimensional kernel, it was not trivial to optimize over it without re-generating the grid. Similarly, we found that we could not find any policies that climbed up

stairs. Perhaps this could be achieved when optimizing over a much larger set of parameters, as in [50].

To test if the hip lean indeed helps climb up a ramp, we hand-tuned the hip lean to be 15°, instead of the original angle of 6° for which the kernel was generated. Indeed, our rate of success on walking on ramp up ground profile increased from 50% to 65%. For walking upstairs, we achieved 10% success. This shows that the hip lean indeed helped walking on these terrains, and ideally we would like to optimize it along with the other parameters. Also, it shows that the DoG-based kernel kernel is robust to changes in parameters that were used to generate it. This is an important property, as parameters in the neuromuscular model are changed slightly for different experiments, for example the ground stiffness, initial conditions of the model, etc. If the kernel results hold across a variety of such conditions, we don't need to regenerate it.

In the future, we would like to include more variables for optimizing over different terrains, and include them as part of the kernel.

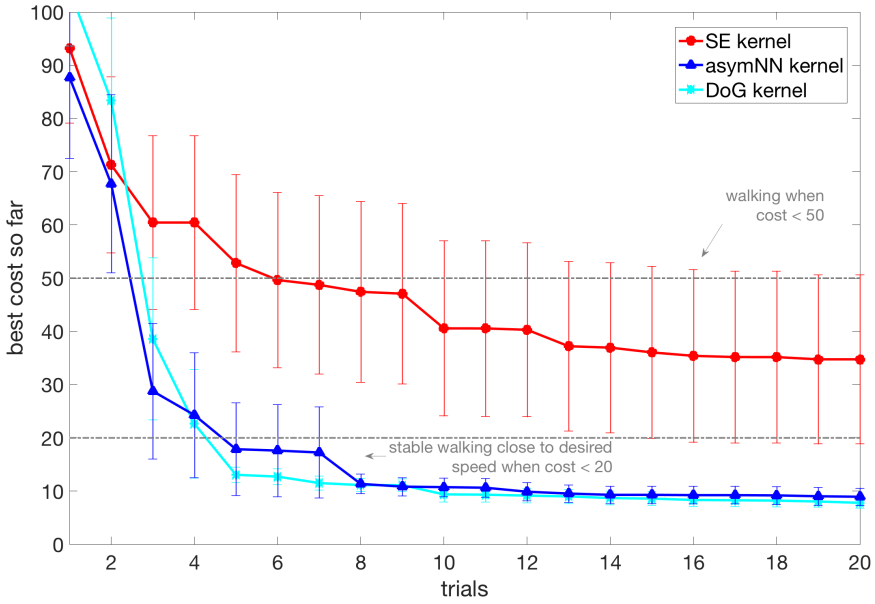
5.2 Using the NN transform

In this section we describe our experiments with cost-based and trajectory-based neural network kernels. We first consider the setting of optimizing a 5-dimensional controller for the ATRIAS robot. We show that the cost-based kernel is able to improve sample efficiency over standard Bayesian Optimization. We present hardware experiments to demonstrate that our kernel allows obtaining a set of parameters close to optimal on the second trial. We then discuss simulation experiments with a 16 dimensional controller that utilizes a Neuromuscular model [50]. These experiments show that our trajectory-based kernel is able to significantly outperform standard Bayesian optimization for a higher-dimensional controller even when a sharply discontinuous cost is used during optimization.

5.2.1 Experiments with Raibert controller on the ATRIAS robot

For our experiments on the ATRIAS robot we used a high-fidelity ATRIAS simulator [63] to generate the data for learning the network. We did an initial analysis of the performance of our approach in simulation, followed by hardware

Figure 5.10: Initial tests in simulation.

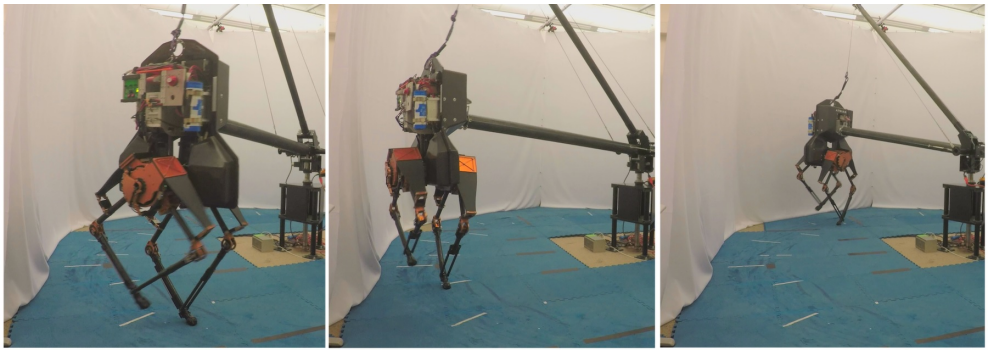


experiments. We constructed a distance metric for a kernel used in Bayesian Optimization by training a neural network to reconstruct cost obtained from short simulations. We created a sobol grid on the input parameter space with 20000 points and ran short 3.5 second simulations on each of the corresponding 20000 parameter sets to compute the costs. We then used a fully connected network with 4 hidden layers (128, 64, 16, 4 units) with L1 loss to reconstruct the transformation of the cost described in section 3.1.2.1.

In Figure 5.10 we first compare the performance of BO that used our neural network kernel (*asymNN*) versus using a standard Squared Exponential kernel (*SE*) in simulation. For these experiments we used the cost in Equation 3.6, Section 3.1.2.1 with a target velocity of $1m/s$. Simulations with cost less than 50 yielded walking behavior, while cost less than 20 resulted in a stable walk close to the desired speed. BO with *asymNN* kernel was able to reliably find points corresponding to stable walking behavior in only 8 trials. In contrast, BO with *SE* kernel did not find stable walking solutions in the first 20 trials reliably. We also compare with the DoG-based kernel kernel. *asymNN* is able to closely match the performance of DoG-based kernel in this setting after 8 trials.

After experiments in simulation suggested that *asymNN* kernel can yield a significant improvement in sample efficiency of BO, we conducted a set of ex-

Figure 5.11: ATRIAS during BO with *asymNN* kernel.



periments on the ATRIAS robot. We completed 6 sets of runs of BO: 3 using *asymNN* kernel and 3 using a standard *SE* kernel with 10 trials each, leading to a total of 60 hardware experiments. Since ATRIAS walks around a rather short boom in 2D, walking at high speeds needs a lot of torque from the robot motors. This means higher lateral forces between the robot and the boom, which do not affect our direction of motion but can lead to a lot of internal forces, eventually breaking the robot. Since the robot was already damaged in some parts, in our first attempt, we tried to start with lower speeds of $0.4m/s$ so that we could do hardware experiments and analyze the validity of our approach on hardware without breaking the robot too often. Setting a lower target speed yielded a problem with more stable walking points: they comprised approximately $\frac{1}{6}$ of the parameter space (measured by sampling 100 random points, 16 of which walked). So we anticipated it would be challenging to improve over BO with *SE* kernel, since it was already able to find stable walking solutions after only 3-4 trials. Another feature of the *asymNN* kernel was that it was biased towards sampling points that walk in simulation. Hence, out of 10 trials, it is more likely to sample stable points even in hardware (Figure 5.13), as compared to SE. This is desirable as stable points are less likely to break the robot.

Figure 5.12 shows the performance of BO with SE versus *asymNN* kernel. To reduce variability in hardware experiments we use one random trial at the start of each run, with a point selected among those with high cost in simulation. SE obtains a stable walking solution on the 3rd trial in one run, and on the 4th trial in the two other runs. *asymNN* kernel is able to find the best-performing set of parameters on the second trial in each of the 3 runs. This confirms that using *asymNN* kernel offers an improvement over using *SE* kernel in this setting. We

Figure 5.12: Best cost so far during BO (mean over 3 runs).

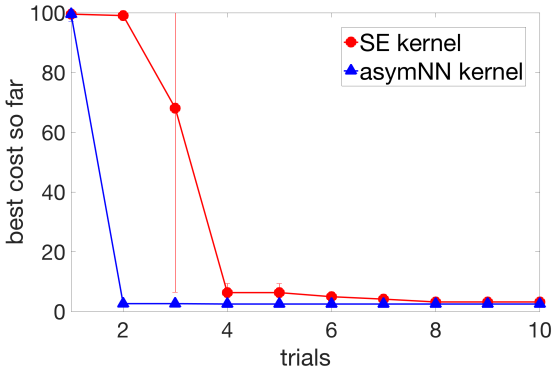
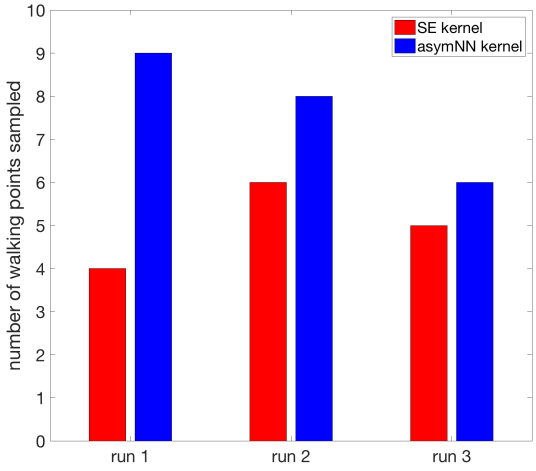


Figure 5.13: Number of "walking" points sampled.



suggest that *asymNN* reliably selects an excellent point on the 2nd trial because such points lie far from poorly performing subspace of parameters (under the distance metric constructed with *asymNN*).

While in our hardware setup most methods are likely to sample walking points within 10 trials, we believe our experimentation is an important step towards optimizing locomotion policies for complex humanoid robots. Bayesian Optimization studies in the past have also used real robot hardware, for example, [4], [6] and [37]. However, [6, 36, 37] used robots which are statically stable for significant parts of their gait, making discontinuities in the cost function landscape less likely and in turn making the optimization easier. On the other hand, ATRIAS is a complex bipedal system which is likely to fall with unstable controllers due to point feet. [4] use a walking robot similar to ours. However, their controller parametrization is very different, and not widely used, unlike our inverse dynam-

ics and force-based controller which is more modern and state-of-the-art [2], [64], [1]. Hence, even if our policy parametrization was chosen so that steady walking points could be found in a few trials, our testbed is fairly complex and our problem formulation is widely applicable.

5.2.2 Simulation experiments with the 16-dimensional Neuromuscular Model

In section 3.1.2.2 we introduced a cost-agnostic approach for constructing an informed kernel from simulations. Our approach is to train a neural network to reconstruct trajectory information. Here we describe our experiments with 16-dimensional controller of the Neuromuscular model. We created a grid of 100K points in the input parameter space and ran short 5 second simulations on each of the corresponding 100K parameter sets to collect the trajectory summaries. We then used a fully connected network with 4 hidden layers (512, 128, 32 units) with L1 loss to reconstruct the summaries of the trajectories (as described in section 3.1.2.2). This transformation induced by the neural network was used as a re-parameterization from the input space of 16-dimensional controller parameters to 8-dimensional space of trajectory summaries. These 8-dimensional outputs of the neural network define kernel distances in the informed kernel (*trajNN*). All experiments were conducted on perturbed models, as described in Section 5.1.5.

Figures 5.14, 5.15 illustrate our experiments comparing using *trajNN* versus using Squared Exponential (*SE*) kernel for BO. To analyze the performance of *trajNN* on different cost functions we conducted experiments on two different costs suggested in prior literature. The first cost promotes walking further and longer before falling, while penalizing deviations from the target speed [49]:

$$cost_{smooth} = 1/(1+t) + 0.3/(1+d) + 0.01(v - v_{tgt}), \tag{5.5}$$

where t is seconds walked, d is the final hip position, v is mean velocity and v_{tgt} is the desired walking velocity ($1.3m/s$ in our case). The second cost function is a simplified version of the cost used in [50], penalizes falls explicitly, and encourages walking at desired speed and with lower cost of transport:

Figure 5.14: Optimizing smooth cost from Equation 5.5 over 50 runs.

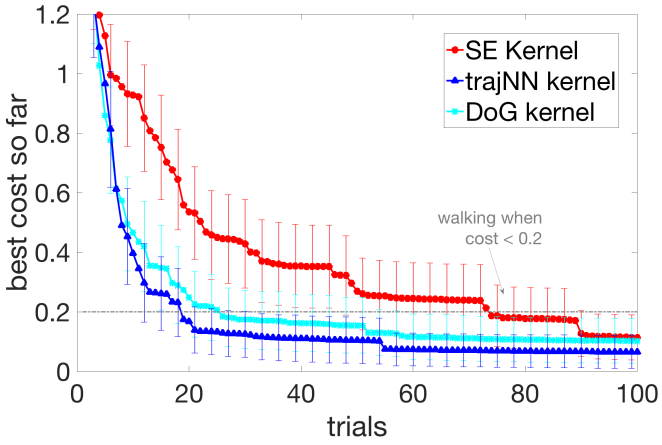
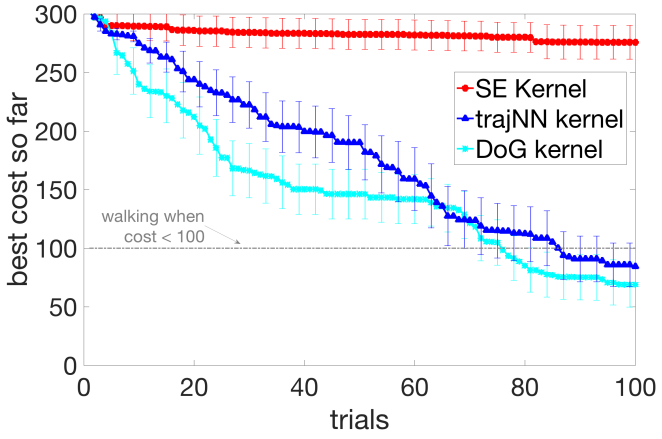


Figure 5.15: Optimizing non-smooth cost from Equation 5.6 over 50 runs.



$$cost_{non\text{-}smooth} = \begin{cases} 300 - x_{fall}, \text{if fall} \\ 100||v_{avg} - v_{tgt}|| + c_{tr}, \text{if walk} \end{cases} \tag{5.6}$$

where x_{fall} is the distance covered before falling, v_{avg} is the average speed of walking, v_{tgt} is the target velocity, and c_{tr} captures the cost of transport.

Figure 5.14 shows that *trajNN* offers a significant improvement in sample efficiency when using $cost_{smooth}$ during optimization. Points with cost less than 0.15 correspond to robust walking behavior. With *trajNN*, more than 90% of runs obtain walking solutions after only 25 trials. In contrast, using *SE* requires more than 90 trials for such success rate. The performance of *trajNN* matches that of a DoG-based kernel kernel. This is notable, since *trajNN* is learned automatically, whereas DoG-based kernel kernel is constructed using domain expertise.

Figure 5.15 shows that *trajNN* also provides a significant improvement when using the second cost. Points with cost less than 100 correspond to walking. With *trajNN*, 70% of the runs find walking solutions after 100 trials. In contrast, optimizing non-smooth cost is very challenging for BO with *SE* kernel: a walking solution is found only in 1 out of 50 runs after 100 trials.

The difference in performance on the two costs is due to the nature of the two costs. If a point walks some distance d, Equation 5.5 penalizes points according to $1/d$ and Equation 5.6 penalizes them according to $-d$. This results in a much steeper fall in cost with the first cost, and BO starts to exploit around points that walk some distance, quickly finding points that walk forever. However, with the second cost, BO continues to explore, and sometimes does not find walking points even in 100 trials. Exploitative methods might be better suited for higher dimensional problems, as compared to exploratory methods, in our experience.

trajNN kernel might be preferable to a cost-based kernel not only for settings with multiple costs. For some higher dimensional problems reduction to a 1-dimensional space in the kernel could be undesirable. So, while 1-dimensional cost-based kernel could yield highly sample-efficient optimization for lower dimensional problems, a higher-dimensional kernel like *trajNN* could provide more flexibility without compromising sample-efficiency for higher dimensional problems.

5.3 Proposed Work

5.3.1 Experiments on hardware

We would like to continue experimenting on hardware - the 50-dimensional controller with the hand-designed DoG transform, as well as with the data-driven NN kernel. We think that transferring the 50-dimensional controller might prove to be quite challenging, and we might have to incorporate mismatch information to successfully implement it.

This brings us to the modelling of mismatch, and testing it on hardware. In simulation, we have performed simple tests and seen an improvement with using the mismatch map, especially on the high dimensional controller. We would like to continue experimenting with it to identify robust ways of defining mismatch, learning it from data, and utilizing it when optimizing controllers.

5.3.2 3-dimensional walking controller

We also propose to build 3-dimensional controllers for walking and implement them on ATRIAS and Sarcos robots. To learn parameters of such controllers we would need a more informed kernel, or learn the same from data. The task of walking in 3-dimensions becomes considerably more challenging, and most of prior research focuses on robots on a boom. But if we would like to extend our approach to a viable solution for robots, we need to test it on a realistic setting, such as 3-dimensional walking.

We want to start with simulation experiments, and then move on to hardware. Since experimenting on two robots might take more time than we have, we plan to focus on experiments with ATRIAS, generalizing to Sarcos if time permits.

CHAPTER 6

Proposed Timeline

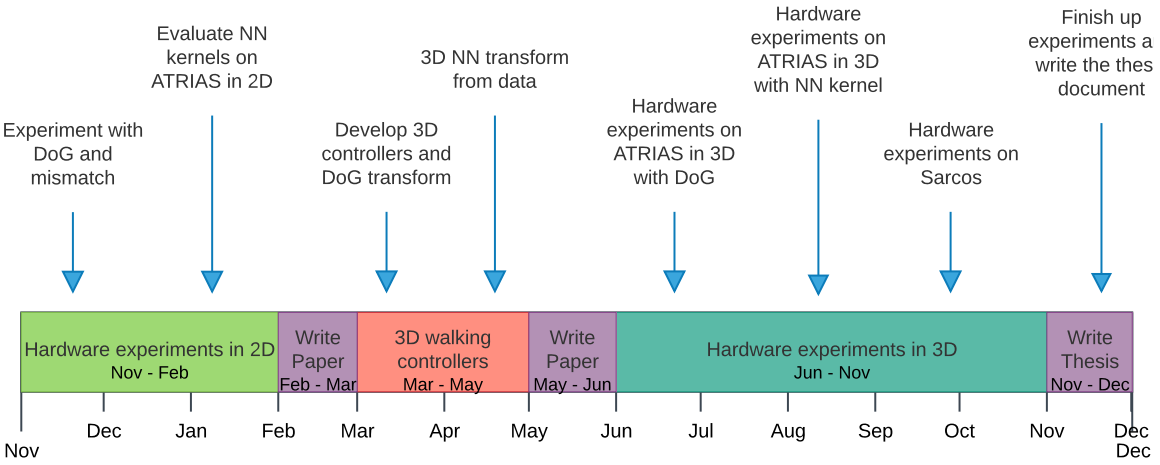
We propose to evaluate our currently ongoing research further, with focus on hardware. We would like to continue studying the efficacy of the DoG transform at optimizing high-dimensional controllers on hardware. We predict that this would need additional tools, such as a mismatch-map, which we are also working towards. We are also interested in experimenting with the neural network kernel and understanding its advantages/shortcomings as compared to the DoG-based kernel kernel.

We would also like to work on 3-dimensional controllers and feature transforms. We would like to extend the DoG to include 3-dimensional features, as well as collect 3-dimensional simulation data to learn a kernel for the same. We would like to use both ATRIAS and Sarcos robot simulations for this to verify that our approach indeed generalizes to multiple robot morphologies. We would like to experiment on 3-dimensional hardware for both robots.

The following schedule summarizes important milestones and approximate time-table for the proposed objectives between Nov 2017 to December 2018.

Task : Evaluating the advantages and limitations of using simulation data to aid learning on robots

- Experiments with DoG transform and mismatch-map on ATRIAS (November - December). Implement a 50-dimensional VNM controller on ATRIAS on a boom.



- Evaluate the Neural Network based kernel on hardware on ATRIAS (January). Study its advantages over hand-designed kernels, if any.
- Aim for a conference submission with results obtained from the hardware experiments (February)
- Develop 3-dimensional walking controllers for ATRIAS and Sarcos simulations. Learn controller parameters in simulation with extended version of DoG transform (March)
- Learn 3-dimensional simulation features using a neural network, and test optimization in simulation (April)
- Aim for a conference paper with simulation results in 3 dimensions (May)
- Evaluate the designed controller and DoG transform on ATRIAS robot in 3-dimensions (June-July)
- Evaluate neural network kernel on 3-dimensional ATRIAS (August)
- Hardware experiments on Sarcos, evaluating the controller and feature transforms (September-November)
- Aim for a journal paper with collective results and write the thesis document (December)

Bibliography

[1] Siyuan Feng, Eric Whitman, X Xinjilefu, and Christopher G Atkeson. Optimization-based full body control for the darpa robotics challenge. *Journal of Field Robotics*, 32(2):293–312, 2015.

[2] Scott Kuindersma, Robin Deits, Maurice Fallon, Andrés Valenzuela, Hongkai Dai, Frank Permenter, Twan Koolen, Pat Marion, and Russ Tedrake. Optimization-based locomotion planning, estimation, and control design for the atlas humanoid robot. *Autonomous Robots*, 40(3):429–455, 2016.

[3] Christian Hubicki, Andy Abate, Patrick Clary, Siavash Rezazadeh, Mikhail Jones, Andrew Peekema, Johnathan Van Why, Ryan Domres, Albert Wu, William Martin, et al. Walking and running with passive compliance: Lessons from engineering a live demonstration of the atrias biped. *IEEE Robotics and Automation Magazine*, 2(4.1):4–1, 2016.

[4] Roberto Calandra, André Seyfarth, Jan Peters, and Marc P. Deisenroth. Bayesian optimization for learning gaits under uncertainty. *Annals of Mathematics and Artificial Intelligence (AMAI)*, 76(1):5–23, 2016.

[5] Alonso Marco, Felix Berkenkamp, Philipp Hennig, Angela P Schoellig, Andreas Krause, Stefan Schaal, and Sebastian Trimpe. Virtual vs. real: Trading off simulations and physical experiments in reinforcement learning with bayesian optimization. *arXiv preprint arXiv:1703.01250*, 2017.

[6] Antoine Cully, Jeff Clune, Danesh Tarapore, and Jean-Baptiste Mouret. Robots that can adapt like animals. *Nature*, 521(7553):503–507, 2015.

[7] Riad Akrour, Dmitry Sorokin, Jan Peters, and Gerhard Neumann. Local bayesian optimization of motor skills. In *Proceedings of the 34th International*

Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017, pages 41–50, 2017.

[8] Igor Mordatch, Kendall Lowrey, and Emanuel Todorov. Ensemble-cio: Full-body dynamic motion planning that transfers to physical humanoids. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, pages 5307–5314. IEEE, 2015.

[9] Sehoon Ha and Katsu Yamane. Reducing hardware experiments for model learning and policy optimization. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 2620–2626. IEEE, 2015.

[10] Patrick MacAlpine, Elad Liebman, and Peter Stone. Adaptation of surrogate tasks for bipedal walk optimization. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*, pages 1275–1276. ACM, 2016.

[11] Jan Peters and Stefan Schaal. Policy gradient methods for robotics. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pages 2219–2225. IEEE, 2006.

[12] Franziska Meier, Daniel Kappler, and Stefan Schaal. Online learning of a memory for learning rates. *arXiv preprint arXiv:1709.06709*, 2017.

[13] Dimitri P Bertsekas and John N Tsitsiklis. Neuro-dynamic programming: an overview. In *Decision and Control, 1995., Proceedings of the 34th IEEE Conference on*, volume 1, pages 560–564. IEEE, 1995.

[14] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[15] Christopher G Atkeson. Randomly sampling actions in dynamic programming. In *Approximate Dynamic Programming and Reinforcement Learning, 2007. ADPRL 2007. IEEE International Symposium on*, pages 185–192. IEEE, 2007.

[16] Eric C Whitman and Christopher G Atkeson. Control of a walking biped using a combination of simple policies. In *Humanoid Robots, 2009. Humanoids*

2009. 9th IEEE-RAS International Conference on, pages 520–527. IEEE, 2009.

[17] Lucian Busoniu, Robert Babuska, Bart De Schutter, and Damien Ernst. *Reinforcement learning and dynamic programming using function approximators*, volume 39. CRC press, 2010.

[18] Nikolaus Hansen. The cma evolution strategy: a comparing review. In *Towards a new evolutionary computation*, pages 75–102. Springer, 2006.

[19] Reuven Rubinstein. The cross-entropy method for combinatorial and continuous optimization. *Methodology and computing in applied probability*, 1(2):127–190, 1999.

[20] Russ Tedrake, Teresa Weirui Zhang, and H Sebastian Seung. Stochastic policy gradient reinforcement learning on a simple 3d biped. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 3, pages 2849–2854. IEEE, 2004.

[21] Jun Morimoto, Jun Nakanishi, Gen Endo, Gordon Cheng, Christopher G Atkeson, and Garth Zeglin. Poincare-map-based reinforcement learning for biped walking. In *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pages 2381–2386. IEEE, 2005.

[22] Hamid Benbrahim and Judy A Franklin. Biped dynamic walking using reinforcement learning. *Robotics and Autonomous Systems*, 22(3-4):283–302, 1997.

[23] Jun Nakanishi, Jun Morimoto, Gen Endo, Gordon Cheng, Stefan Schaal, and Mitsuo Kawato. Learning from demonstration and adaptation of biped locomotion. *Robotics and autonomous systems*, 47(2):79–91, 2004.

[24] Petar Kormushev, Barkan Ugurlu, Sylvain Calinon, Nikolaos G Tsagarakis, and Darwin G Caldwell. Bipedal walking energy minimization by reinforcement learning with evolving policy parameterization. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 318–324. IEEE, 2011.

[25] Siyuan Feng, X Xinjilefu, Christopher G Atkeson, and Joohyung Kim. Robust dynamic walking using online foot step optimization. In *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on*, pages 5373–5378. IEEE, 2016.

[26] Donald R Jones, Matthias Schonlau, and William J Welch. Efficient Global Optimization of Expensive Black-box Functions. *Journal of Global optimization*, 13(4):455–492, 1998.

[27] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando de Freitas. Taking the Human Out of the Loop: A Review of Bayesian Optimization. *Proceedings of the IEEE*, 104(1):148–175, 2016.

[28] Eric Brochu, Vlad M Cora, and Nando De Freitas. A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning. *arXiv preprint arXiv:1012.2599*, 2010.

[29] J Mockus, V Tiesis, and A Zilinskas. Toward Global Optimization, volume 2, chapter Bayesian Methods for Seeking the Extremum. 1978.

[30] Niranjan Srinivas, Andreas Krause, Sham M Kakade, and Matthias Seeger. Gaussian Process Optimization in the Bandit Setting: No Regret and Experimental Design. *arXiv preprint arXiv:0912.3995*, 2009.

[31] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.

[32] Michael L Stein. *Interpolation of Spatial Data: Some Theory for Kriging*. Springer, 1999.

[33] Peter Englert and Marc Toussaint. Combined Optimization and Reinforcement Learning for Manipulation Skills. In *Robotics: Science and Systems*, 2016.

[34] Marcus Frean and Phillip Boyle. Using gaussian processes to optimize expensive functions. In *Australasian Joint Conference on Artificial Intelligence*, pages 258–267. Springer, 2008.

[35] Ruben Martinez-Cantin, Nando de Freitas, Arnaud Doucet, and José A Castellanos. Active policy learning for robot planning and exploration under uncertainty. In *Robotics: Science and Systems*, pages 321–328, 2007.

[36] 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.

[37] Matthew Tesch, Jeff Schneider, and Howie Choset. Using response surfaces and expected improvement to optimize snake robot gait parameters. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 1069–1074. IEEE, 2011.

[38] Roberto Calandra. *Bayesian Modeling for Optimization and Control in Robotics*. PhD thesis, Darmstadt University of Technology, Germany, 2017.

[39] Roberto Calandra, André Seyfarth, Jan Peters, and Marc Peter Deisenroth. Bayesian Optimization for Learning Gaits Under Uncertainty. *Annals of Mathematics and Artificial Intelligence*, 76(1-2):5–23, 2016.

[40] Aaron Wilson, Alan Fern, and Prasad Tadepalli. Using Trajectory Data to Improve Bayesian Optimization for Reinforcement Learning. *The Journal of Machine Learning Research*, 15(1):253–282, 2014.

[41] Alonso Marco, Philipp Hennig, Stefan Schaal, and Sebastian Trimpe. On the design of lqr kernels for efficient controller learning. *arXiv preprint arXiv:1709.07089*, 2017.

[42] Xue Bin Peng, Glen Berseth, and Michiel van de Panne. Terrain-adaptive locomotion skills using deep reinforcement learning. *ACM Transactions on Graphics (TOG)*, 35(4):81, 2016.

[43] Nicolas Heess, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, Ali Eslami, Martin Riedmiller, et al. Emergence of locomotion behaviours in rich environments. *arXiv preprint arXiv:1707.02286*, 2017.

[44] Sergey Levine and Vladlen Koltun. Guided policy search. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1–9, 2013.

[45] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research*, 17(39):1–40, 2016.

[46] Sergey Levine, Nolan Wagener, and Pieter Abbeel. Learning contact-rich manipulation skills with guided policy search. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 156–163. IEEE, 2015.

[47] Pieter Abbeel, Morgan Quigley, and Andrew Y Ng. Using inaccurate models in reinforcement learning. In *Proceedings of the 23rd international conference on Machine learning*, pages 1–8. ACM, 2006.

[48] Verne T Inman, Howard D Eberhart, et al. The major determinants in normal and pathological gait. *JBJS*, 35(3):543–558, 1953.

[49] Rika Antonova, Akshara Rai, and Christopher G Atkeson. Sample efficient optimization for learning controllers for bipedal locomotion. In *Humanoid Robots (Humanoids), 2016 IEEE-RAS 16th International Conference on*, pages 22–28. IEEE, 2016.

[50] Seungmoon Song and Hartmut Geyer. A Neural Circuitry that Emphasizes Spinal Feedback Generates Diverse Behaviours of Human Locomotion. *The Journal of Physiology*, 593(16):3493–3511, 2015.

[51] Mark Cutler, Thomas J Walsh, and Jonathan P How. Real-world reinforcement learning via multifidelity simulators. *IEEE Transactions on Robotics*, 31(3):655–671, 2015.

[52] Matthias Poloczek, Jialei Wang, and Peter I Frazier. Multi-information source optimization. *arXiv preprint arXiv:1603.00389*, 2016.

[53] Christian Hubicki, Jesse Grimes, Mikhail Jones, Daniel Renjewski, Alexander Spröwitz, Andy Abate, and Jonathan Hurst. Atrias: Design and validation of a tether-free 3d-capable spring-mass bipedal robot. *The International Journal of Robotics Research*, 35(12):1497–1521, 2016.

[54] Hartmut Geyer and Hugh Herr. A Muscle-reflex Model that Encodes Principles of Legged Mechanics Produces Human Walking Dynamics and Muscle

Activities. *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, 18(3):263–273, 2010.

[55] Ruta Desai and Hartmut Geyer. Robust Swing Leg Placement Under Large Disturbances. In *ROBIO 2012*, pages 265–270. IEEE, 2012.

[56] JB Morrison. The Mechanics of Muscle Function in Locomotion. *Journal of Biomechanics*, 3(4):431–451, 1970.

[57] KangKang Yin, Kevin Loken, and Michiel van de Panne. Simbicon: Simple Biped Locomotion Control. In *ACM Transactions on Graphics (TOG)*, volume 26, page 105. ACM, 2007.

[58] Nitish Thatte and Hartmut Geyer. Toward Balance Recovery with Leg Prostheses Using Neuromuscular Model Control. *IEEE Transactions on Biomedical Engineering*, 63(5):904–913, 2016.

[59] Nicolas Van der Noot, Auke J Ijspeert, and Renaud Ronsse. Biped Gait Controller for Large Speed Variations, Combining Reflexes and a Central Pattern Generator in a Neuromuscular Model. In *ICRA 2015*, pages 6267–6274. IEEE, 2015.

[60] Marc H Raibert. *Legged robots that balance*. MIT press, 1986.

[61] Albert Wu and Hartmut Geyer. Highly robust running of articulated bipeds in unobserved terrain. In *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, pages 2558–2565. IEEE, 2014.

[62] Zachary Batts, Seungmoon Song, and Hartmut Geyer. Toward a virtual neuromuscular control for robust walking in bipedal robots. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, pages 6318–6323. IEEE, 2015.

[63] William C Martin, Albert Wu, and Hartmut Geyer. Robust spring mass model running for a physical bipedal robot. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 6307–6312. IEEE, 2015.

[64] Alexander Herzog, Nicholas Rotella, Sean Mason, Felix Grimminger, Stefan Schaal, and Ludovic Righetti. Momentum control with hierarchical inverse dynamics on a torque-controlled humanoid. *Autonomous Robots*, 40(3):473–491, 2016.