Coordination of Multiple Dynamic Programming Policies for Control of Bipedal Walking

Eric C. Whitman

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

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

September 2013

Thesis Committee:
Christopher G. Atkeson, Chair
J. Andrew (Drew) Bagnell
Hartmut Geyer
Jerry Pratt

I was personally supported by an NSF Graduate Fellowship (grant number 0750271). My work was supported in part by the US National Science Foundation (ECCS-0824077, and IIS-0964581) and the DARPA M3 and Robotics Challenge programs. The robot was originally created with support from NSF equipment grant CNS 0224419.

Abstract

Walking is a core task for humanoid robots. Most existing walking controllers fall into one of two categories. One category plans ahead and walks precisely; they can place the feet in desired locations to avoid obstacles but react poorly to unexpected disturbances. The other category is more reactive; they can respond to unexpected disturbances but can not place the feet in specific locations. In this thesis, we present a walking controller that has many of the strengths of each category: it can place the feet to avoid obstacles as well as respond successfully to unexpected disturbances.

Dynamic programming is a powerful algorithm that generates policies for a large region of state space, but is limited by the "Curse of Dimensionality" to low dimensional state spaces. We extend dynamic programming to higher dimensions by introducing a framework for optimally coordinating multiple low-dimensional policies to form a policy for a single higher-dimensional system. This framework can be applied to a class of systems, which we call Instantaneously Coupled Systems, where the full dynamics can be broken into multiple subsystems that only interact at specific instants. The subsystems are augmented by coordination variables, then solved individually. The augmented systems can then be coordinated optimally by using the value functions to manage tradeoffs of the coordination variables.

We apply this framework to walking on both the Sarcos hydraulic humanoid robot and a simulation of it. We use the framework to control the linear inverted pendulum model, a commonly used simple model of walking. We then use inverse dynamics to generate joint torques based on the desired simple model behavior, which are then applied directly to either the simulation or the Sarcos robot. We discuss the differences between the hardware and the simulation as well as the controller modifications necessary to cope with them, including higher order policies and the inclusion of inverse kinematics.

Our controller produces stable walking at up to 1.05 m/s in simulation and at up to 0.22 m/s on the Sarcos robot. We also demonstrate the robustness of this method to disturbances with experiments including pushes (both impulsive and continuous), trips, ground elevation changes, slopes, regions where it is prohibited from stepping, and other obstacles.

Acknowledgments

First and foremost, I would like to thank my advisor, Chris Atkeson, for all of his help and support throughout my time as a graduate student. Thanks are also due to the rest of my committee, Drew Bagnell, Hartmut Geyer, and Jerry Pratt for their advice and comments on this draft. I would also like to thank my entire lab group for their comments and feedback on all my research. I am indebted to all the people who worked with the Sarcos robot before me who helped build the understanding, models, and code that my work is based on. Ben Stephens has been particularly helpful both in discussions about my research and in giving me a place to start for working with the robot. I would also like to thank Stuart Anderson for his help, especially with regard to modeling the robot. Pranav Bhounsule has been extremely helpful with data collection and calibrating the robot.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis is concerned with the problem of humanoid walking. Chapter 2 discusses the underlying Dynamic Programming algorithm that we use throughout the remainder of the document. We also consider two proposed improvements aimed at improving the robustness to modeling error: a multiple-model approach and combining the Dynamic Programming optimization with learning of the model. In Chapter 3, we propose a novel method for coordinating multiple controllers, which we apply to simulated humanoid walking in Chapter 4. We also present results quantifying the robustness of our simulated walking to various types of perturbations. In Chapter 5, we discuss modifying the simulation controller to function on real hardware and present results for the Sarcos hydraulic humanoid robot. Chapter 6 discusses ways of modifying the baseline Dynamic Programming algorithm to make it more robust to modeling error.

1.1 Motivation

Humanoid walking robots (as opposed to wheeled robots) have been promoted for several applications including rough terrain locomotion, working in environments built for humans, and working with people. To be effective in any of these roles, the system needs to robust to unexpected disturbances of many kinds. They must be capable of walking in unknown, unstructured environments. The robot should also be compliant, both for safety when working around humans and for stability in the face of an imperfectly sensed environment. It should also be able to place its feet in specific, desired locations. We aim to create a walking system that is both reactive, able to react instantaneously to disturbances or changing conditions, and deliberative, able to follow a plan, avoid obstacles, and place its feet as necessary.

1.2 Types of Walking Robots

The robot's hardware typically dictates much about the style of controller that is used for walking. At one end of the spectrum, are passive dynamic walkers, which are able to walk with no actuation entirely due to the passive dynamics of the mechanical system [20, 65]. Some energy is inevitably dissipated in the impact at foot touch down, so unactuated robots can only walk down a slope, where gravity replaces the lost energy. There are also powered robots based on passive walkers [19], which use actuation to replace the lost energy and can walk on a flat surface. They typically use simple, possibly open-loop, controllers. The design goal for these controllers (and much of the mechanical design) is generally to increase the (typically very limited) stability, which is often expressed in terms of gait sensitivity [37]. For these robots, the controller has little or no control over the internal joint configuration of the robot, which is entirely determined by the dynamics.

At the other end of the spectrum are non-backdrivable robots. Most electric motors produce power at high RPM and low torque, which requires a high gear ratio (often achieved by harmonic drives [60]) to produce the torques and speeds necessary for humanoid walking, especially in human-sized humanoids. Gear ratios amplify the motor inertia by the square of the gear ratio, which can easily dominate the dynamics for large gear ratios. Large gear ratio gear boxes can easily become non-backdrivable. Some of the most successful humanoid robots fall roughly into this category including Honda's ASIMO [36], HUBO [76], and the HRP series [2, 48]. These robots typically use high bandwidth feedback control to precisely track desired positions and velocities at individual joints. With precise control of the internal joint configuration, the primary dynamic concern is tipping over. Accordingly, many of the control strategies used for control of such robots focus on regulation of the Zero Moment Point (ZMP) [47, 97], often by pre-planning trajectories and following them precisely [40].

Somewhere in between these two extremes are compliant force (or torque) controlled robots. For such robots, the actuators produce a force that interacts with the rigid body dynamics rather than completely dominating it. This generally makes it more difficult to track precise desired positions, but compliance can be useful when interacting with an unknown environment. For robots with electric motors, compliance can be achieved either by use of series-elastic actuators [79] or by using backdrivable gearboxes. It is easier to make gearboxes backdrivable for low gear ratios, which necessitates either low torque requirements or special high-torque motors. The ATRIAS robot uses both series elasticity and backdrivable gearing to achieve force control [34]. The Sarcos Primus robot [16, 52], which we use, achieves compliance by force control of hydraulic actuators. Hydraulic actuators do not require gearing to achieve the necessary joint torques, making it easier to achieve high performance force control. Unlike a physical spring, however, force control does have a bandwidth limit, and it may not be compliant on the time scales involved in an impact.

1.3 Simple Models

Humanoid robots often have a large number of joints, meaning that they have a high dimensional state space. High dimensional state spaces are difficult to deal with directly because of the “Curse of Dimensionality” [12]. Instead, many researchers choose to control a simplified model that captures some aspect of the dynamics of the full system. This is related to the concept of “templates” and ”anchors” [29] introduced for studying biological systems. The simple models are easier to control and allow for greater physical intuition. Additionally, certain simple models are well studied and provide a convenient common ground between disparate hardware platforms. The Linear Inverted Pendulum Model (LIPM) [47] and compass gait model [33] [41] are commonly used simple models of walking that consider the motion of the center of mass.

Control of the simplified model only produces control of the aspects of the full system represented by the full model. This includes center of mass motion and footstep placement for the LIPM model. Inverse kinematics can then be used to find joint angles for the full model of the robot [97]. Joint torques, if needed, can be produced by computed torque methods.

The advantage of using simple models is that it is easier to produce controllers for the simple model than for the full system. There are, however, two disadvantages. The first disadvantage is that the full system must be made to behave like the simple system. There will generally be some map from the full state to the simplified state. The full model must be controlled such that the simlpified state evolves in a way consistent with the simple model dynamics. Any discrepancy can result in unpredictable, potentially catastrophic, behavior. The second disadvantage is that use of a simplified model limits the options available to the controller. A controller for a simple model has less representational capacity than a controller for the full system and can not take advantage of aspects of the full system not represented in the simplified system. These three items should be considered when determining which simple model to use in a controller (or whether to use a simple model at all). Virtual Model Control (VMC) is a method that treats design of the simple model as part of the controller design process [81].

1.4 Thesis Contributions

The primary focus of this thesis is on a novel optimal control method for humanoid walking and applying that method to the Sarcos humanoid robot as well as supporting technologies. The core of our approach is the coordination scheme built around the principle of **Instantaneously Coupled Systems**.

Our controller produces **flexible and robust walking in simulation**. It can start, stop, turn, and walk at controllable speed. It achieves robustness to trips, low friction, slopes, abrupt changes in ground height, and pushes by **simultaneously altering the center of mass motion, footstep locations, and footstep timing**.

We have also demonstrated **stable walking on the Sarcos humanoid robot** including robustness tests with slopes, obstacles, and pushes.

Chapter 2

Dynamic Programming

2.1 Introduction

Trajectory optimization [43, 68] can be used to generate a walking pattern offline. Feedback gains are then typically used to stabilize the system around the trajectory. Such a controller is only optimal on the optimized trajectory and often simple feedback approaches are only effective for a small region of state space near the nominal trajectory. If a disturbance or modeling error results in the system straying significantly from the nominal trajectory, a new trajectory can be generated starting from the current state [74]. Model Predictive Control and Receding Horizon Control continuously generate new trajectories from the current state regardless of whether the system is tracking accurately or not [23, 107]. The problem with these methods is that they require optimizing a new trajectory online. Optimizing a walking trajectory is generally computationally expensive, meaning that it can take a long time, resulting in low bandwidth control and a delayed response to perturbations. These methods often must employ approximations (such as linearity) to keep computation time short. Additionally, they may not be able to find globally optimal, or even acceptable, trajectories.

Another possibility is to prepare for perturbations offline by generating a policy that is effective for a large region of state space. These methods generally suffer from the "Curse of

Dimensionality" [12] because high dimensional state spaces are very large and therefore difficult to fill. Trajectory libraries [27, 61, 108] are a common approach in which a large number of different trajectories are generated. A policy can then be implemented by choosing the action from the nearest trajectory or by interpolating between them [6]. It is sometimes possible to analytically compute the convergence region for a given trajectory [99], making it possible to know where more trajectories need to be added to the library and which regions are already covered.

Dynamic Programming (DP) [12] [17] [88] generates optimal controllers for a large region of state space. DP is a class of algorithms for solving Markov Decision Processes [13] [82]. It is based on the key observation that segments of an optimal trajectory are themselves optimal, as formalized by the Bellman equation [12],

$$V(\mathbf{x}) = \min_{\mathbf{u}} L(\mathbf{x}, \mathbf{u}) + \gamma V(f(\mathbf{x}, \mathbf{u})), \tag{2.1}$$

where $\mathbf{x}$ is the state, $\mathbf{u}$ is the control action, $\mathbf{x}_{k+1} = f(\mathbf{x}_k, \mathbf{u}_k)$ is the discrete time system dynamics, γ is a discounting factor slightly less than 1, and V is the value function (total cost to go). This observation makes it possible to reuse computation to efficiently solve for the value function, $V(\mathbf{x})$, and a deterministic policy, $\mathbf{u} = \pi(\mathbf{x})$ everywhere. Typically, a solution is achieved iteratively by using (2.1) to update the value and the $\mathbf{u}$ that achieved the minimum to update the policy,

$$\pi(\mathbf{x}) = \arg\min_{\mathbf{u}} L(\mathbf{x}, \mathbf{u}) + \gamma V(f(\mathbf{x}, \mathbf{u})). \tag{2.2}$$

There are many variants of DP, often based on how value and policy updates are ordered. In value iteration [12], the action is not stored during computation, but only computed as part of the minimization in (2.1). In policy iteration [39], a sweep of updating the policy at every state according to (2.2) is done once followed by multiple sweeps of updating the value with (2.1) until the value converges. This whole process is then repeated. In modified policy iteration [83], which we use a form of, the value update sweeps are not continued until convergence before doing another round of policy update.

If we have a continuous state space, we must approximate this process because it is impossible to store the policy and value function at all states. The most straightforward solution is to represent the continuous state space as a grid of discrete states. The grid can be regularly spaced (as we use in this thesis), or it can have adaptive spacing [71]. Alternately, the states can be randomly sampled [7, 86]. Additionally, $V(f(\mathbf{x},\mathbf{u}))$ will not generally be available if the value function is only defined at some set of discrete points in a continuous space. It must therefore be estimated by interpolation or a local model of the value function. Local models can be Taylor series like, such as those produced by Differential Dynamic Programming [43] or fit to a set of state, value pairs [101].

Continuous action spaces pose a similar problem. Again, the most straightforward approach is to break the continuous space up into a grid of discrete points. However, for actions, the approximation can be avoided completely (at significant computational cost) by performing the minimization in (2.2) with a local optimizer. Random sampling of actions is an alternative for both discrete actions spaces [57] and continuous action spaces [5].

DP policies are globally optimal (up to the grid resolution), avoiding potential problems with local minima. They can handle both discrete decisions such as where to place a footstep as well as continuous decisions such as how much torque to apply. Additionally, DP is useful for optimizing transient responses to perturbations as well as optimizing a steady state periodic gait. In [64], DP on the Poincaré state was used to determine stride-level variables for walking. DP was used for continuous control of some joints in [102] and of all joints in [95].

DP also provides value functions, which give a measure of the cost-to-go from anywhere in the state space. Value functions are useful when coordinating multiple controllers (see Section 3.1.2 for theory and Section 4.2.2 for practice). From just a policy, it is difficult to determine whether a decision is critically important or your optimal controller made it arbitrarily. Value functions provide an automatic way to determine which of multiple competing controllers should have precedence; value is a natural currency for managing tradeoffs between controllers in an optimal control setting.

2.2 Dynamic Programming Algorithm

We use a version of modified policy iteration (based on [5]) that generates time-invariant policies for systems with continuous state and action spaces. We represent the continuous state space by dividing it into a grid. Each grid point stores the current value estimate, $V(\mathbf{x})$, and the current best control vector, $\mathbf{u} = \pi(\mathbf{x})$ ($\mathbf{u} = \tau$ for the pendulum). Because we are interested in time-invariant policies, we do not store a value function for each time step, backing up from the final time. Instead, we store a single value function and continue to refine it until it converges. In practice, we generally stop policy generation once there are few changes per iteration. We do, however, buffer our value function by always using the previous iteration's value when interpolating the terminal value and not updating the value function until the end of the entire iteration. Buffering makes the order in which we update the points not matter, which is sometimes useful, and sometimes detrimental to convergence speed.

During an iteration, for each grid point, we use the Bellman equation to update the value function. Because we have a continuous action space, we cannot enumerate all possible actions. We have found that trying only a single random action and comparing it to the current best action [5] is a good tradeoff between finding the best action for the current estimate of our value function and completing a large number of iterations so that our value function accurately reflects our policy. Our Bellman value update therefore looks like

$$V(\mathbf{x}) = \min_{\mathbf{u} \in \{\mathbf{u}_0, \mathbf{u}_r\}} L(\mathbf{x}, \mathbf{u}) + \gamma V(f(\mathbf{x}, \mathbf{u})), \tag{2.3}$$

where $\mathbf{u}_0$ is the current best action, $\mathbf{u}_r$ is a random action drawn from the legal range. The policy is then updated to the corresponding action. Note that this is an extreme form of modified policy iteration that behaves much like value iteration. Convergence of DP with random actions is discussed in [5].

If the final state, $f(\mathbf{x}, \mathbf{u})$, is outside of the grid, we assign an infinite cost, $V(f(\mathbf{x}, \mathbf{u})) = \infty$. Otherwise, we use multilinear interpolation [21] on the previous iteration's estimate of $V(\mathbf{x})$ to estimate the terminal value. Multilinear interpolation on a grid has the useful property that the value is continuous, even at grid cell boundaries. Simulating forward multiple time steps before evaluating the value function at the result can help minimize the effects of the approximation introduced by interpolating [56]. To ensure that we get a good estimate of the value function, if the final state is in a cell with the originating grid point as a corner, we simulate additional time steps (with the same $\mathbf{u}$, adding the appropriate one step cost, $L(\mathbf{x}_k, \mathbf{u})$, until we enter a cell not bordering the originating grid point. Additionally, if we enter a cell where some corners have infinite value and some corners have finite value, we continue to simulate until we reach a cell where either all corners have infinite value or all corners have finite value.

We can run this algorithm on multiple separate grids, even grids with different dimensionality. For example, the sagittal and coronal stance policies discussed in Section 4.2.1 have three grids each. Multiple grids can also be useful in situations where variable grid resolution is required in various parts of the state space. This usually occurs because the second derivatives of the value function are very large in some region (requiring a finer grid), but not the whole space. Rather than wasting computation power by having a fine grid everywhere, we can have a coarse grid for most of the state space, and a fine grid just for the region that needs it. We do this, for example in the Swing-X and Swing-Y policy discussed in Section 4.2.1. For more extreme situations, such as near constraints, the necessary grid spacing for an accurate representation of the value function may be so fine that it is best to abandon the grid altogether and use an analytical value function (and policy) in this region.

The computational and memory costs of DP are linear in the number of grid points, and therefore exponential in the number of state space dimensions. This limits DP to problems with low dimensional state spaces (about 5 dimensions on a modern desktop computer), though in some cases it is possible to treat higher dimensional problems by breaking them into multiple lower dimensional problems and coordinating the policies, as we will discuss in Section 3.1.

Chapter 3

Instantaneously Coupled Systems

We wish to use dynamic programming (DP) to design a nonlinear optimal controller for a simple model of a biped. DP suffers from the "Curse of Dimensionality", with storage and computation costs proportional to R^d, where R is the grid resolution and d is the dimension of the state. However, breaking the control design problem into parts greatly reduces the storage and computation costs. For example:

$$R^{d/2} + R^{d/2} << R^d. \tag{3.1}$$

By breaking the model into multiple subsystems of lower dimensionality, we are able to work with a higher-dimensional model than would otherwise be computationally feasible. To capture the coupling between the subsystems while keeping them low-dimensional, we augment the subsystems with additional coordination variables. We use dynamic programming to produce optimal policies and value functions for each of the augmented subsystems. Then, by using the value functions to manage tradeoffs between the coordination variables, we coordinate the subsystem controllers such that the combined controller is optimal. Finally, we use the output of this high-level controller (CoM and swing foot accelerations) as the input to a low-level controller, which provides the joint torques necessary to produce those accelerations. In this Chapter, we will discuss our method for decoupling a more complicated system into multiple simpler subsystem. In Chapter 4, we will apply this method to control of bipedal walking.

Because of many walkings systems' high-dimensionality, which makes control difficult, it is common to model parts of a walking system as decoupled so that the lower-dimensional subsystems can be controlled separately [102, 110]. PD servos on individual joints is a very basic form of such decoupling. Unless coordination is handled carefully, the combined controller will be sub-optimal because the subsystem controllers lack the information necessary to make optimal decisions. We present a method of coordination that produces an optimal combined controller.

Decoupling methods are also used in the field of multi-robot coordination, where it is natural to view individual robots as subsystems of the full dynamics including all the robots, and they face the same tradeoff between computational complexity and performance. Decoupled methods such as [49, 59] can find solutions more quickly, but generally have worse performance or make fewer guarantees. When the interaction between agents are only conflicts (e.g. they get in each other's way), we need only resolve the conflicts that result from individual, decoupled planning. In [55], heuristics are used to resolve conflicts by modifying velocities. Learning can also be used to determine when coordination is desirable and plan individually otherwise [53]. In [67], coordination is also assumed to be sparse, and a coordinate action is included as one option in a learned individual policy so that the agents learn when to coordinate with each other. As in our method, M* [104] takes advantage of the fact that subsystems are sometimes coupled, but usually not, to produce solutions that are optimal for the full system while retaining much of the computational advantage of decoupled methods.

In the field of swarm robots, a large number of robots are coordinated, often with no centralized planning and limited communication between individuals. Such methods are often biologically inspired. Control inspired by social insects is discussed in depth in [18]. They can also be based on imitating physics [90]. Control of swarms is a difficult task, and most work in this field is suboptimal, focusing primarily on stabilizability or accomplishing basic tasks.

The field of decentralized control is concerned with using separate controllers for (hopefully lightly coupled) parts of a larger system [89]. Much of the work in this field is focused on proving the existence of and creating stabilizing controllers for various classes of system [3, 22, 105]. The interactions between subsystems can be treated as perturbations to which individual subsystem controllers must be robust to as in [31]. Lyapunov function approaches can be used to show that the system is stabilizable under the decentralized controller [15]. Our method of control based on Instantaneously Coupled Systems evolved from the decentralized control approach. We show that in certain situations, the interaction from other subsystems can be accounted for exactly, so we can maintain optimality.

3.1 Controlling Instantaneously Coupled Systems

For a certain type of system, which we call Instantaneously Coupled Systems (ICS), it is possible to construct an optimal controller by coordinating multiple optimal lower-dimensional controllers. First, subsystems are augmented with coordination variables, which provide enough information to account for coupling to other systems. Then, value functions are used to trade off the coordination variables. This is useful because it reduces an optimal control problem to several lower-dimensional optimal control problems, which can be solved more easily.

3.1.1 Instantaneously Coupled Systems

We define an instantaneously coupled system (ICS) as a dynamic system made up of a set of N lower-dimensional systems. The state of, $\mathbf{x}_f$, and input to, $\mathbf{u}_f$, the full system are given by the composition of the states of and inputs to the lower-dimensional systems,

$$\mathbf{x}_f = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N\} \tag{3.2}$$

and

$$\mathbf{u}_f = \{\mathbf{u}_1, \mathbf{u}_2, ..., \mathbf{u}_N\}. \tag{3.3}$$

The dynamics of each system evolve independently,

$$\dot{\mathbf{x}}_i = f_i(\mathbf{x}_i, \mathbf{u}_i). \tag{3.4}$$

At M specific instants, however, the systems may be coupled such that the dynamics of the subsystems instantaneously depend on the full state,

$$\mathbf{x}_i^+ = f_i^c(\mathbf{x}_f^-, \mathbf{u}_i), \tag{3.5}$$

where the superscripts $-$ and $+$ indicate before and after the coupling event.

The time of the coupling, t_j, is determined by some condition on the full state:

$$\Phi(\mathbf{x}_f(t_j)) = 0 \tag{3.6}$$

There can be one or multiple coupled instants. We only consider systems with a finite number, M, of coupled instants.

3.1.2 Obtaining the Optimal Policy

For an ICS with a cost function of the form

$$C = \int \sum_{i=1}^{N} L_i(\mathbf{x}_i(t), \mathbf{u}_i(t))\, dt + \sum_{j=1}^{M} \left(g(t_j) + h(\mathbf{x}_f(t_j))\right), \tag{3.7}$$

we can construct the optimal policy by finding the optimal policies and value functions for augmented versions of the subsystems and then combining them. Costs of coupling event times and state (g and h) are optional and are not used by the simulation controller, though they will be used on the robot (Chapter 5).

First, we define a coordination state, $\mathbf{x}_c$, as some set of features of the full state, $\mathbf{x}_c = \Theta(\mathbf{x}_f)$. The features, $\mathbf{x}_c$, are a compact means of communicating the essential information about the full state between the subsystems, and must be selected such that it is possible to:

I. Rewrite the coupling dynamics (3.5) as

$$f_i^c(\mathbf{x}_f, \mathbf{u}_i) = \tilde{f}_i^c(\mathbf{x}_i, \mathbf{x}_c, \mathbf{u}_i). \tag{3.8}$$

II. Rewrite the last term in (3.7) as

$$h(\mathbf{x}_f(t_j)) = \tilde{h}(\mathbf{x}_c(t_j)). \tag{3.9}$$

III. Rewrite (3.6) as the intersection of conditions on the low-dimensional systems

$$\Phi(\mathbf{x}_f(t)) = \Phi_1(\mathbf{x}_1(t), \mathbf{x}_c(t)) \cap ... \cap \Phi_N(\mathbf{x}_N(t), \mathbf{x}_c(t)). \tag{3.10}$$

It is always possible to choose $\mathbf{x}_c = \mathbf{x}_f$, but this method will be more useful if an $\mathbf{x}_c$ that is lower-dimensional than $\mathbf{x}_f$ can be found.

Next, we construct the decision space, $\mathbf{x}_d$, by composing t_j and $\mathbf{x}_c(t_j)$ from each of the coupled instants.

$$\mathbf{x}_d = \{t_1, \mathbf{x}_c(t_1), t_2, \mathbf{x}_c(t_2), ..., t_M, \mathbf{x}_c(t_M)\} \tag{3.11}$$

If we hold $\mathbf{x}_d$ constant, the subsystems are completely decoupled and the conditions from (3.10) are constraints:

$$\Phi_i(\mathbf{x}_i(t_j), \mathbf{x}_c(t_j)) = 0. \tag{3.12}$$

With the systems decoupled, we can individually optimize each one with respect to

$$C_i = \int L_i(\mathbf{x}_i(t), \mathbf{u}_i(t))\, dt, \tag{3.13}$$

the only part of (3.7) that depends on the ith system. It then remains only to optimize over all possible choices of $\mathbf{x}_d$ and select the best one.

To accomplish this, we augment the state of each of the subsystems with $\hat{\mathbf{x}}_d$,

$$\hat{\mathbf{x}}_d = \{\hat{t}_1, \mathbf{x}_c(t_1), \hat{t}_2, \mathbf{x}_c(t_2), ..., \hat{t}_M, \mathbf{x}_c(t_M)\} \tag{3.14}$$

$$\hat{t}_j = t_j - t \tag{3.15}$$

which has the trivial dynamics $\dot{\hat{t}}_j = -1$ and $\dot{\mathbf{x}}_c = 0$. This allows us to generate subsystem controllers that can apply the coupling dynamics (3.8) and know when to do so. We switch from the time of coupling in $\mathbf{x}_d$ to the time until coupling in $\hat{\mathbf{x}}_d$ to eliminate the dependence on time in our subsystem controllers. We then produce optimal (with respect to (3.13)) policies and value functions for each of the augmented systems subject to (3.12). Any method that produces both policies and value functions can be used, but we use dynamic programming.

Now, if we have an $\mathbf{x}_f$, we can hold each of the $\mathbf{x}_i$'s constant and get the value as only a function of $\mathbf{x}_d$. This allows us to rewrite (3.7) as only a function of $\mathbf{x}_d$, t, and $\mathbf{x}_f$:

$$C = \sum_{i=1}^{N} V_i(\mathbf{x}_d, t|\mathbf{x}_i) + k(\mathbf{x}_d) \tag{3.16}$$

where $k(\mathbf{x}_d) = \sum_{j=1}^{M} g(t_j) + \tilde{h}(\mathbf{x}_c(t_j))$. We then select the best decision state,

$$\mathbf{x}_d^* = \arg\min_{\mathbf{x}_d} C(\mathbf{x}_d, t, \mathbf{x}_f). \tag{3.17}$$

Having selected $\mathbf{x}_d$, we can look up each of the $\mathbf{u}_i$'s from the individual optimal policies and compose them to form $\mathbf{u}_f$ according to (3.3).

Intuitively, we are able to optimally coordinate multiple subsystems by requiring that the interaction with other subsystems can be exactly described by a small number of values, the decision space, $\mathbf{x}_d$. If we knew these values ahead of time, we could plan for each subsystem independently because nothing else about the other subsystems matters. Since we do not know $\mathbf{x}_d$ ahead of time, we must generate a controller for each subsystem for all possible values of $\mathbf{x}_d$, which we do by augmenting each subsystem's state with $\hat{\mathbf{x}}_d$. This makes generating a subsystem controller more difficult, but if $\mathbf{x}_d$ is low-dimensional, still potentially much easier than controlling the full system.

At run time, the decision space, $\hat{\mathbf{x}}_d$, serves as a means of communication between the subsystems. Each subsystem can produce a value function that is only dependent on the decision space, $V_i(\hat{\mathbf{x}}_d|\mathbf{x}_i)$, which essentially states the preference of the ith subsystem for the decision variables as well as how much it cares. Minimizing the summed value functions gives the best decision variables for the full coordinated system, taking into account the cost for each subsystem. This minimization may not be convex, so may be nontrivial, but this will generally be an easier problem than the original control problem. Once the decision variables, $\mathbf{x}_d$, are fixed, control of each subsystem is independent and can be accomplished using the individual subsystem controllers.

3.2 A Simple Example: Multi-Agent Rendezvous

In this Section, we will consider a simple example, Multi-Agent Rendezvous, and walk through how Sections 3.1.1 and 3.1.2 work in this context. Suppose we have N point robots whose state is an $\{x, y\}$ position on a map, and whose actions are speed, v, in the range $[0, 2]$ and a bearing, θ, in the range $[0, 2\pi]$. Please note that the non-bold x will represent the position in the cartesian direction, whereas the bold $\mathbf{x}$ will represent a state vector (possibly including x). The goal is to synthesize a controller which will move all the robots to meet at a point on the map.

First, we will verify that this meets the requirements for an ICS as set forth in Section 3.1.1. The full state is the concatenation of the individual robot states, $\mathbf{x}_f = \{x_1, y_1, x_2, y_2, ..., x_N, y_N\}$. Similarly the full action space is the concatenation of the individual robot actions, $\mathbf{u}_f = \{v_1, \theta_1, v_2, \theta_2, ..., v_N, \theta_N\}$. The obvious way to partition the state and action space is so that each robot is a subsystem: $\mathbf{x}_i = \{x_i, y_i\}$, $\mathbf{u}_i = \{v_i, \theta_i\}$. The dynamics of each subsystem are then independent and given by

$$\dot{x_i} = v_i \cos(\theta_i), \tag{3.18}$$

$$\dot{y_i} = v_i \sin(\theta_i). \tag{3.19}$$

Each robot moves independently of the other robots' states and actions. First we will consider the problem of optimizing only the rendezvous time, using a pre-determined rendezvous location. Then we will extend the method to optimize the rendezvous time and location.

3.2.1 Optimizing Rendezvous Time

In this example, the only coupling instant is the rendezvous, which terminates the problem. The problem ends with the rendezvous, so we do not care about the state afterwards. Therefore, we can ignore the coupling dynamics (3.5). This single coupling instant occurs when all N robots reach the rendezvous point, which is the condition on the full state described above as (3.6).

If we knew when the rendezvous was to occur, each robot could be controlled independently.

If instead, we include the time of the rendezvous in the cost function, and we want the controller to pick an optimal timing for the rendezvous, the subsystems are coupled by this decision. If it will take one robot a long time to get to the rendezvous, the closer robots need not hurry. However, if they are all close by, they should change their behavior to get there more quickly.

We must select a cost function that has the form given by (3.7). A simple, reasonable choice is

$$C = \int \sum_{i=1}^{N} \left(m(\mathbf{x}_i)^2 v_i + v_i^2 \right) \, dt + aT^2. \tag{3.20}$$

For each robot, we have a terrain cost term (the map cost function, $m(\mathbf{x})$, specifies the cost for moving through a given map location) and a speed term, which penalizes moving quickly. Speed is included in the map cost term to prevent the robots from moving quickly through the high-cost areas. Note that when integrated the speed through time the map cost term is a path integral - the integral of the term is path-dependent, but independent of speed. The rendezvous time is given by T, and the aT^2 term penalizes taking a long time. The coefficient, a, is a weight specifying the importance of speed. In terms of the form required by (3.7), the two individual robot terms in the sum are the one step cost function, L_i, and the T^2 term is g. We have no h term ($h = 0$).

Next, we must select a coordination state, $\mathbf{x}_c$, that meets the three requirements given by (3.8), (3.9), and (3.10). For this simple example, we do not need any states in $\mathbf{x}_c$ ($\mathbf{x}_c = \{\}$). As mentioned above, we do not have any coupling dynamics or h term in the cost function, so satisfying requirements (3.8) and (3.9) is trivial. The condition for the coupling instant, that all robots reach the target location very naturally breaks down into individual requirements to be at that location for each robot, satisfying condition (3.10). That means the decision space is only one dimensional, $\mathbf{x}_d = \{T\}$.

Now, we can synthesize the subsystem controllers. In this example, the subsystems are all identical, so they can all use the same controller. We augment the subsystem state, $\mathbf{x}_i = \{x, y\}$ with $\hat{\mathbf{x}}_d = \{t_{tr}\}$, where $t_{tr} = T - t$ is the time until rendezvous. This extra state is simply a countdown, so the dynamics are given by $\dot{t}_{tr} = -1$. We now use the Dynamic Programming algorithm discussed in Chapter 2 to synthesize a subsystem controller using the state $\mathbf{x} = \{x, y, t_{tr}\}$ and the action $\mathbf{u} = \{v, \theta\}$. We use a one step cost function of $L(\mathbf{x}, \mathbf{u}) = m(\mathbf{x})^2 v + v^2$. To enforce the constraint that it must reach the goal by $t_{tr} = 0$, we use an analytic policy for the final four time units. The analytic policy moves the robot directly towards the goal at whatever velocity is required to reach it at $t_{tr} = 0$. If this requires a speed greater than the maximum (2), then an infinite value is assigned.

At run time, we still have to pick the optimal $t_{tr} = 0$ at every time step. For this example, $\mathbf{x}_d$ is only one-dimensional, so we can brute force the minimization in (3.17). We evaluate the system for many choices of t_{tr} by starting at $t_{tr} = 0$ and scanning up to a maximum reasonable value at a fine resolution. To evaluate the system for a given t_{tr}, we evaluate (3.7), using the value functions to evaluate the integral. This is given by

$$C = \sum_{i=1}^{N} V(x_i, y_i, t_{tr}) + a(t + t_{tr})^2. \tag{3.21}$$

After evaluating this for all choices of t_{tr}, we pick the best one, t_{tr}^*. We can then use the dynamic programming policy to control the robot according to $\mathbf{u}_i = \pi(x_i, y_i, t_{tr}^*)$.

A disadvantage of this method is that running the Dynamic Programming algorithm to generate the policy and value function takes a long time (a few hours on a modern desktop for a grid resolution of 100 for each dimension). This computation can be done offline, but it is dependent on both the map and the target rendezvous location. However, the same policy can be used for any number of robots starting from any locations on the map and for any value of the a coefficient. The coordination step does take time proportional to the number of robots, but should be very quick even for large numbers. Fig. 3.1 shows an example for 6 robots with $a = 0.1$. We reoptimize t_{tr}^* on every time step, but if nothing unexpected happens (the system actually evolves according to the expected dynamic model), then it will have the expected dynamics of $\dot{t}_{tr}^* = -1$.

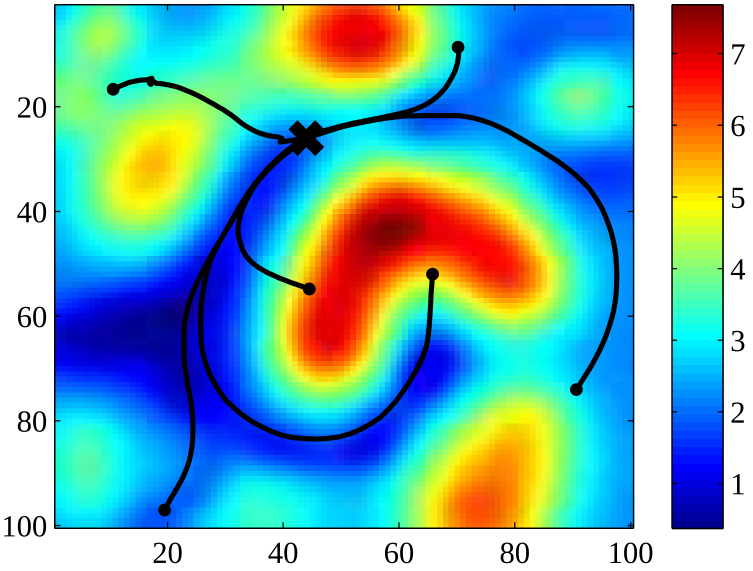

Figure 3.1: An example of optimizing the robot paths and rendezvous timing for 6 robots starting at the dots and meeting at the 'X' with the cost function coefficient $a = 0.1$. The coloring indicates the map cost, $m(\mathbf{x})$.

3.2.2 Optimizing Rendezvous Time and Location

The dynamics of the full system are the same, but this time we want to control a more flexible behavior. We will also use the same cost function as in the previous version, (3.20). The constraint that controls the time of the rendezvous (3.6) must be relaxed from requiring all robots to be at the same pre-specified point to requiring all robots to be at the same point anywhere on the map. This time, we will include the location of the rendezvous, $\{x_r, y_r\}$ in the decision state, $\mathbf{x}_d$, rather than just the time. We will therefore include the average location of the N robots in the coordination state, $\mathbf{x}_c = \{x_{\text{avg}}, y_{\text{avg}}\}$ rather than leaving it empty as before. When we construct the decision space according to (3.11), we will have the rendezvous location.

$$\mathbf{x}_d = \{T, \mathbf{x}_c(T)\} = \{T, x_r, y_r\} \tag{3.22}$$

It is again trivial to ensure that the coordination state, $\mathbf{x}_c$ meets requirements I, (3.8), and II, (3.9), because we do not have either coupling dynamics nor an h term in the cost function. The rendezvous condition can be written as an intersection of subsystem conditions as specified by requirement III, (3.10). We require that for each subsystem, $\mathbf{x}_i = \mathbf{x}_c$.

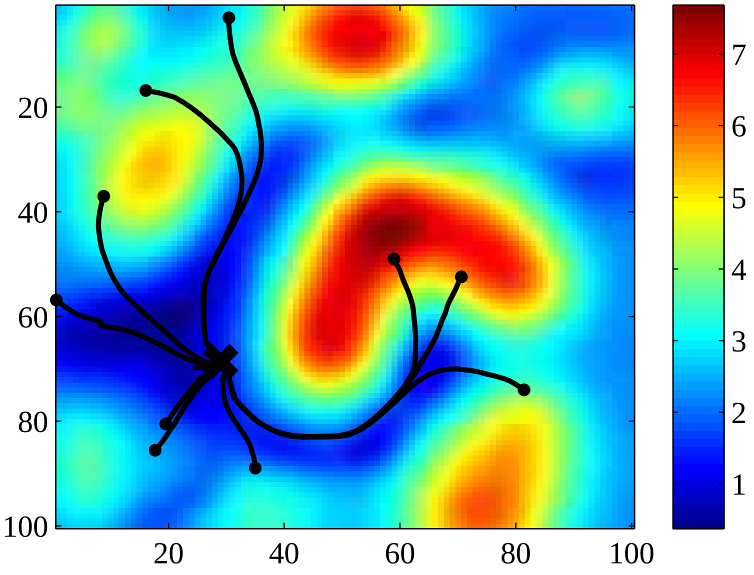

Figure 3.2: An example of optimizing the robot paths and rendezvous timing for 10 robots starting at random locations (dots) and meeting at an optimized rendezvous location ('X') and time with the cost function coefficient $a = 0.1$. The coloring indicates the map cost, $m(\mathbf{x})$.

The rest of the method is identical to the previous time-only optimization except that the minimization in (3.17) is now three-dimensional rather than one-dimensional. It takes a bit longer, but we can still brute force the search at a reasonable resolution in real time. Figure 3.2 shows the paths for ten robots with random starting locations on the same map meeting at an optimized rendezvous location and time.

Chapter 4

Simulated Walking

A humanoid robot should be able to operate in the presence of large disturbances. We propose a method of control for bipedal walking that is capable of responding immediately to unexpected disturbances by modifying center of mass (CoM) motion, footstep location, and footstep timing. The central problem faced by walking controllers is managing reaction forces, which are constrained by friction and the requirement that the center of pressure (CoP) be within the convex hull of the region of support. Many walking controllers focus on CoM motion. A standard method of control is to first generate a CoM trajectory and then track that trajectory with inverse kinematics [97]. Preview control of the CoP can be used to generate CoM trajectories [47]. By modifying the inverse kinematics for force control, it is possible to deal with small disturbances [28].

Unfortunately, even when tracking an optimal trajectory, the resulting controller is only optimal when near the desired trajectory and using full state feedback gains that match those of the globally optimal controller. This will not be the case following a significant unexpected disturbance. Due to constraints on reaction forces, linear independent joint controllers often can stabilize only a small region of state space. It is possible to frequently recalculate the CoM trajectory, taking into account the current robot state [74]. Model Predictive Control (MPC) and Receding Horizon Control (RHC) offer methods of generating trajectories online that continu-

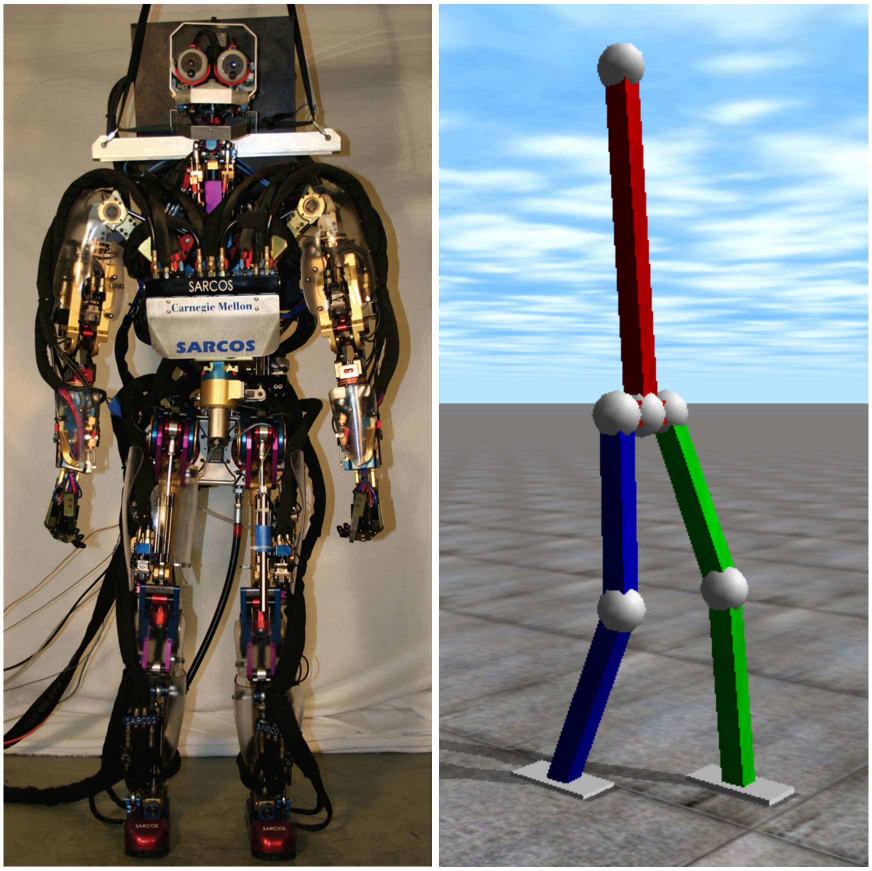

Figure 4.1: The Sarcos Primus hydraulic humanoid robot (left) and the simulation based on it (right). We have modeled the upper body (torso, head, arms) as a single rigid object.

ously start from the current robot state [107].

Model Predictive Control methods are probably the most closely related to our method for walking in terms of capabilities, if not in method. Both MPC and our method are optimal control-based methods that achieve reactiveness by continuously replanning based on the current robot state. Both can produce desired accelerations directly from robot state and control a biped without ever tracking a desired trajectory, though many MPC-based methods replan less frequently and do track trajectories. Unlike our method, MPC does not require massive pre-computation. This makes it more flexible and less subject to the curse of dimensionality because it need not precompute all possible situations. On the other hand, it is limited by run-time computation constraints. Linear dynamics can be used to make the computation faster, as in [107]. This can be modified to optimize footstep locations as well as the CoM trajectory [23], but optimizing the timing is still difficult. Alternatively, more complex dynamics can be handled by recomputing the trajectory at a lower frequency and using the old trajectory for an extended period. In [26], a simulated hopping robot is stabilized using a very short horizon (and therefore computationally cheap) MPC. A nominal periodic trajectory is pre-computed and used as a target for the short-horizon MPC.

For the system to recover from large disturbances, it is necessary to modify the reaction force constraints by adjusting the footstep placement or timing. One possible approach to this is trajectory libraries, where multiple trajectories are generated in advance and an appropriate one is used depending on the current robot state. Examples of trajectory libraries are given for standing balance in [61] and for walking in [108]. It is also possible to modify MPC so that it determines foot placement online [23]. In [70], the footstep timing is modified online in response to manually changed footstep locations.

Many methods that control the LIPM, such as preview control [45] and some forms of model predictive control [107], focus on control of the zero moment point (ZMP). For many of these methods, it is either impossible or computationally expensive to extend them to work with a model that considers angular momentum or upper body rotation.

Due to disturbances and un-modeled dynamics, angular momentum and posture regulation are required, which can directly interfere with a controller based solely on the LIPM. Several authors have derived models of angular momentum for biped robots [32, 45] and it has been shown that exploiting angular momentum can add significant stability to the system [80, 92]. The subject of upper body angular momentum coordination and control for locomotion in position controlled humanoid robots has been considered [50, 98].

Full body torque controllers based on force-based objectives such as desired COM acceleration and change of angular momentum have been presented [1, 38, 58, 63]. Controllers such as these have achieved hip-strategy-like behaviors by making the angular momentum or posture objective less important than COM regulation. However, angular momentum and posture objectives have been mostly limited to regulation tasks.

4.1 Walking as an ICS

To generate a walking controller, we first approximate walking as an ICS. Summing the forces and torques on the system gives us dynamics equations for the CoM

$$\mathbf{f}_L + \mathbf{f}_R + \mathbf{f}_g = m\ddot{\mathbf{c}} \tag{4.1}$$

$$(\mathbf{p}_L - \mathbf{c}) \times \mathbf{f}_L + \boldsymbol{\tau}_L + (\mathbf{p}_R - \mathbf{c}) \times \mathbf{f}_R + \boldsymbol{\tau}_R = \dot{\mathbf{L}} \tag{4.2}$$

where $\mathbf{c}$, $\mathbf{p}_L$, and $\mathbf{p}_R$ are the positions of the CoM, left and right feet, $\mathbf{f}_L$, $\mathbf{f}_R$, $\boldsymbol{\tau}_L$, and $\boldsymbol{\tau}_R$ are the reaction forces and torques generated at the feet, $\mathbf{f}_g = [0, 0, -g]^\mathsf{T}$ is the force of gravity, m is the mass, and $\mathbf{L}$ is the angular momentum. Since the absolute position is rarely relevant, it is useful to place the origin of the coordinate system at the stance foot so that the CoM location, $\mathbf{c}$, and the swing foot location, $\mathbf{p}_w$, are defined relative to the stance foot. During double support, the foot that will be in stance next is considered the stance foot. It is also useful to define the total reaction force and torque as follows:

$$\begin{aligned} \mathbf{f} &= \mathbf{f}_L + \mathbf{f}_R \\ \boldsymbol{\tau} &= \boldsymbol{\tau}_L + \boldsymbol{\tau}_R. \end{aligned} \tag{4.3}$$

During single support, the swing foot cannot generate reaction force, so one of the pairs of force and torque must be zero. If we then constrain our policy such that $\dot{\mathbf{L}} = 0$ and $\ddot{\mathbf{c}}_z = 0$, (4.1) and (4.2) simplify to the well-known Linear Inverted Pendulum Model (LIPM) [44] [46]. We further constrain the dynamics with $\dot{\mathbf{c}}_z = 0$ and $\mathbf{c}_z = h$ and write the LIPM dynamics as

$$\ddot{\mathbf{c}}_x = \mathbf{c}_x \frac{g}{h} + \frac{\boldsymbol{\tau}_y}{mh} \tag{4.4}$$

$$\ddot{\mathbf{c}}_y = \mathbf{c}_y \frac{g}{h} + \frac{\boldsymbol{\tau}_x}{mh}. \tag{4.5}$$

We model the swing leg as fully controllable and treat the acceleration of the swing foot, $\ddot{\mathbf{p}}_w$, as a control variable.

During double support, there is no swing foot to accelerate, but the horizontal CoM acceleration depends on how the weight is distributed between the two feet, which we define as

$$w = \frac{\mathbf{f}_{L,z}}{\mathbf{f}_{L,z} + \mathbf{f}_{R,z}}. \tag{4.6}$$

We assume that we can select w during double support such that

$$\ddot{\mathbf{c}}_x = \frac{\boldsymbol{\tau}_y}{mh} \tag{4.7}$$

$$\ddot{\mathbf{c}}_y = \frac{\boldsymbol{\tau}_x}{mh}. \tag{4.8}$$

Equations (4.7) and (4.8) are approximations because they require that both

$$w = \frac{\mathbf{c}_x - \mathbf{p}_{L,x}}{\mathbf{p}_{R,x} - \mathbf{p}_{L,x}} \tag{4.9}$$

and

$$w = \frac{\mathbf{c}_y - \mathbf{p}_{L,y}}{\mathbf{p}_{R,y} - \mathbf{p}_{L,y}}. \tag{4.10}$$

It is only possible to simultaneously satisfy (4.9) and (4.10) if the CoM is directly above the line between the two feet. However, this approximation is small because the CoM is usually near this line during double support as shown in Figure 4.2, double support is brief, and the low-level controller can often fix some of the discrepancy by adjusting $\boldsymbol{\tau}$. This approximation is necessary because it allows us to decouple the sagittal and coronal dynamics, and it is useful because it allows us to calculate the CoM acceleration without knowing the position of both feet.

These dynamics constitute a 5 degree of freedom (DoF) ICS with a 10-dimensional state space (position and velocity for each DoF),

$$\begin{gathered}\mathbf{x}_f = \{\mathbf{c}_x, \dot{\mathbf{c}}_x, \mathbf{c}_y, \dot{\mathbf{c}}_y, \mathbf{p}_{w,x}, \\ \dot{\mathbf{p}}_{w,x}, \mathbf{p}_{w,y}, \dot{\mathbf{p}}_{w,y}, \mathbf{p}_{w,z}, \dot{\mathbf{p}}_{w,z}\}\end{gathered} \tag{4.11}$$

and a 5-dimensional action space (one for each DoF),

$$\mathbf{u}_f = \{\boldsymbol{\tau}_y, \boldsymbol{\tau}_x, \ddot{\mathbf{p}}_{w,x}, \ddot{\mathbf{p}}_{w,y}, \ddot{\mathbf{p}}_{w,z}\}. \tag{4.12}$$

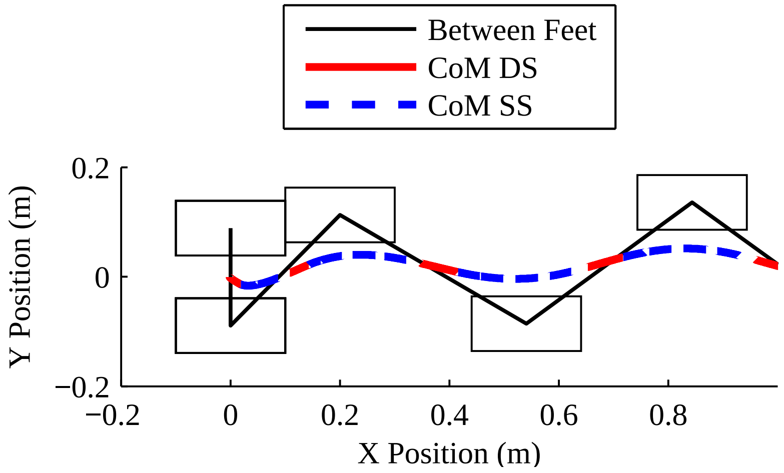

Figure 4.2: The CoM and footstep pattern of the walking simulation starting from rest (top view). Note that during double support (DS), the CoM is near the line between the two feet.

We can then partition the state and action space into 5 subsystems, one for each DoF:

$$\begin{aligned}
\mathbf{x}_s &= \{\mathbf{c}_x, \dot{\mathbf{c}}_x\} & \mathbf{u}_s &= \{\boldsymbol{\tau}_y\} \\
\mathbf{x}_r &= \{\mathbf{c}_y, \dot{\mathbf{c}}_y\} & \mathbf{u}_r &= \{\boldsymbol{\tau}_x\} \\
\mathbf{x}_x &= \{\mathbf{p}_{w,x}, \dot{\mathbf{p}}_{w,x}\} & \mathbf{u}_x &= \{\ddot{\mathbf{p}}_{w,x}\} \\
\mathbf{x}_y &= \{\mathbf{p}_{w,y}, \dot{\mathbf{p}}_{w,y}\} & \mathbf{u}_y &= \{\ddot{\mathbf{p}}_{w,y}\} \\
\mathbf{x}_z &= \{\mathbf{p}_{w,z}, \dot{\mathbf{p}}_{w,z}\} & \mathbf{u}_z &= \{\ddot{\mathbf{p}}_{w,z}\}
\end{aligned} \tag{4.13}$$

where the subscripts, s, r, x, y, and z, refer to the sagittal stance, coronal stance, swing-x, swing-y, and swing-z subsystems.

The systems are only coupled during stance transitions (touch down and lift off). We choose a common state that describes the horizontal location of the swing foot, $\mathbf{x}_c = \{\mathbf{p}_x, \mathbf{p}_x\}$. In order to keep the decision state, $\hat{\mathbf{x}}_d$, low-dimensional, we consider only the next transition ($M = 1$) and make assumptions about all future transitions. This gives us a decision state of

$$\hat{\mathbf{x}}_d = \{t_t, x_{td}, y_{td}\} \tag{4.14}$$

where t_t is the time until transition, and $\{x_{td}, y_{td}\}$ is the location where the swing foot will touch down. For lift off transitions, x_{td} and y_{td} can be omitted. The stance subsystems assume that subsequent transitions will have the nominal timing (0.1 second double support and 0.4 second single support), but that they will be able to select future touchdown locations. The swing subsystems assume that subsequent transitions will have nominal values from steady state walking. Figure 4.3 shows t_t as a function of time for the walking simulation starting from rest and accelerating to steady state walking at 0.56 m/s. During single support, it is convenient to refer to t_t as time until touchdown, t_{td}.

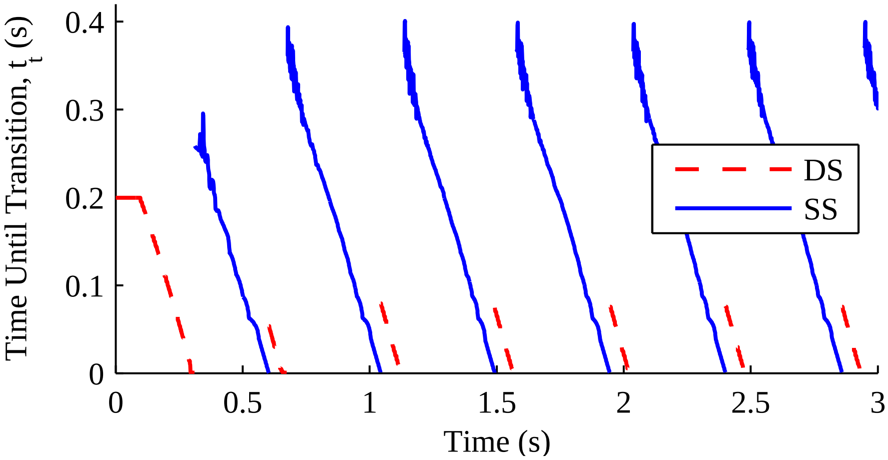

Figure 4.3: Time until transition, t_t versus time. To reduce computation, policies are only computed for $t_t < 0.2$ during double support.

This selection of $\mathbf{x}_c$ and the resulting $\hat{\mathbf{x}}_d$ allows our subsystem controllers to determine the optimal action for all possible choices of footstep timings and locations. The value functions can then be used to determine which choice of these variables is optimal for the full ICS.

We minimize the cost function

$$C = \int (w_1 \boldsymbol{\tau}_x^2 + w_2 \boldsymbol{\tau}_y^2 + w_3(\dot{\mathbf{c}}_x - v_{des})^2 + w_4(\mathbf{c}_y - w_h)^2 + \\ w_5(\mathbf{p}_{w,z} - h_{fc})^2 + \ddot{\mathbf{p}}_{\mathbf{w}}^{\mathsf{T}} \mathbf{W}_6 \ddot{\mathbf{p}}_{\mathbf{w}}) \, dt \tag{4.15}$$

subject to the constraint that

$$\mathbf{p}_w(t_{td} = 0) = \{x_{td}, y_{td}, 0\}. \tag{4.16}$$

The values w_1 through w_5 are weighting constants, $\mathbf{W}_6$ is a 3 by 3 diagonal weighting matrix,

$w_h = 0.09\text{m}$ is half the width of the hips, and $h_{fc} = 0.03\text{m}$ is the nominal foot clearance height. Typical values of the weights are given in Table 4.1.

Table 4.1: Typical values for the weights in the simple model cost function (4.15). We typically normalize all cost terms by dividing by a "reasonable maximum value" to make tuning the weights more intuitive. We manually tune the normalized weights, but present both the normalized and unnormalized weights here.

Weight	Value	Normalized Value
w_1	0.001	1
w_2	0.0003	1
w_3	2	2
w_4	300	3
w_5	1111	1
$\mathbf{W}_6$	$\begin{pmatrix} 0.0008 & 0 & 0 \\ 0 & 0.0008 & 0 \\ 0 & 0 & 0.0032 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 2 \end{pmatrix}$

Note that (4.15) has the form of (3.7) and that (4.16) can be decoupled as in (3.10). This model of walking thus meets all the criteria of an ICS. If we omit the dimensions of $\hat{\mathrm{x}}_d$ that do not affect the dynamics, the original 10-dimensional system is equivalent to a coordinated set of one 3-dimensional system (swing-Z) and four 4-dimensional systems. In practice, we are able to use a change variables to reduce the swing-X and swing-Y policies from 4 dimensions to 3 dimensions and combine them (discussed in Section 4.2.1). We also add weight distribution to the state of the sagittal and coronal policies (discussed in the next Section) during double support. After these modifications, we have two 3-dimensional policies (one of which is used twice) and two policies that are 4-dimensional during single support and 5-dimensional during double support.

Non-ICS Modifications

The requirements for a system to be an Instantaneously Coupled System and therefore optimally coordinated by the method described in Section 3.1.2 are somewhat limiting. We make two modifications to the high-level walking model: we include weight distribution as a control during double support and we do not use the LIPM dynamics in the sagittal plane. These modifications result in the system not truly being an ICS, so our coordination is no longer optimal. Despite the loss of optimal coordination, the modifications significantly improve results by changing the restrictions on our walking controller.

Rather than allow the weight distribution between the two feet in double support, w, to be determined by the CoM position as described in (4.7) and (4.8), we wish to actively control the weight distribution to help with balance. This allows our controller to control center of mass location by adjusting how much of its total weight is on each foot, which moves the center of pressure. This is particularly helpful when starting from rest because it allows the controller to shift the center of mass towards the first stance foot during the initial double support phase. Without shifting weight in this way, the system must step far out to the side of its first step when it begins walking from rest. Unfortunately, there is no way to coordinate the weight distribution within the ICS framework because it couples the sagittal and coronal sub-systems (they must have the same weight distribution) continuously during double support rather than just at the transition points. The run-time coordination between the two policies to determine the weight distribution during double support is discussed in Section 4.2.2.

The LIPM dynamics, (4.4) and (4.5), require that we do not accelerate the CoM vertically ($\ddot{\mathbf{c}}_z = 0$). For walking on flat ground, this implies a constant CoM height, $\mathbf{c}_z = h$. To achieve this, we must pick a height, h, that is low enough for the foot to reach the desired touch down locations. If we have this same CoM height during the middle of single support, the stance knee will be very bent, giving the appearance of a crouch-walk. If we then lower the height a bit more to add margin for the occasional abnormally long step required to recover from a perturbation, the problem becomes even more pronounced. This both looks unnatural and drastically increases the torque required at the knee. We therefore wish to allow the CoM to move up and down as the system walks. Unfortunately, the LIPM dynamics were the reason that sagittal and coronal plane motion decoupled completely. Clearly, the CoM must be at the same height for both subsystems at all times. We allow the sagittal plane controller to control the CoM height and treat the effect of the vertical motion as a disturbance on the coronal plane, for which we continue to use the LIPM dynamics for generating a high-level policy. We consider the LIPM dynamics acceptable for high-level policy generation in the coronal plane because steps are much shorter in the y direction and the legs typically remain near vertical in the coronal plane.

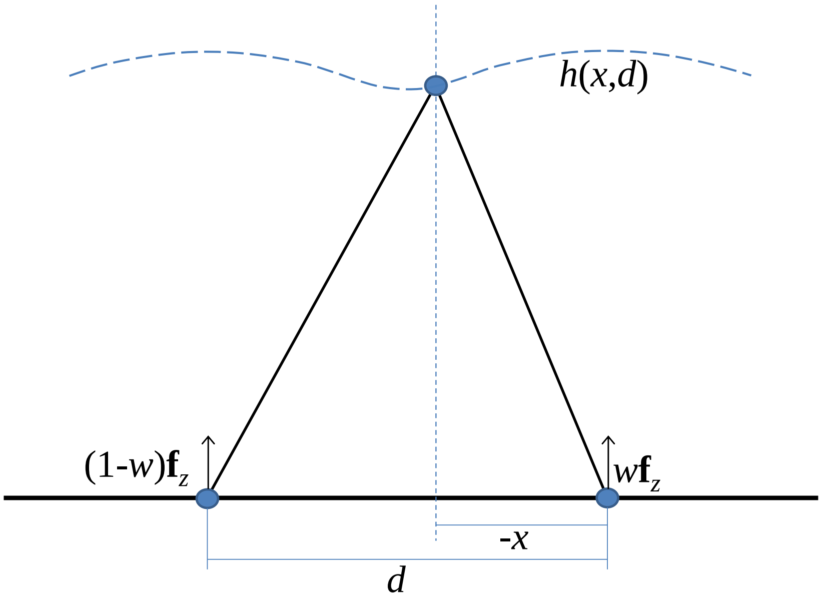

Figure 4.4: Diagram of the sagittal system in double support not using LIPM dynamics. The fraction of the total vertical force, $\mathbf{f}_z$ on the right leg is given by w. Note that in the configuration drawn, x has a negative sign.

We do not wish to increase the dimensionality of our system by adding $\mathbf{c}_z$ and $\dot{\mathbf{c}}_z$ as states in our system, so we set the CoM height to a function of x and d, $h(x, d)$, where x and d are the CoM location in the X direction and the distance between the feet, measured as shown in Figure

4.4. Measuring torque around the CoM gives us

$$\dot{\mathbf{L}} = \boldsymbol{\tau}_y + \mathbf{f}_x h(x,d) - xw\mathbf{f}_z - (d+x)(1-w)\mathbf{f}_z. \tag{4.17}$$

We have the constraint that $\dot{\mathbf{L}} = 0$. When $\mathbf{f}_z$ is not constant, the ZMP constraint does not give us a constant limit on $\boldsymbol{\tau}_y$, so it is convenient to write our control in terms of center of pressure location on the foot. The relative CoP location, r is related to the torque by $\tau = r\mathbf{f}_z$ and constrained by $|r| \leq l_f$ where $l_f = 0.1$ is half the length of the foot. We know from Newton's Second Law that $\mathbf{f}_x = m\ddot{x}$ and adding gravity gives us that $\mathbf{f}_z = m(g + \ddot{h}(x,d))$. If we assume that $\dot{d} = 0$, we get that

$$\ddot{h}(x,d) = \frac{\partial^2 h}{\partial x^2}\dot{x} + \frac{\partial h}{\partial x}\ddot{x}. \tag{4.18}$$

Substituting all of these relations into (4.17) gives us

$$0 = m\ddot{x}h(x,d) + m(g + \frac{\partial^2 h}{\partial x^2}\dot{x} + \frac{\partial h}{\partial x}\ddot{x})(r + dw - d - x), \tag{4.19}$$

which we can solve for $\ddot{x}$ to get

$$\ddot{x} = \frac{(g + \frac{\partial^2 h}{\partial x^2}\dot{x})(x + d - dw - r)}{h(x,d) + \frac{\partial h}{\partial x}(r + dw - d - x)}. \tag{4.20}$$

In single support, $w = 1$ and (4.20) still holds with $dw - d = 0$ canceling out. This removes any dependence on d, (so long as h does not depend on d), which makes sense because the second foot is not on the ground during single support.

During single support, $h(x,d)$ is based on the compass gait, with the hip moving in an arc around the stance foot and the CoM located a fixed offset, $a = 0.2$ above the hip:

$$h(x,d) = a + \sqrt{l_{SS}^2 + x^2}, \tag{4.21}$$

where l_{SS} is the constant length of the stance leg during single support. As the hip moves forward during double support, the front leg must get shorter (bend the knee more) and the back leg must get longer (straighten the knee). For this to be possible, we can not start double support with the back leg completely straight. For this reason, we keep the knee slightly bent during single support, making l_{SS} shorter than the full leg length. Note that this is necessary because our controller walks with flat footsteps, with no toe off. Humans (and robots that have toe off in their gait) lengthen their back leg during double support by lifting their heel off of the ground.

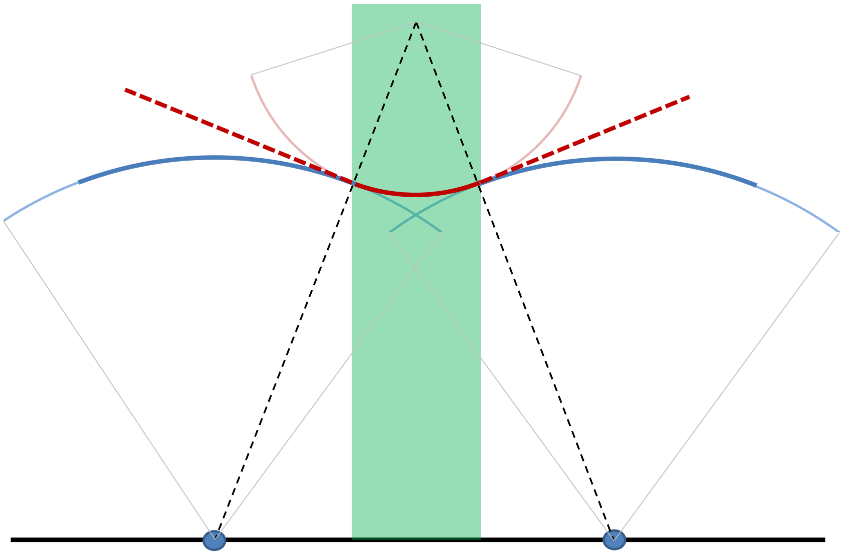

Figure 4.5: Diagram of $h(x, d)$. The green region is the expected double support region. The blue lines show the single support arcs. The red line shows $h(x, d)$ during double support with the joining arc solid and the tangent lines outside of the expected region dashed.

During double support, we wish to make a smooth transition between the preceding and following single support arcs. This is impossible with h being only a function of x and d because we do not know when/where touchdown occurred or when/where liftoff will occur, so we attempt to make it as smooth as possible using nominal values. First, we find the expected double support region (the region the CoM will pass through in a nominal step), which has a length of the desired velocity times the nominal double support duration and is centered between the two feet. If the feet are outside this region (as they usually will be), we find the circle that is tangent to the two single support arcs as they enter this region (shown in Figure 4.5). If the CoM is in the expected double support region, $h(x, d)$ is on that joining circle. If outside the expected region, $h(x, d)$

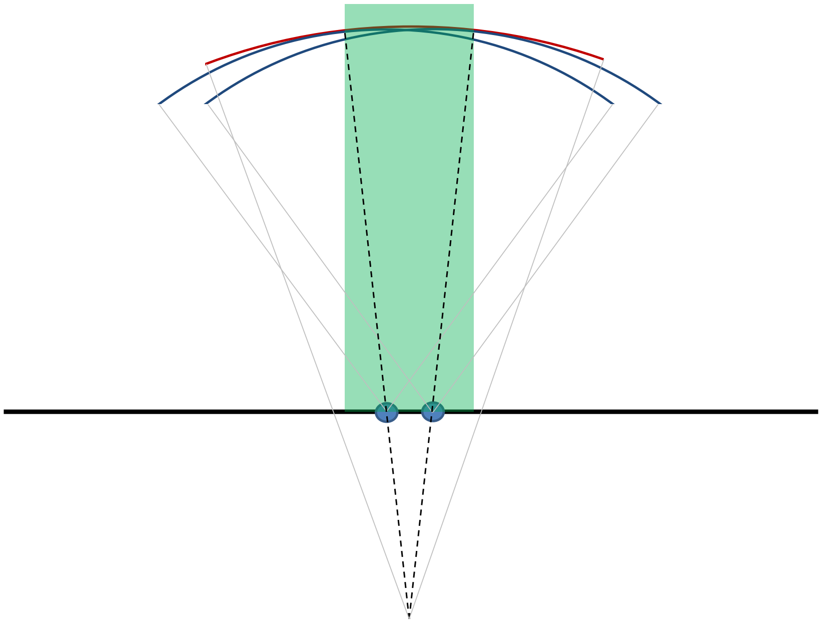

Figure 4.6: Diagram of $h(x, d)$ when the feet are inside the expected double support region (green region). The blue lines show the single support arcs, and the red line shows the double support arc when the feet are within the expected double support region.

is on the tangent lines. If the feet are very close together and inside the expected DS region, we again find the circle that is tangent to both single support arcs as they enter the expected DS region. This time the circle's center will be below the ground (shown in Figure 4.6). We then assign $h(x, d)$ to this circle regardless of whether or not it is in the expected DS region. In the boundary case where the feet are right at the edge of the expected DS region, $h(x, d)$ will be a simple horizontal line during double support.

4.2 Walking Controller

We use the principle of an ICS to generate a walking controller for a simulated biped based on our Sarcos Primus System [16] [52] hydraulic humanoid robot with force-controlled joints. The simulation is of approximately human size (CoM is 1.0 m high when standing straight) and mass

(78 kg). It is a 3D 5-link (torso and two 2-link legs) rigid body simulation with 16 degrees of freedom: 6 to locate and orient the torso as well as 3 at each hip and 1 at each knee. It is controlled by 12 torque controlled joints: 3 at each hip, 1 at each knee, and roll and pitch actuation between each point foot and the ground. The CoP constraint of a finite-size foot is simulated by enforcing

$$\begin{aligned} |\boldsymbol{\tau}_x| &\leq w_f \mathbf{f}_z \\ |\boldsymbol{\tau}_y| &\leq l_f \mathbf{f}_z \end{aligned} \tag{4.22}$$

on each foot where $w_f = 0.05m$ and $l_f = 0.1m$ are approximately half the width and length of a human foot. Friction (coefficient of friction is $\mu = 1.0$) is modeled as a spring and damper between the foot and the ground. When the friction cone,

$$\frac{\sqrt{\mathbf{f}_x^2 + \mathbf{f}_y^2}}{\mathbf{f}_z} < \mu, \tag{4.23}$$

or yaw torque constraint,

$$\frac{\boldsymbol{\tau}_z}{\mathbf{f}_z} < \mu_r, \tag{4.24}$$

is violated, slipping is modeled by resetting the rest position of the spring and switching to a lower kinetic coefficient of friction ($\mu_k = 0.8$).

We use coordinated DP polices to produce an optimal controller for the ICS described in Section 4.1. This functions as our high-level controller, providing input CoM and swing foot accelerations to a low-level controller, which outputs joint torques.

4.2.1 Policy Generation

Policies and value functions are generated for each of the five subsystems using dynamic programming as discussed in Section 2.2. A discount factor, γ, of 0.9995 is used, which corresponds to costs fading to half importance after 1.4 seconds (nearly 3 steps).

Swing-Z Policy Generation

For the swing-z system, the dynamics are not affected by x_{td} or y_{td}, so it is sufficient to generate a policy on the 3-dimensional state space of $\{\mathbf{p}_{w,z}, \dot{\mathbf{p}}_{w,z}, t_{td}\}$ - denoted $\{z, \dot{z}, t_{td}\}$ here for convenience. Our only control action is $\dot{\mathbf{p}}_{w,z}$ - denoted $\ddot{z}$. We are controlling acceleration directly in the high-level walking system, so it has simple second order dynamics and t_{td} simply counts down. We wish to immediately lift the swing foot, hold it steady at a nominal height of $z_{\text{nom}} = 0.03\text{m}$, then reach the ground ($z = 0$) with a small nominal touchdown velocity of $\dot{z}_{td} = -0.04\text{m/s}$ when $t_{td} = 0$. We use a non-zero touchdown velocity to ensure that we get firm contact and to avoid numerical issues, but we keep it small to avoid large impacts. We divide the state space into a grid with minimum states of {0 m, -3 m/s, 0 s}, maximum states of {0.08 m, 3 m/s, 0.6 s}, and resolutions of {81, 301, 121}. To simplify analysis, where possible we select grid resolutions such that we have round numbers for the grid spacing. For this policy, for example, we have spacing of {0.001 m, 0.02 m/s, 0.005 s}. We use a cost function of

$$L(\mathbf{x}, \mathbf{u}) = 0.5 \left(\frac{\ddot{z}}{\ddot{z}_{\text{max}}} \right)^2 + \left(\frac{z - z_{\text{nom}}}{z_{\text{nom}}} \right)^2, \tag{4.25}$$

where z_{max} is the maximum allowable acceleration. The first term minimizes acceleration, and the second term causes the foot to lift off and hold at the nominal height, z_{nom}. The weight, 0.5, was selected manually by experimentation and the denominators are used to normalize terms that have different units.

For $t_{td} \leq 0.03\text{s}$, we use an analytical controller to select an action and produce a corresponding value function. An analytical function is necessary because the second derivatives of the policy and value function grow arbitrarily large as we approach touchdown, which would require an arbitrarily fine grid spacing to approximate accurately. Our analytic controller selects the single acceleration, $\ddot{z}$, to use from now until touchdown that minimizes the cost

$$C = 2t_{td}\ddot{z}^2 + 1000(\dot{z}_f - \dot{z}_{\text{nom}})^2, \tag{4.26}$$

subject to the constraint that it actually touch down ($z = 0$) within $T_{\text{slop}} = .00075\text{s}$ (3/4 of a simulation time step) of the nominal t_{td}. The final velocity is given as $\dot{z}_f = \dot{z} + \ddot{z}t_{td}$. The constant 2 was selected to scale the cost of terminal acceleration relative to the cost of acceleration the rest of the time. Note that this is not normalized and is in the continuous rather than discrete time setting. We somewhat arbitrarily determined that a terminal acceleration of 10 m/s/s was about equally bad as missing the final velocity by 0.02 m/s, which tells us that the second constant should be 500 times larger than the first.

First we find the $\ddot{z}$ which minimizes (4.26) ignoring the constraint by expanding (4.26), which gives a quadratic in $\ddot{z}$, which can be easily minimized. Then we find the range of $\ddot{z}$ that satisfies the timing constraint by finding the $\ddot{z}$ that achieves $z = 0$ in $t_{td} + T_{\text{slop}}$ and the $\ddot{z}$ that achieves $z = 0$ at $t_{td} - T_{\text{slop}}$ and constrain our action to this range. We must also satisfy the constraint that $|\ddot{z}| \leq \ddot{z}_{\text{max}}$. If these ranges do not overlap, the problem is not solvable and we assign an infinite value. Otherwise, we plug our $\ddot{z}$ into (4.26) to get the value. For much of the state space, it will not be possible to satisfy both constraints so we will get an infinite value. For most of the rest of the state space, the timing constraint will be active and the cost function only serves to determine whether we touch down as early as possible ($t_{td} - T_{\text{slop}}$) or as late as possible ($t_{td} + T_{\text{slop}}$). The value of T_{slop} was selected such that for every physical state $\{z, \dot{z}\}$, there is at least one integer number of time steps for which touchdown is feasible (some $\ddot{z}$ satisfies both constraint pairs).

Swing-X and Swing-Y Policy Generation

We use the same policy for both the Swing-X and Swing-Y sub-systems because they have the same dynamics and cost function (just with different names for the variables). From Section 4.1, we have that the state for the Swing-X system is $\{\mathbf{p}_{w,x}, \dot{\mathbf{p}}_{w,x}, x_{td}, t_{td}\}$. Because x_{td} has constant dynamics, we can reduce the state space of the policy by putting the origin at x_{td}, making the state $\{\mathbf{p}_{w,x} - x_{td}, \dot{\mathbf{p}}_{w,x}, 0, t_{td}\}$. We can drop the 0 from the state and for convenience, we denote the state as $\{p, \dot{p}, t_{td}\}$. We denote our control as $\ddot{p}$. We wish for our controller to smoothly move the foot to the target location ($p = 0$ with the change of coordinates) and arrive there when $t_{td} = 0$ with zero velocity ($\dot{p} = 0$). We will handle the destination requirements in a terminal analytic controller, so we need only minimize acceleration,

$$L(\mathbf{x}, \mathbf{u}) = \left(\frac{\ddot{p}}{\ddot{p}_{\text{max}}}\right)^2, \tag{4.27}$$

in the grid.

Our grid must cover a larger region of state space for this policy than for the Swing-Z policy because the foot moves much farther in the x direction than in the z direction. To keep the computation reasonable, we use two grids: a coarse grid for the entire state space and a fine grid near the goal. The coarse grid has minimum states of $\{$0 m, -4 m/s, 0 s$\}$, maximum states of $\{$1.2 m, 4 m/s, 0.6 s$\}$, and resolutions of $\{$121, 401, 121$\}$. Because the dynamics and cost function are symmetric, we know that $\ddot{p} = \pi(p, \dot{p}, t_{td}) = -\pi(-p, -\dot{p}, t_{td})$. Therefore, we can cut our grid in half and only consider $p \geq 0$. If $p < 0$, we can return $-\pi(-p, -\dot{p}, t_{td})$, which is in our grid. We also have to include this transformation in our dynamics when computing the policy so that simulating forward one time step from, for example, $\{0\text{m}, -1\text{m/s}, 0.1\text{s}\}$ does not end up off of the grid.

The smaller grid has minimum states of $\{$0 m, -3.5 m/s, 0.03 s$\}$, maximum states of $\{$0.175 m, 3.5 m/s, 0.1 s$\}$, and resolutions of $\{$176, 701, 71$\}$. The limits of 3.5 m/s and 0.175 m were found by starting at the origin and working backward for 0.1 s at the maximum acceleration, $\ddot{p}_{\max} = 35\text{m/s/s}$. The two grids overlap for simplicity, but we only use the finer grid in the region where it exists. We can stop the finer grid at 0.03 seconds because we use an analytic controller for $t_{td} \leq 0.03\text{s}$.

Our analytic controller is similar to the one used for the Swing-Z policy. It minimizes

$$C = 4t_{td}\ddot{p}^2 + 2000\dot{p}_f^2, \tag{4.28}$$

which is similar to (4.26), and subject to the constraint that $|p_f| \leq p_{\text{slop}}$ where p_f is the position, p, when $t_{td} = 0$ and $p_{\text{slop}} = 0.0005\text{m}$ allows just enough mismatch to ensure that a fine search (resolution 1 mm) of potential x_{td}'s (or y_{td}'s) will find at least one x_{td} (or y_{td}) that is feasible. The constants in (4.28) are half those in (4.26) because the coefficient of 2 in (4.25) is absent from (4.27).

As for the Swing-Z policy, the touchdown location constraint will be active for most of the state space, and the minimization will only serve to determine in which direction to miss the desired location by p_{slop}. To solve, we first do the minimization, then constrain it by the touchdown location constraint, then constrain it by the maximum acceleration constraint, $|\ddot{p}| \leq \ddot{p}_{\max} = 35\text{m/s/s}$.

Sagittal Policy Generation

During single support, the sagittal policy has a 4-dimensional state space: CoM position relative to the stance foot, CoM velocity, time until touchdown, and touchdown location relative to the stance foot, which we denote as $\{x, \dot{x}, t_{td}, x_{td}\}$. The only control action is the center of pressure location on the stance foot, $|r_x| \leq l_f$, where $l_f = 0.08\text{m}$ is half the length of the foot. For the dynamics, x_{td} is constant, t_{td} counts down, and we find $\ddot{x}$ according to (4.20), which we integrate one time step to update x and $\dot{x}$. For 0.5 m/s walking, we use a cost function of

$$L(\mathbf{x}, \mathbf{u}) = 2\left(\frac{\dot{s}}{v_{\text{des}}}\right)^2 + \left(\frac{r_x}{l_f}\right)^2, \tag{4.29}$$

where v_{des} is the desired walking speed in m/s. We can generate policies for walking slower by decreasing v_{des}, but more complicated cost functions appear to be necessary to walk faster stably. We use a grid with minimum states of {-0.4 m, -0.3 m/s, 0 s, 0 m}, maximum states of {0.4 m, 1.5 m/s, 0.6 s, 0.7 m}, and resolutions of {81, 46, 121, 71}.

During double support, we have a 5-dimensional state space. Time until liftoff, t_{lo}, replaces time until touchdown, stance width, d, replaces x_{td}, and we add weight fraction on the front leg, w, giving us a state of $\{x, \dot{x}, t_{lo}, d, w\}$. For convenience, we measure the CoM position, x, relative to the center point between the two feet during double support. We also add a second action, $\dot{w}$. The dynamics are similar to single support except that we use $\dot{w}$ to update w. Note that we must first get the CoM position relative to the lead foot before we can use (4.20) to get the CoM acceleration. We must add a few extra terms to the cost function, (4.29), to get

$$L(\mathbf{x}, \mathbf{u}) = 2\left(\frac{\dot{s}}{v_{\text{des}}}\right)^2 + \left(\frac{r_x}{l_f}\right)^2 + 0.7\left(\dot{w}T_{\text{DS}}\right)^2 + \left(\frac{x}{v_{\text{des}}T_{\text{DS}}}\right)^2, \tag{4.30}$$

where $T_{\text{DS}} = 0.1\text{s}$ is the nominal duration of double support. The third term is added to limit $\dot{w}$, and the fourth term is added to keep the CoM near the middle of the stance region. Without this cost, the coordinated system tends to continue double support for too long and start dragging its back foot forward when the CoM moves far enough forward that the back leg gets completely straight. We use a grid with minimum states of {-0.2 m, -0.3 m/s, 0 s, 0 m, 0.1}, maximum states of {0.2 m, 1.5 m/s, 0.2 s, 0.7 m 0.9}, and resolutions of {41, 46, 41, 71, 17}. To enforce that $w = 0.9$ at the end of double support, when $t_{lo} \leq 0.01$s, rather than selecting $\dot{w}$ as an action, we use $\dot{w} = (0.9 - w)/t_{lo}$.

At the end of double support, when $t_{lo} = 0$, we transition to single support. At this time, we select the length of the step as an action. To reduce memory costs in implementation, we transition to a special liftoff dynamics for one step (representing zero time) between double support and single support. The liftoff dynamics have only a 2-dimensional state space, $\{x, \dot{x}\}$, and a single action, x_{td}. The grid has the same minimum state, maximum state, and resolution as the first two states of the single support grid. To transition from double support to the liftoff dynamics, we need only change the reference frame for x (from measuring from the center of stance to measuring from the new stance foot). To transition from the liftoff dynamics to single support, we set t_{td} to the nominal single support duration of 0.4 seconds and set the state x_{td} according to the action in the liftoff dynamics. There is no cost for the liftoff dynamics ($L(\mathbf{x}, \mathbf{u}) = 0$).

Coronal Policy Generation

The coronal policy is very similar to the sagittal policy. During single support, the state is identical, except with y instead of x: $\{y, \dot{y}, t_{td}, y_{td}\}$. When the left foot is in single support, the positive-y direction is measured to the right, and vice versa. The only action is the center of pressure location relative to the stance foot, r_y, which is constrained by $|r_y| \leq w_f$, where $w_f = 0.04$m is half the width of the foot. We use the LIPM dynamics (4.5) to compute the CoM acceleration, $\ddot{y}$. Again, t_{td} counts down, and y_{td} is constant. We use the cost function

$$L(\mathbf{x}, \mathbf{u}) = \left(\frac{r_y}{w_f}\right)^2. \tag{4.31}$$

We use a grid with minimum states of {-0.05 m, -1 m/s, 0 s, 0 m}, maximum states of {0.4 m, 1 m/s, 0.6 s, 0.6 m}, and resolutions of {46, 45, 121, 61}.

During double support, t_{td} becomes t_{lo}, y_{td} becomes the stance width, d, and we add the fraction of the vertical force on the next single support leg, w, making the state $\{y, \dot{y}, t_{lo}, d, w\}$. We again add $\dot{w}$ as a second action, which controls the dynamics of w. We again use the LIPM dynamics to find $\ddot{y}$, but now we need to consider how much vertical force is on each foot, w, as well as the shift in CoP due to ankle torque, r_y to find the total CoP. To get the cost function, we add a term limiting $\dot{w}$ to the single support cost function, producing

$$L(\mathbf{x}, \mathbf{u}) = \left(\frac{r_y}{w_f}\right)^2 + 0.7 \left(\dot{w} T_{\text{DS}}\right)^2. \tag{4.32}$$

As in the sagittal sub-system, we enforce that $w = 0.9$ at liftoff by assigning $\dot{w} = (0.9 - w)/t_{lo}$ when $t_{lo} \leq 0.01\text{s}$. During double support, we measure the CoM position, y, relative to the next single support leg with the positive-y direction pointing towards the other foot. We use a grid with minimum states of {-0.05 m, -1 m/s, 0 s, 0 m, 0.1}, maximum states of {0.4 m, 1 m/s, 0.2 s, 0.6 m 0.9}, and resolutions of {46, 45, 41, 31, 17}.

To transition from single support to double support at touchdown, we need to shift the reference frame for y and flip the sign of $\dot{y}$. We also set $t_{lo} = T_{\text{DS}}$, $d = y_{td}$, and $w = 0.1$.

To transition from double support to single support at lift off, we again use a special lift off dynamics as in the sagittal subsystem. Again, it is a 2-dimensional grid with step width for the next step, y_{td} as the only action. Here we have a cost on the step width to encourage reasonable step widths,

$$L(\mathbf{x}, \mathbf{u}) = 0.5 \left(\frac{y_{td} - y_{td,\text{NOM}}}{0.1}\right)^2 \frac{T_{\text{DS}} + T_{\text{SS}}}{\Delta t}, \tag{4.33}$$

where $y_{td,\text{NOM}} = 0.22\text{m}$ is the nominal touchdown width and Δt is the time step. The second fraction is necessary to scale this cost to compensate for the fact that it happens once per step rather than continuously every time step.

These five DP policies (three 3-dimensional and two 4-dimensional policies) are equivalent to a single 10-dimensional DP policy for the entire ICS. If we use a resolution of 100 states per dimension, the coordinated version uses 2.3×10^8 states as opposed to 1.0×10^{20} states for the equivalent single policy. Computing the DP policies is computationally intensive and can take on the order of a day for our 4-dimensional policies. They are computed before use, and this computation does not affect the run-time performance of the controller.

4.2.2 Policy Coordination

Double Support

At run time, we combine the value functions to obtain $\mathbf{x}_d^*$ as in (3.17). During double support, the $\arg\min$ operation is only a 1-dimensional search (we must only find the time until lift off), so it can be performed by a fine resolution brute force search. During single support, however, the search space is 3-dimensional(t_{td}, x_{td}, and y_{td}), so a brute force search would be too computationally expensive.

During double support, we also have to determine how to shift weight from the foot that used to be in single support to the foot that is about to be in single support. Both the sagittal and coronal policies have the fraction of weight on the lead leg w as a state and $\dot{w}$ as an action in double support. Averaging the two commanded $\dot{w}$'s would be the simplest solution, but this will do poorly if one of the policies absolutely needs a specific $\dot{w}$ and the other one can handle just about anything. To tell how important achieving the desired $\dot{w}$ is for a given policy, we must look at the value function.

We generally wish to minimize value, and we already have the value as a function of w, so we find the $\dot{w}$ that reduces the value the most while holding everything else constant, $\min \partial V/\partial t|_{x!=w}$. We can expand this to

$$\frac{\partial V}{\partial t} = \frac{\partial V}{\partial w}\frac{dw}{dt} + \frac{1}{T}L(\mathbf{x}, \mathbf{u}) \tag{4.34}$$

where the first term is the change in value due to changing the state and the second term is the cost of changing the state (T is the time step). If we drop all the terms in L that do not contain w

this gives us

$$\frac{\partial V}{\partial t} = \left(\frac{\partial V_s}{\partial w} + \frac{\partial V_c}{\partial w}\right)\dot{w} + \frac{1}{T}(a_s + a_c)\dot{w}^2 \tag{4.35}$$

where a_s and a_c are the cost function weights for the sagittal and coronal policies. To minimize $\partial V/\partial t$, we take the derivative of this with respect to $\dot{w}$ and set it equal to zero:

$$\frac{\partial V_s}{\partial w} + \frac{\partial V_c}{\partial w} + \frac{2}{T}(a_s + a_c)\dot{w} = 0, \tag{4.36}$$

which we can solve for $\dot{w}$, giving us

$$\dot{w} = \frac{-T\left(\frac{\partial V_s}{\partial w} + \frac{\partial V_c}{\partial w}\right)}{2(a_s + a_c)}. \tag{4.37}$$

This essentially gives the ratio of the benefit of changing w (change in value in the numerator) to the cost of changing w (cost function terms in the denominator).

Parabolic Approximation Method

To speed up the search, we note that all five value functions depend on t_{td}, but that only 2 each depend on x_{td} and y_{td}, and that none of the value functions depend on both x_{td} and y_{td}. We wish to first find $x^*_{td}(t_{td})$ and $y^*_{td}(t_{td})$, so that we can then perform a 1-dimensional search over $V(t_{td}|x^*_{td}(t_{td}), y^*_{td}(t_{td}), \mathbf{x}_f)$.

To do this, we approximate the value functions (during pre-computation) in such a way that they can be added quickly and that $x^*_{td}(t_{td})$ and $y^*_{td}(t_{td})$ of the sums can be found analytically. For the coronal and swing-y value functions, we approximate the value function, $V(t_{td}, y_{td}|\mathbf{x}_i)$, with a series of parabolic approximations to $V(y_{td}|t_{td}, \mathbf{x}_i)$ for evenly spaced values of t_{td}. Each parabola is created by placing the vertex at the minimum of $V(y_{td}|t_{td}, \mathbf{x}_i)$ and using a point to either side to estimate the second derivative. Figure 4.7 shows an example surface approximation. The sum of two parabolas is also a parabola, so two surfaces can then be added quickly by adding the parabolas, creating a new surface also represented by a series of parallel parabolas. The location of the vertex of each of these new parabolas gives us $y^*_{td}(t_{td})$, and the value at each vertex gives us $V_{C+Y}(t_{td}|y^*_{td}, \mathbf{x}_f)$, where V_{C+Y} indicates the value for the coronal and swing-y subsystems together.

To do this quickly at run-time, we must first pre-compute the parabolic approximations to $V(y_{td}|t_{td}, \mathbf{x}_i)$ for each of the policies. We compute the approximation (as shown in Figure 4.7) for a grid of $\mathbf{x}_i$ with the same resolution as the main DP grid. At run time, when we look up the surface approximation, we use the actual $\mathbf{x}_i$ and multilinear interpolation on the grid of approximations. We have found that parameterizing the parabola as

$$y = a(x - h)^2 + k \tag{4.38}$$

and interpolating the parameters a, h, and k works better than parameterizing the parabolas as

$$y = ax^2 + bx + c \tag{4.39}$$

and interpolating the parameters a, b, and c because the former parameterization does a better job of capturing the things we care about: the location and value of the vertex. It is worth noting that these parabolic approximations are both much easier to compute and smaller to store than the original DP value function, so using them does not add significantly to our pre-computation requirements.

The same is done with the sagittal stance and swing-x value functions, using x_{td} instead of y_{td}. With the value functions reduced to only a function of t_{td} as shown in Figure 4.8, they can be efficiently added

$$V(t_{td}|x^*_{td}, y^*_{td}, \mathbf{x}_f) = V_{C+Y}(t_{td}|y^*_{td}, \mathbf{x}_C, \mathbf{x}_Y) + V_{S+X}(t_{td}|x^*_{td}, \mathbf{x}_S, \mathbf{x}_X) + V_Z(t_{td}|\mathbf{x}_Z) \tag{4.40}$$

to produce a value dependent only on t_{td}. This value function is represented as a series of values, $V(t_{td}|x^*_{td}, y^*_{td}, \mathbf{x}_f)$, with the same spacing as the parabolas in the surface approximation. We pick the point with the lowest value, then fit a parabola to it and its two neighbors. The location of the fit parabola's vertex gives us t^*_{td}. We can then plug t^*_{td} into the $x^*_{td}(t_{td})$ and $y^*_{td}(t_{td})$ functions we found earlier to look up x^*_{td} and y^*_{td}. Having determined $\hat{\mathbf{x}}_d$, we can now look up the appropriate controls, $\mathbf{u}_f$, from the individual policies.

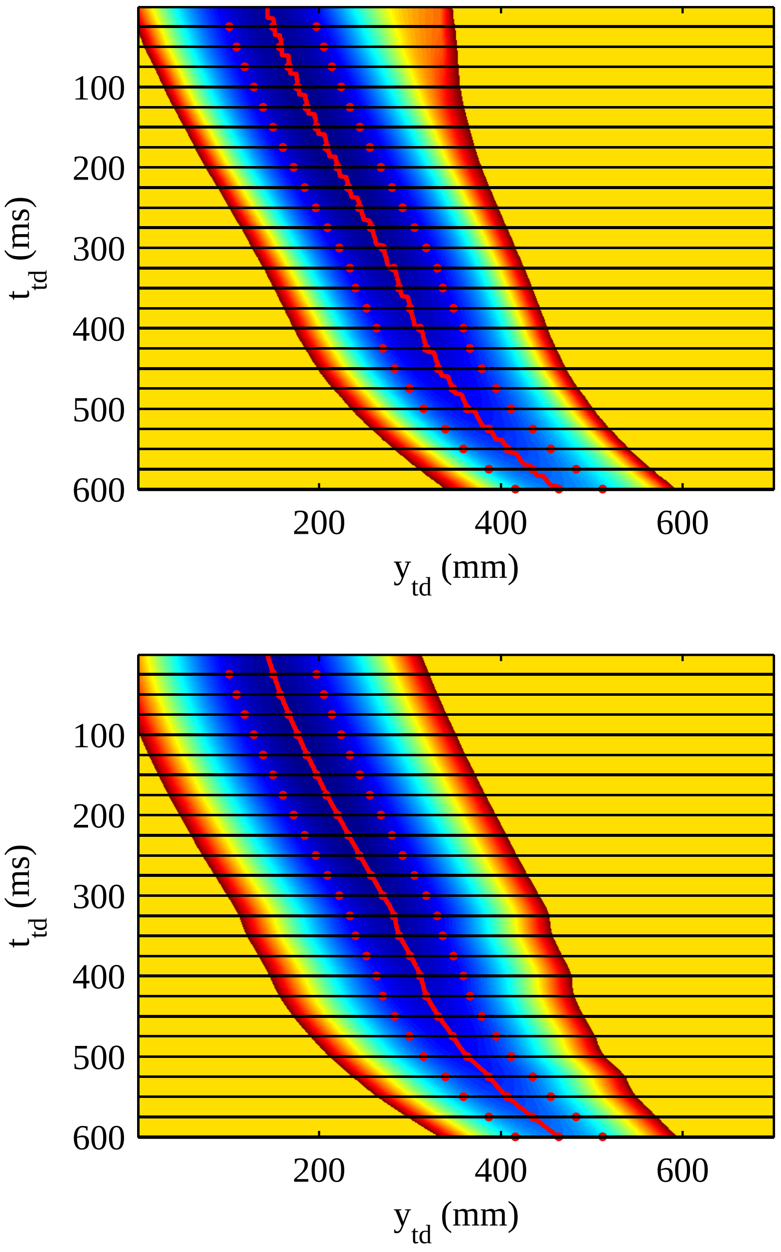

Figure 4.7: The coronal stance value function, $V(t_{td}, y_{td}|\mathbf{c}_y = 0.08, \dot{\mathbf{c}}_y = 0)$, from the DP tables (top) and from the parabolic approximation (bottom). The red line shows $y^*_{td}(t_{td})$. The dots show the points used to generate the parabolic approximation, and the horizontal black lines show the location of the parabolas.

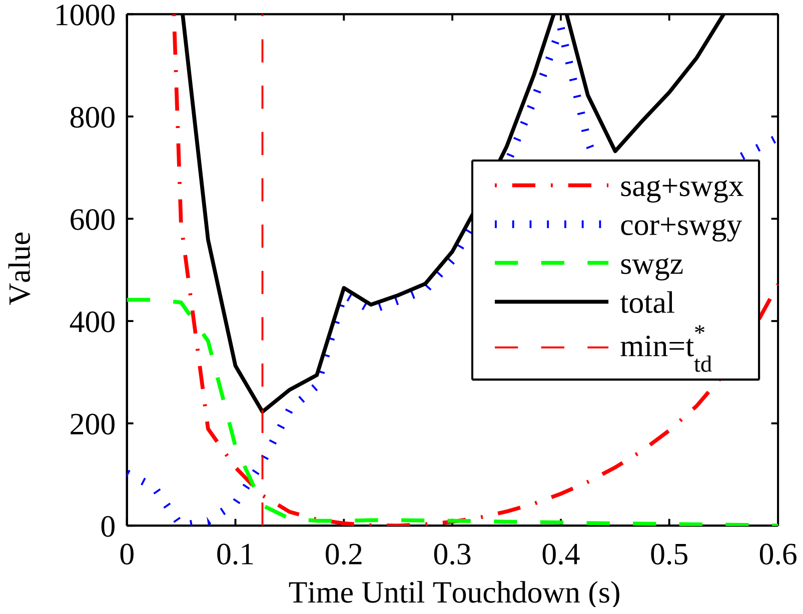

Figure 4.8: Value as a function of t_{td} following a push to the side. For ease of view, values are normalized so that the minimum is 0. Due to the push, the coronal policy wants to touch down soon, but it must compromise with the other policies to pick the best time, t^*_{td} for the full ICS.

The run-time computation for this method is extremely simple, and requires only about 6 microseconds of computation time on a 6 core 3.33 GHz Pentium i7 processor.

Unfortunately, this method relies on the structure of the value functions produced by dynamic programming; if they do not have the form we expect, then our approximations will not be sufficiently accurate to find a good solution. If, for example, we wish to handle varied terrain, we would modify the value function by using a non-zero h function in (3.7). For complicated terrain, this can make the final cost function arbitrarily complex.

Optimization Method

To handle situations where we do not have the simple form of the value functions for which the convenient approximations work, we use a more generic method. We treat finding $\hat{\mathrm{x}}_d = \{t^*_{td}, x^*_{td}, y^*_{td}\}$ as a generic optimization problem and use the Nelder-Mead method [72] to find the optimum. To avoid local minima, we start optimizations from several points each time step (and select the result with the lowest value): four random points and the best guess based on the previous time step. The best guess is simply the same as the solution from the previous time step with t_{td} decremented by the timestep, except at the beginning of single support, when a default value is used. Even if the global optimum has a small basin of attraction, after several time steps, one of the random starts is very likely to start in it and find the global minimum. The controller is continuously updating its touchdown target, so it need not know where it will touch down at the beginning of the step so long as it figures it out quickly.

For the simple case of flat ground, but with some regions we are not allowed to step, there is still some problem structure that we can exploit to speed up the optimization and help find the global optimum step location as early in the step as possible. We implement the stepping requirement by modifying the h function in (3.7). We set $h(\mathbf{x}_f(t_j)) = \infty$ for $\mathbf{x}_f(t_j)$'s that correspond to stepping in illegal locations (where touchdown occurs at t_j) and $h(\mathbf{x}_f(t_j)) = 0$ for all other values of $\mathbf{x}_f(t_j)$. For any situation where the terrain imposes costs dependent on stepping location, we can expect an arbitrarily complicated dependence of the value on x_{td} and y_{td}, but still expect the same smooth behavior of $V(t_{td}|x_{td}, y_{td})$. Additionally, optimization will not work if it starts in an illegal region because it will not have a gradient (all points on the simplex will have the same value). We therefore start by randomly sampling only $\{x_{td}, y_{td}\}$ and checking to make sure it is a legal location. We resample new points until we find a legal stepping location. We then attempt to speed up the optimization and avoid local minima by picking the best (lowest value) t_{td} from a list of 19 choices. We use a list rather than even spacing so that we can have closer spacing for small t_{td}'s, where minima can have narrower basins of attraction. We then form a simplex with this point as one of the vertices and run the optimization. Choosing initial t_{td}'s in this way rather than randomly sampling does run the risk of systematically missing the global minimum because the nearest choice in the list is not the lowest of the list values. However, since we expect smooth behavior of $V(t_{td}|x_{td}, y_{td})$, this is unlikely. On the other hand, using this method often avoids local minima and therefore improves the chance of finding the global minimum early in the step.

The optimization method of coordination of policy coordination is more flexible than the parabolic approximation method, but it is also much slower. It takes an average of 586 microseconds on a 6 core 3.33 GHz Pentium i7 processor (compared to 6 microseconds for the parabolic approximation method). Fortunately, it is trivial to multithread; we can run the optimization from each of the five starting points in a different thread. This allows us to take advantage of multiple cores. In practice, multithreading decreased computation to an average of 325 microseconds, which is reasonable as a component of a controller that we wish to run at 1 kHz.

In practice, we tend to get slightly worse robustness (can withstand slightly smaller pushes) with the optimization method. We believe this is because the parabolic approximation provides some smoothing to our value functions. They can be somewhat noisy because we stop optimizing our policies without them converging. Additionally, we use a coarse grid, which can result in weird effects near the border between where the value is infinite (recovery is impossible) and where it is finite.

4.2.3 Low-Level Control

The output of the high-level controller is the desired horizontal CoM acceleration, $\ddot{\mathbf{c}}_{x,des}$ and $\ddot{\mathbf{c}}_{y,des}$, as well as the desired swing foot acceleration $\ddot{\mathbf{p}}_{w,des}$. The objective of the low-level controller is to generate joint torques which will achieve these accelerations as well as enforce the constraints assumed by the high-level control. The sagittal policy assumes the CoM will be at a height dependent on the stance width and horizontal CoM position, so we use a feedforward vertical acceleration as well as PD gains on $\mathbf{c}_z$ and $\dot{\mathbf{c}}_z$ to compute $\ddot{\mathbf{c}}_{z,\text{des}}$. Both the sagittal and coronal policies assume a vertical torso to approximately enforce the $\dot{\mathbf{L}} = 0$ constraint; it also keeps the hips approximately stationary relative to the CoM, which ensures that the reachability constraints on the feet imposed by full extension of the legs can be accurately represented in the simple CoM models. We therefore use PD gains on the torso angle and angular velocity to compute desired total moment.

It is important to note that the high-level controller does not generate trajectories. Instead, it maps directly from system state to desired accelerations without maintaining any controller state. This allows it to react to perturbations and accumulated modeling error in real time, but it also means that we do not have desired positions or velocities, which precludes the use of traditional trajectory tracking techniques. In the place of trajectory tracking, we use a form of inverse dynamics to generate joint torques. The feedback gains of our controller are embedded in the gradients of the high-level DP policies.

Dynamic Balance Force Control

Based on the desired CoM acceleration and total moment, it is straightforward to compute the desired total reaction force, $\mathbf{f}$, and torque, $\boldsymbol{\tau}$. During double support, we divide the total reaction force between the two feet while enforcing the CoP (4.22) and friction (4.23), (4.24) constraints by minimizing

$$\begin{aligned} C = & \frac{\mathbf{f}_{L,x}^2}{\mathbf{f}_{L,z}} + \frac{\mathbf{f}_{R,x}^2}{\mathbf{f}_{R,z}} + \frac{\mathbf{f}_{L,y}^2}{\mathbf{f}_{L,z}} \frac{\mathbf{f}_{R,y}^2}{\mathbf{f}_{R,z}} + \\ & a\left(\frac{\boldsymbol{\tau}_{L,x}^2}{\mathbf{f}_{L,z}} + \frac{\boldsymbol{\tau}_{R,x}^2}{\mathbf{f}_{R,z}} + \frac{\boldsymbol{\tau}_{L,y}^2}{\mathbf{f}_{L,z}} \frac{\boldsymbol{\tau}_{R,y}^2}{\mathbf{f}_{R,z}}\right). \end{aligned} \tag{4.41}$$

This cost function has the useful property that it produces the same CoP offset for both feet, ensuring that there is as much margin as possible between the CoP and the edge of the foot.

We then use Dynamic Balance Force Control (DBFC) as presented in [94] to generate joint torques, $\boldsymbol{\tau}_j$. DBFC uses a weighted pseudo-inverse with regularization to solve

$$\begin{bmatrix} \mathbf{M}(\mathbf{q}) & -\mathbf{S} \\ \mathbf{J}(\mathbf{q}) & 0 \\ \epsilon_1 I & 0 \\ 0 & \epsilon_2 I \end{bmatrix} \begin{pmatrix} \ddot{\mathbf{q}} \\ \boldsymbol{\tau}_j \end{pmatrix} = \begin{pmatrix} N(\mathbf{q}, \dot{\mathbf{q}}) + \mathbf{J}(\mathbf{q})\hat{\mathbf{f}} \\ -\dot{\mathbf{J}}(\mathbf{q})\dot{\mathbf{q}} + \ddot{\mathbf{p}} \\ 0 \\ 0 \end{pmatrix} \tag{4.42}$$

where $\mathbf{q}$ is a vector of generalized coordinates including 6 values specifying the position and orientation of the base and 12 joint angles, $\mathbf{M}(\mathbf{q})$ is the mass matrix, $\mathbf{J}(\mathbf{q})$ is the Jacobian of both feet, $\mathbf{S} = [0, \mathbf{I}]$ selects the actuated elements of $\mathbf{q}$, $\hat{\mathbf{f}} = [{\mathbf{f}_L}^\mathsf{T}, {\boldsymbol{\tau}_L}^\mathsf{T}, {\mathbf{f}_R}^\mathsf{T}, {\boldsymbol{\tau}_R}^\mathsf{T}]^\mathsf{T}$, and $\ddot{\hat{\mathbf{p}}} = [\ddot{\mathbf{p}}_L^\mathsf{T}, \ddot{\mathbf{p}}_R^\mathsf{T}]^\mathsf{T}$. The bottom two sets of equations provide regularization and $\epsilon_1 = 1.0 \times 10^{-5}$ and $\epsilon_2 = 1.0 \times 10^{-5}$ are small constants. It can be computed quickly, and the entire low-level controller requires only about 36 microseconds of computation time on a 6 core 3.33 GHz Pentium i7 processor.

Since we do not use PD joint torques in addition to the DBFC output, even small errors in the foot acceleration produced can accumulate over time. In order to more accurately match desired foot accelerations, we add an integrator on foot acceleration to our low level controller,

$$\ddot{\mathbf{p}}_{w,\mathrm{DBFC}} = \ddot{\mathbf{p}}_{w,\mathrm{HL}} + K_I \int (\ddot{\mathbf{p}}_{w,\mathrm{HL}} - \ddot{\mathbf{p}}_w)\, dt, \tag{4.43}$$

where $\ddot{\mathbf{p}}_{w,\mathrm{DBFC}}$ is the swing foot acceleration used by the low-level DBFC controller, $\ddot{\mathbf{p}}_{w,\mathrm{HL}}$ is the swing foot acceleration produced by our high-level policies, and K_I is the integral gain. The constant $K_I = 3.0$ was found by experimentation to work well. It gives a time constant shorter than a step, making it large enough to quickly correct for errors/disturbances but small enough not to produce instability. The integrator is not necessary for stable walking, but it significantly improves the robustness to perturbations.

DBFC Modes

Dynamic Balance Force control, as described above, fails when the system is at or near a kinematic singularity (e.g. straight knees). Rather than simply avoiding these situations, we use a few modifications to the basic DBFC algorithm that each allow us to handle a specific situation.

During single support, we wish to keep the stance leg at a fixed length (fixed knee angle). The knee does have to be somewhat bent to allow for extension during double support (discussed in Section 4.1), but we do keep it mostly straight. Using the regular DBFC, we would use forward kinematics to get a desired CoM height, use feedback based on the actual CoM height to get a desired vertical foot force, and use DBFC to get joint torques. Essentially, we would be attempting to control knee angle by commanding a desired vertical force. Unfortunately, with the knee mostly straight, the Jacobian is nearly singular, so we have very little control over that vertical force. The result is that slight inaccuracies in our model or small perturbations can result in large errors. If what we really care about is knee angle, it is simpler to command it directly.

To directly command knee angle, we replace the regularizing equation for stance knee acceleration with a high gain PD controller that attempts to servo the stance knee to a fixed angle. It is contradictory to independently control stance knee angular acceleration and vertical force, so rather than commanding a desired force, we make the vertical force, $\mathbf{f}_{x,z}$, an additional variable. To do this, we must move the appropriate column of the Jacobian, $\mathbf{J}(\mathbf{q})$ from the right side of (4.42) to a new column in the large matrix on the left hand side.

We also have to handle two situations where the knee can become completely straight: 1) If the CoM moves too far forward during double support, the back knee can get completely straight. 2) If it attempts to step too far forward, the front knee can become completely straight. In either case, the result is a kinematic singularity and therefore a singular Jacobian, which causes DBFC to fail. The policies are set up with appropriate constraints such that these situations do not occur during normal walking. However, if there is a large perturbation, they can happen, especially if the perturbation results in large torso rotations, which move the hip horizontally relative to the CoM.

If we find that the swing knee has become almost straight while attempting to reach its touchdown location, we rotate the three equations governing the translational acceleration of the swing foot (part of the second set of equations in (4.42)). We rotate the three equations from $\{x, y, z\}$ coordinates to coordinate system with one of the basis vectors pointing from the swing foot to the hip. We can then drop the equation for acceleration in that direction, leaving acceleration in the direction we can not control unspecified. We then directly control the swing knee by replacing a regularization equation with a PD controller on knee acceleration that servos it to a constant slightly bent angle (similar to what we did for the stance knee in single support). This way, the foot will touch down at the desired location as soon as the hip moves far enough forward that it can reach.

We use a nearly identical method to recover if the back leg becomes straight during double support. In that case, however, rather than servoing the knee to a fixed angle, we bend it until we are far enough from the kinematic singularity that normal control can resume.

Weighted-Objective Inverse Dynamics

An alternative low-level controller that can be used instead of DBFC is the Weighted-Objective Inverse Dynamics adapted from [93]. To use the DBFC controller, we first determine the desired contact forces and torques at each foot. Then DBFC uses a pseudo-inverse to find the least squares solution to a set of equations to get the joint torques and accelerations. It is important to realize that it had the same number of variables as it had equations with large weights (not counting the regularization equations). Essentially, it functioned as a perfectly determined system (same number of equations and variables) with some regularization rather than as an overdetermined system. For the Weighted-Objective Inverse Dynamics, we combine the two steps by simultaneously solving for the joint accelerations, joint torques, and contact forces/torques. We also wish to handle constraints such as the ZMP constraints, friction cone constraints, torque limits, and joint limit constraints, so we solve using off-the-shelf Quadratic Programming software [30]. We implement joint-limit constraints by constraining the joint acceleration when near the limit.

In addition to rearranging the 60 equations (18 dynamics equations, 12 foot acceleration equations, 18 acceleration regularization equations, and 12 torque regularization equations) in (4.42), we add many new equations. We add 1 equation for the vertical weight distribution between the two feet, 3 equations for CoM acceleration, 3 equations for the total moment around the CoM, 12 equations to regularize the contact forces, 3 equations to control torso angular acceleration, 2 equations for the CoP, and 12 equations to accelerate towards the reference pose produced by the inverse kinematics, q_{IK} (described in the next Section). The equations for tracking the reference pose look like

$$\ddot{\mathbf{q}}_i = k_p(\mathbf{q}_{\mathrm{IK},i} - \mathbf{q}_i) + 2\sqrt{k_p}((\dot{\mathbf{q}}_{\mathrm{IK},i} - \dot{\mathbf{q}}_i) - \mathbf{q}_{g,i}, \tag{4.44}$$

where the subscript i indicates the ith joint, $\mathbf{q}_{\mathrm{ref}}$ is the reference pose, $\mathbf{q}_g$ is the joint acceleration induced by gravity, and k_p is a proportional gain. The $2\sqrt{k_p}$ term ensures that this acceleration is critically damped.

This gives us a system of 42 variables and 96 equations, which we write as

$$\mathbf{A}\begin{pmatrix} \ddot{\mathbf{q}} \\ \boldsymbol{\tau}_j \\ \mathbf{f} \end{pmatrix} = \mathbf{b}, \tag{4.45}$$

where $\mathbf{A}$ is a 96 by 42 matrix and $\mathbf{b}$ is a 96-vector. Even without the regularization equations, this is an overdetermined system. In order to modify the relative importance of each equation, we multiply both sides of (4.45) by a diagonal weighting matrix, $\mathbf{W}$, which multiplies each row of $\mathbf{A}$ and $\mathbf{b}$ by the corresponding weight,

$$\mathbf{WA}\begin{pmatrix} \ddot{\mathbf{q}} \\ \boldsymbol{\tau}_j \\ \mathbf{f} \end{pmatrix} = \mathbf{Wb}. \tag{4.46}$$

We have selected the actual weights manually through experimentation. To turn this into a QP, we construct $\mathbf{A}^{\mathsf{T}}\mathbf{WWA}$ as the quadratic cost matrix and $-b^{\mathsf{T}}\mathbf{WA}$ as the linear cost matrix.

In simulation, either low-level controller can be used nearly interchangeably. Since it is exactly constrained rather than over constrained, the DBFC controller tends to more accurately achieve the commands of the high-level controller, but degrades less gracefully once that becomes impossible. On the robot, however, WOID performs much better. DBFC can be used for simple standing, but WOID responds better to perturbations because optimizing the contact forces at the same time as joint torques and accelerations allows for a larger range of tradeoffs to be considered. For complicated tasks such as walking on the real robot, DBFC fails completely because it is difficult to prevent it from attempting to control the hardware too aggressively. WOID has the additional advantage that it can be more easily extended to control more than 12 joints. It is more computationally expensive, requiring 187 microseconds rather than 36 microseconds for DBFC, but it is still easily fast enough.

Peel Mode for WOID

While WOID can function near kinematic singularities, we still need to use a structural change to handle certain situations. If the rear leg fully extends (straight knee) during double support before liftoff, the foot must rapidly accelerate from stationary to the approximate velocity of the hip, rendering nearly any desired foot acceleration produced by the high-level controller impossible to achieve. In this case, we remove the equation specifying foot acceleration in the forward (x) direction, while retaining the equation that specifies vertical (z) foot acceleration. We also add an equation requesting a large knee acceleration to rapidly bend the knee and quickly get us to a situation where we can achieve desired foot accelerations again. These structural changes help ensure that some solution exists.

4.2.4 Selecting Constants

Our controller contains a large number of constants that must come from somewhere. One of the advantages of optimal control, however, is that most of the constants either have physical meaning (e.g. desired walking speed or swing foot height) or have an intuitively easy to understand effect (e.g. cost function weights). We have generally found that the behavior in simulation is not unduly sensitive to the exact value of the cost function weights and will work for a fairly wide range. Running the controller on the robot (as discussed in the next Chapter) tends to require somewhat more careful tuning of the various weights and gains. In most cases, the DP cost function terms have different units, so we start out by normalizing each term by some maximum reasonable value. Often, that is sufficient and we can get adequate performance using all normalized weights of 1.0. From there, we can adjust the cost function for better performance in a few iterations (usually one to four). Normalized and unnormalized weights can be seen in Table 4.1.

Each term in the WOID algorithm also has a manually selected weight, but they also do not require precise tuning. They primarily provide a rough ordering of priorities for which terms to violate. They only become important when the high level commands are contradictory and can not all be satisfied simultaneously.

The grid bounds and resolutions are a straightforward tradeoff between performance and (offline) computation time. However, once they have sufficient size and resolution, more does not generally help. We rely on intuition to pick grid bounds and resolution that are sufficient and only change them if they are an obvious cause of failure.

4.3 Capabilities

To be useful, a walking system must be able to do more than walk in a straight line. It must be able to start from rest, walk at varying speeds, turn, and stop. It also must be able to handle walking on complex terrain in addition to flat ground. Our controller can avoid specified regions where it is not allowed to step. It can also handle slopes and step changes in ground height, but because we have only tested these sorts of terrain when the controller was unaware of them, we cover them in the following section on robustness.

4.3.1 Speed

Walking speed can be changed by switching the sagittal stance policy to one computed with a different v_{des}. The policies are global, so no transition is necessary, and the policies can be switched at any point during the step. Similarly, the system can start from rest and achieve steady state walking without switching policies. Figure 4.9 shows how the velocity varies after changing policies. We do not exactly achieve the desired speed because it is part of a cost function that

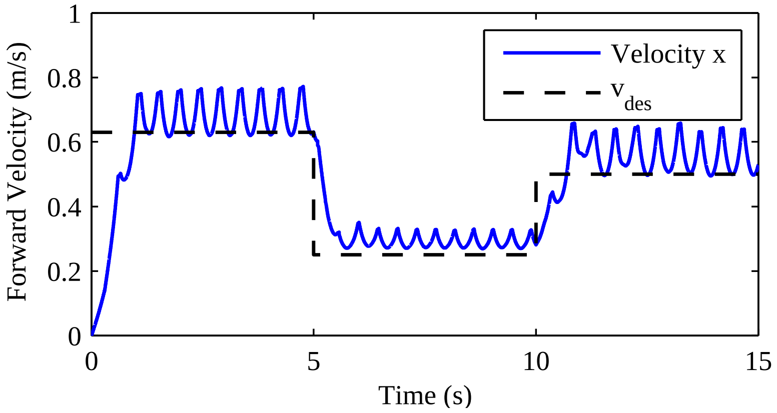

Figure 4.9: Forward speed as the sagittal stance policy is changed. Starts from rest with $v_{\text{des}} = 0.63$, switches to $v_{\text{des}} = 0.25$ after 5.0 seconds, and to $v_{\text{des}} = 0.5$ after 10.0 seconds.

also has other terms such as ankle torque (effort). Additionally, the full model does not behave exactly like the simple model used to generate the policy (in part due to angular momentum, as discussed in detail in Section 4.6).

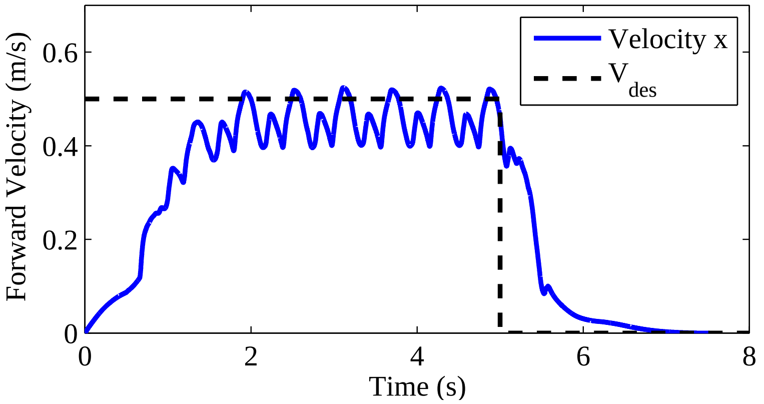

Figure 4.10: Forward speed as the system starts from rest, walks with $v_{\text{des}} = 0.5$, and stops by switching to a policy generated with $v_{\text{des}} = 0.0$. We switch to the stopping policy after 5.0 seconds, after which it takes about 1.5 steps to stop. During steady walking, the large humps in the speed occur during single support and the small humps during double support.

If the system is walking slowly enough, it can stop simply by switching from a walking high level controller to a standing high level controller and continuing to run the same low level controller. If the switch is done during double support, it will simply stop, but if it is done during single support, it will stomp its swing foot as it tries to put weight on a foot that is not on the ground. This does not result in failure. However, if it is walking forward at any significant speed, it is necessary to slow down before stopping. To slow down, we switch to a sagittal policy generated with $v_{\mathrm{des}} = 0.0$. This quickly brings the system to a stop and remains in an infinitely long double support phase, as shown in Figure 4.10. The normal coordination procedure results in a permanent double support phase, though it does stand with its CoM slightly off center because the coronal policy thinks it is going to step again. To achieve balanced standing, it is necessary to switch to a standing controller once the system has come to a stop.

4.3.2 Turning

The walking controller is designed to always face and walk forwards. The controller can turn by simply changing which direction "forward" is. We rotate the coordinate frame that the robot state is measured in while still having the controller attempt to walk in the positive x direction. Practically this means rotating all vectors in the robot state (positions, velocities, and accelerations) and adding to the yaw Euler angle, then reconstructing a new quaternion from the modified Euler angles. Using this method, we can turn at up to about 1.5 radians/second while walking forward with $v_{\mathrm{des}} = 0.5\mathrm{m/s}$. If the robot were a point and it truly was walking at 0.5 m/s, this would correspond to walking in a circle of about 2/3 m diameter. To walk in a tighter circle or turn in place, we have to set a slower forward walking speed (0 m/s for turning in place).

The simulated system retains much of its robustness while turning. It can withstand a push of 38 Ns to the inside of its turn as compared to a 41 Ns push of the same direction and timing when walking forward. It can withstand a push of 21 Ns to the outside of its turn as compared to a 33 Ns push of the same direction and timing when walking forward.

We also attempted to improve turning performance by adding centripetal acceleration to the

CoM and the swing foot as offsets to what the normal controller outputs. Neither modification proved beneficial. In the case of adding centripetal acceleration to the CoM, we found decreased robustness, and for adding acceleration to the swing foot, we found that the system failed to walk entirely. It is possible that centripetal acceleration will play a bigger part and therefore these modifications will be more beneficial when walking at faster speeds.

We were able to get slightly improved robustness by rotating the swing foot to the average desired orientation during the next step rather than servoing it to the current heading. This way the stance foot matches the heading at the middle of the step rather than the beginning. This modification increased the size of the push that could be withstood from 21 Ns to 24 Ns for pushes to the outside of the turn without having any affect on pushes to the inside of the turn.

4.3.3 Terrain

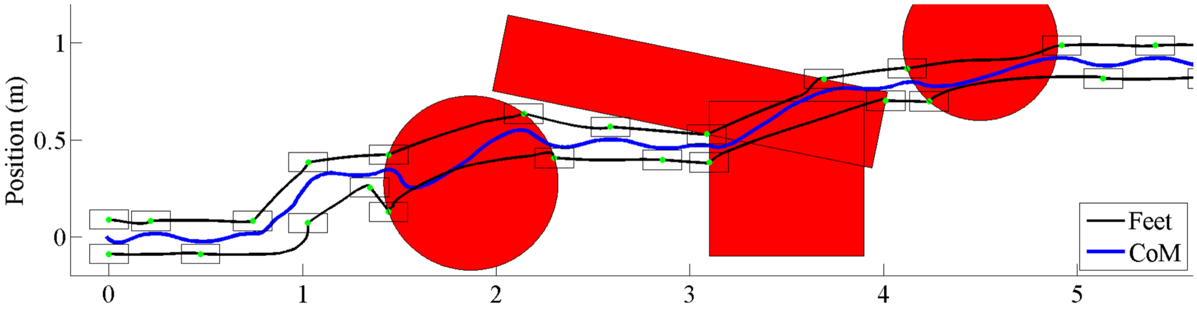

Figure 4.11: The footstep pattern as the system starts from rest, gets pushed (a 30Ns push to the system's left during the 4th step), and avoids obstacles (red regions). The red regions represent areas where the center of the foot (green dots) may not be placed, though the foot may overlap with the red region.

We are able to avoid stepping in regions that are specified as regions we are not allowed to step in. Switching from the coordination method based on parabolic approximations to the coordination method based on a generic optimization (both described in Section 4.2.2) gives us the flexibility to add an arbitrary cost function for where we are stepping. For the case of avoiding specified regions, we add an infinite cost for stepping in the forbidden regions. Figure 4.11 shows the footstep pattern as the controller avoids obstacles. The largest gap that the system can step over is 48 cm, which corresponds to a step of 68 cm: the 48 cm gap plus the 20 cm length of the foot. This same method can also be used if we have a general terrain cost map. Such a cost map could potentially be generated from local features of the terrain, and Inverse Optimal Control could potentially be used to find the weights for these features [111].

4.4 Robustness

An important characteristic of any controller is its ability to reject perturbations. In particular, the size of the largest disturbance that does not cause the system to fail is a useful metric for systems where failure is well defined. One practical difficulty with using this as a metric of performance for walking is that there are many several types of disturbance. We discuss here the robustness of our controller to several types of disturbance.

4.4.1 Pushes

A major type of disturbance experienced by walking systems are pushes. We apply pushes to the torso center of mass, which is about 1.25 m high during walking (30 cm above the system CoM). Figure 4.12 shows the effect of push angle and timing on the maximum survivable perturbation. Force perturbations lasting 0.1 seconds are administered to the torso CoM at various angles while the system is walking with a nominal speed of 0.5 m/s. Data is shown in Figure 4.12 for perturbations beginning at increments of 0.1 seconds after the left foot lifts off. These pushes are shorter than most of the system dynamics, so we consider them to be essentially impulsive. We therefore measure their magnitude in units of impulse, Ns.

We also tested our system with more prolonged pushes lasting 3.0 seconds. Since the pushes last for multiple steps, the exact time within a step at which it begins is not important. Figure

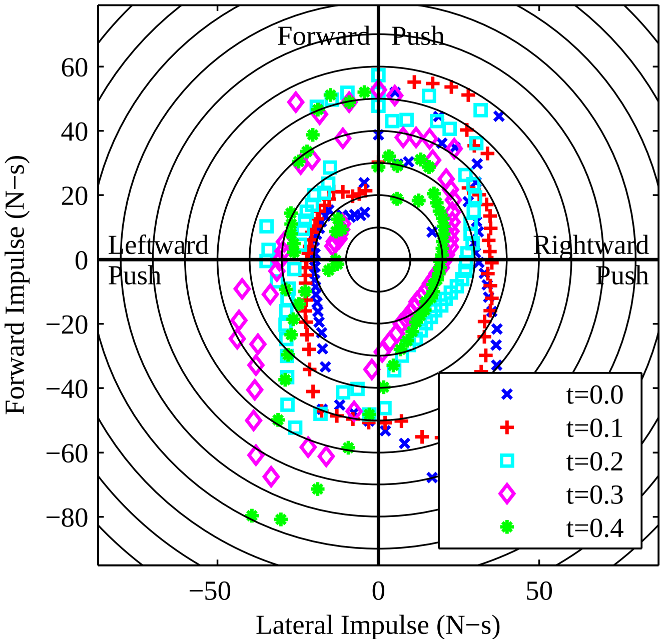

Figure 4.12: Polar plot of the maximum survivable 0.1 second push for our walking simulation as a function of angle and timing. Data is shown for perturbations occurring at various times after left foot lift off. A point represents the maximum survivable perturbation in a given direction. Concentric circles are in increments of 10 Newton-seconds.

4.13 shows these results. Note that for the longer pushes, we measure the magnitude in force (Newtons).

4.4.2 Slips

Our controller can walk on surfaces with a coefficient of static friction $\mu_s \geq 0.35$. Once slipping begins, we model friction as having a coefficient of kinetic friction, $\mu_k < \mu_s$. The exact value of this coefficient has almost no effect on whether slipping results in a fall: with $\mu_k = 0.8\mu_s$, we found that the minimum μ_s was 0.35, but with $\mu_k = 0.3\mu_s$, we found that the minimum μ_s was 0.36.

It can tolerate small regions of even lower friction, but only very small regions. (We simulate

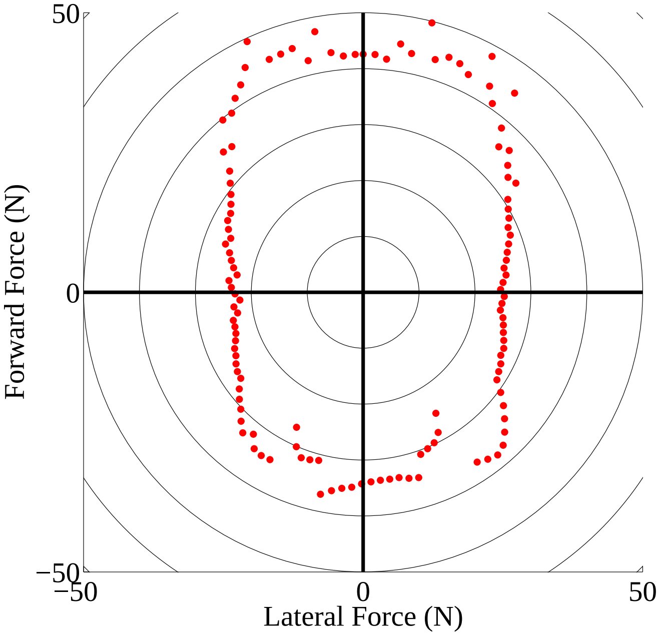

Figure 4.13: Polar plot of the maximum survivable 3.0 second push for our walking simulation as a function of angle. Data is shown for perturbations occurring at various times after left foot lift off. A point represents the maximum survivable perturbation in a given direction. Concentric circles are in increments of 10 Newton-seconds.

changes in friction by making μ dependent on the location of the center of the foot, without averaging over the entire foot.) Counter-intuitively, it can tolerate larger regions of low friction if the friction is lower: 2.5 cm for $\mu_s = 0.01$, 1.5 cm for $\mu_s = 0.1$, and less than 1 cm for $\mu_s = 0.15$. The reason for this unexpected behavior is that the failure mode is different from that found in human walking. When the foot is first put down in a low-friction area, it slides slightly forward during double support, which generally does not lead to failure. Then, during single support, it slides backward rapidly, which does result in failure. Once the foot starts slipping backward, it does not recover even if it gets onto a higher friction area. However, if the low-friction area is very low friction, it will slide forward farther during double support, and be more likely to slip onto a higher-friction area before slipping in the other direction during single support.

It is likely that special sagittal and coronal policies generated with appropriate friction constraints would allow for successful very-low friction walking. Alternately, a simple traction-control controller decreasing the ground reaction force once slipping was detected would likely help somewhat.

4.4.3 Trips

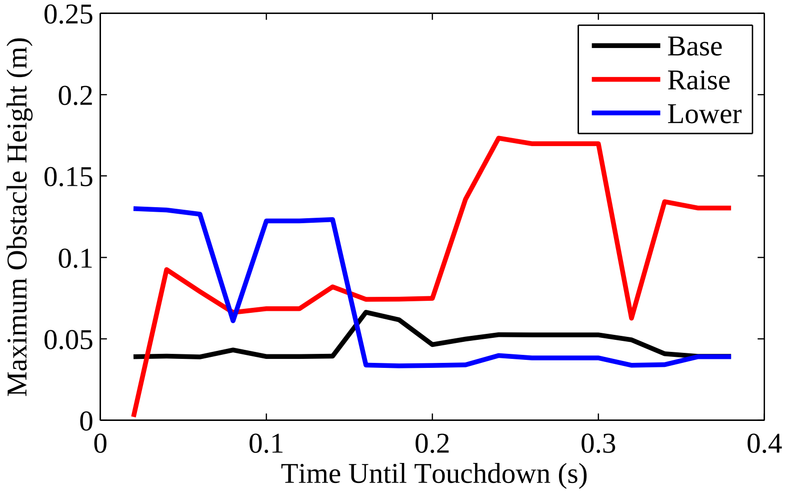

Figure 4.14: The largest tripping obstacle that our controller can handle as a function of when it is contacted during the step. Results are shown for the baseline controller, an explicit raising strategy, and an explicit lowering strategy.

We also tested our controller's response to tripping obstacles. The obstacles were wide enough to obstruct one, but not both, feet, of varying heights, and only 1.0 cm long (the foot must be lifted, moved forward one foot length plus 1.0 cm, then lowered). The controller does not know about the obstacles ahead of time - they are detected by the anomalous foot force, but we do assume it knows when it has raised its foot high enough to clear the obstacle. We also assume that the front face of the obstacle is frictionless, meaning the toe can slide up or down it easily.

Figure 4.14 shows the maximum height of tripping obstacles our controller can handle as a function of when in the step they are contacted. It shows result for the baseline walking strategy as described in Section 4.2. It also shows results for a simple raising strategy and a simple lowering strategy.

For the raising strategy, when the obstacle is detected (via anomalous foot forces), the high level controller immediately overrides the output of its Swing-X and Swing-Z policies. Instead, it commands zero acceleration in the x direction and a large upwards acceleration (25 m/s/s) in the z direction. Once the foot has cleared the obstacle vertically, the Swing-X policy is used as normal, but the Swing-Z policy acceleration is ignored in favor of servoing the foot to 0.5 cm above the obstacle. Once the foot has passed the obstacle horizontally, control returns to normal.

For the lowering strategy, we again override the Swing-X and Swing-Z policies, but use a large downwards acceleration (-25 m/s/s) in the z direction. Once the foot reaches the ground, we begin a normal double support phase. At the beginning of the lowered foot's next step, it uses the raising strategy to clear the obstacle.

Based on the results shown in Figure 4.13, the lowering strategy works better towards the end of a step, and the raising strategy works better earlier in the step. We should therefore use the lowering strategy if an obstacle is detected with $t_{td} < 0.15\mathrm{s}$ and use the raising strategy otherwise. This is qualitatively consistent with strategies observed in human walking [42] [62].

4.4.4 Steps Up/Down

We tested the system's response to unexpected changes in ground height to find the largest unexpected step up and step down it could handle. In both cases (up and down), we found that the largest step it could handle was about 7 cm (though for stepping up, it depended somewhat on when in the swing phase it encountered the change in ground height).

When stepping down, the controller first becomes aware of the step when it does not make contact with the ground when it expects to. At this point, it moves the swing foot rapidly downward (it attempts to servo to 1.0 m/s downward velocity) until it hits the ground. It must find the ground quickly because otherwise the CoM will move too far ahead of the foot, and it will fall forwards. Once the foot reaches the ground the normal controller resumes. The other foot assumes its starting position is the ground height (for purposes of the Swing-Z policy) until it passes the stance foot at the lower elevation, after which it switches to using the new ground height.

When stepping up, because we wish to handle steps that are larger than our normal ground clearance, we use the same raising strategy as was used for the tripping obstacles described in Section 4.4.3. Double support begins when it unexpectedly lands on the ground on the far side of the apparent obstacle. When stepping up, the swing foot immediately uses the higher ground elevation for its Swing-Z policy because too much ground clearance is safer than too little.

4.4.5 Slopes

The steepest constant slope that our simulation controller could walk up for a large number of steps had a rise of 7.5 cm per horizontal meter (4.3 degrees from horizontal). We tested slopes in simulation only changing the height of the ground; we did not change the surface normal. This makes it a little bit like walking up steps, but without having to worry about where you put your feet. Changing the surface normal is problematic because our simulation is implemented without true rigid-body feet. It has point feet, and the ankle joints apply torque directly to the ground during stance. For walking on flat ground, this is nearly dynamically identical to having true rigid-body feet, but it makes changing the foot-ground interaction difficult.

Changing the surface normal should only affect the friction cone constraints, which are rarely active during normal walking. The ZMP constraints, which play a much bigger role in walking, are barely affected by non-horizontal feet. The only effect is that the projection of the feet into the horizontal plane is slightly smaller. On a real robot, non-vertical surface normals are more of a problem for state estimation than from a dynamics perspective, but we use perfect state knowledge in our simulation.

4.5 Upper Body Rotation

The walking controller described above always attempts to maintain the torso in a vertical posture. Our policies are generated under the assumption that there will be no change in system angular momentum, $\dot{\mathbf{L}} = 0$. Most of the angular momentum in the system is in the torso, so we approximately obey this constraint by maintaining the torso at a fixed orientation. However, angular momentum and torso rotation can be used to aid in balance.

We have identified a mode of motion that rotates the upper body and translates the CoM without moving the CoP. We have also identified a point, which we refer to as the augmented CoM, which is unaffected by these motions, but follows LIPM-like dynamics. By performing a simple change of variables, we can use this point (instead of the CoM) with any LIPM-based control algorithm (including the DP policies described above). The key is that the identified mode functions as an additional source of control authority that looks mathematically similar to moving the CoP. Our DP policies can use this rotational mode of control the same way that they use variations of the CoP without any need for them to understand the difference. We generate versions of both the sagittal and coronal policies that replace the old ZMP constraint with a looser constraint that allows for the added effect of both rotating the torso and moving the ZMP. At run time, a lower level controller determines how much of the command to achieve by actually moving the CoP and how much to achieve by rotating the torso. In addition to giving us greater control authority, this modification allows us to predict the result of undesired torso rotations on CoM motion and compensate for it.

We found that this modification greatly improves robustness to pushes when walking with fixed footstep locations, but actually reduces robustness when walking normally, where the controller is free to choose the footstep locations. During normal walking, most of the robustness to pushes comes from the ability to greatly modify the footstep location and take large steps in the direction the system was pushed. Balancing by torso rotation involves rotating the upper body so that the head moves in the direction of the push and the hip moves away, which prevents the system from taking a large step in that direction. The gains from rotating the torso are smaller than the gains from stepping farther in that direction, so our torso rotation strategy results in a net loss of robustness for normal walking. This result is not entirely surprising given human experience: humans will lean their bodies significantly when trying to walk on a balance beam or on stepping stones, but if pushed when walking on flat ground will keep their torso mostly upright and recover by stepping in the direction of the push.

4.5.1 System Model

We wish to modify the LIPM dynamics (4.4) and (4.5) to avoid the $\dot{\mathbf{L}} = 0$ constraint and model upper body rotation. It is often assumed that the majority of the mass is in the torso, and that we can approximately enforce $\dot{\mathbf{L}} = 0$ by maintaining a fixed torso orientation. However, it may not be possible to perfectly control the torso orientation. Undesired torso rotations may be caused by error in the model of the robot or the environment as well as by external pushes. These rotations can be particularly large in torque-controlled systems or systems with relatively low position gains. Any angular acceleration of the torso (or more generally, any change in angular momentum) without changing the COM acceleration will require some foot torque, which means some change in the COP. Conversely, if the COP of the robot is matched to that of a LIPM model, but there is a change in angular momentum, then the COM motion of the robot will not match that predicted by the LIPM model.

In order to model the interaction between upper body rotation and COM motion, we model the upper body as a single rigid flywheel, which rotates about the system center of mass. This model is known as the Linear Inverted Pendulum plus Flywheel Model (LIPFM) [80] with dynamics given by

$$\ddot{x} = \frac{g}{h}(x - z) - \frac{I\ddot{\theta}}{mh} \tag{4.47}$$

where I is the moment of inertia of the upper body about the system COM and θ is the angle of the upper body relative to the nominal. The upper body angle will also be subject to the kinematic constraint $\theta_{\min} \leq \theta \leq \theta_{\max}$. The LIPFM does not fully account for angular momentum, but many humanoid systems have a large fraction of their mass in the upper body (about 2/3 of the total mass in our system), so a large fraction of the angular momentum will be in the upper body. Additionally, while the lower body motion is highly constrained by walking, we are free to rotate the upper body as desired to aid in control.

We now perform a change of variables so that upper body rotation does not require any change in the COP. Additionally, it will put the LIPFM dynamics in the same form as the LIPM dynamics, allowing us to leverage the existing technology for controlling the LIPM.

Change of Variables

We start with the forward (rotational and translational) dynamics for the x-z plane,

$$\begin{bmatrix} m\ddot{x} \\ I\ddot{\theta} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -h & mg \end{bmatrix} \begin{bmatrix} F_x \\ -z \end{bmatrix} + mg \begin{bmatrix} 0 \\ x \end{bmatrix}, \tag{4.48}$$

where F_x is the horizontal ground reaction force. We then multiply by a change of variables matrix,

$$\mathbf{D} \begin{bmatrix} m\ddot{x} \\ I\ddot{\theta} \end{bmatrix} = \mathbf{D} \begin{bmatrix} 1 & 0 \\ -h & mg \end{bmatrix} \begin{bmatrix} F_x \\ -z \end{bmatrix} + \mathbf{D}mg \begin{bmatrix} 0 \\ x \end{bmatrix}. \tag{4.49}$$

where

$$\mathbf{D} = \begin{bmatrix} 1 & \frac{1}{h} \\ 0 & 1 \end{bmatrix}. \tag{4.50}$$

We now define the augmented COM acceleration, $\ddot{\tilde{x}} \equiv \ddot{x} + \frac{I\ddot{\theta}}{mh}$, then combine and simplify to get

$$\begin{bmatrix} m\ddot{\tilde{x}} \\ I\ddot{\theta} \end{bmatrix} = \begin{bmatrix} 0 & \frac{mg}{h} \\ -h & mg \end{bmatrix} \begin{bmatrix} F_x \\ -z \end{bmatrix} + \begin{bmatrix} \frac{mg}{h}x \\ mgx \end{bmatrix}. \tag{4.51}$$

Double integrating $\ddot{\tilde{x}}$ gives us the augmented COM position, $\tilde{x} = x + \frac{I\theta}{mh}$, which we substitute into (4.51) to get

$$\begin{bmatrix} m\ddot{\tilde{x}} \\ I\ddot{\theta} \end{bmatrix} = \begin{bmatrix} 0 & \frac{mg}{h} \\ -h & mg \end{bmatrix} \begin{bmatrix} F_x \\ -z \end{bmatrix} + \begin{bmatrix} \frac{mg}{h}\tilde{x} + \frac{gI\theta}{h^2} \\ mg\tilde{x} + \frac{gI\theta}{h} \end{bmatrix}. \tag{4.52}$$

Multiplying by the inverse of the 2x2 matrix and solving for the controls, F_x and z, gives the inverse dynamics,

$$\begin{bmatrix} F_x \\ -z \end{bmatrix} = \begin{bmatrix} 1 & \frac{-1}{h} \\ \frac{h}{mg} & 0 \end{bmatrix} \begin{bmatrix} m\ddot{\tilde{x}} \\ I\ddot{\theta} \end{bmatrix} + \begin{bmatrix} 0 \\ \tilde{x} + \frac{I\theta}{mh} \end{bmatrix}. \tag{4.53}$$

The benefit of the change of variables is that we now have a 0 in the lower right of the 2x2 matrix. This 0 means that changes in angular momentum without modifying $\ddot{\tilde{x}}$ have no effect on the COP.

If we define an augmented COP,

$$\tilde{z} = z + \frac{I\theta}{mh}, \tag{4.54}$$

we are able to put the LIPFM dynamics,

$$\ddot{\tilde{x}} = \frac{g}{h}(\tilde{x} - \tilde{z}), \tag{4.55}$$

in the same form as the original LIPM dynamics in (4.4).

Since (4.4) and (4.55) have the same form, control methods that work on the LIPM dynamics can also be used on the augmented LIPM dynamics, but by working in the space of $\tilde{x} = x + I\theta/mh$ and its derivatives instead of the space of x and its derivatives. Additionally, controllers will now be requesting a $\tilde{z}$, and (4.54) must be used to find z before applying control to the system. The change of variables eliminated a $\ddot{\theta}$ term (by hiding it within $\ddot{\tilde{x}}$), but created a θ term. The angular position term is preferable from a modeling perspective because it changes more slowly and can be more easily measured.

Orientation Control

Since θ can be controlled independently of $\tilde{x}$, the θ term in $\tilde{z}$ can be used for more than canceling the effect of small undesired deviations from the desired posture. It can also be actively controlled to something other than 0 and treated as an additional control for the augmented COM. Torso orientation, θ, has the same effect on the system as does z, so a LIPM controller need not differentiate between them; instead, it can simply request a $\tilde{z}$. In addition, the controller can operate under a more lenient constraint (but with the same form) than the ordinary COP constraint, $|z| \leq z_{\max}$. For the LIPFM dynamics, the constraint is

$$|\tilde{z}| \leq z_{\max} + \frac{I_{\text{eff}}\theta_{\max}}{mh} \tag{4.56}$$

where $\theta_{\max}$ is the maximum allowable lean angle. Use of the more lenient constraint (and the torso rotation necessary to produce the larger requested $\tilde{z}$) makes it possible to plan more aggressive maneuvers that are impossible with a fixed-orientation torso. It also allows for improved feedback in response to unexpected disturbances.

There is also an additional constraint arising from the fact that while contact forces such as z can be changed quickly, torso angle will have a maximum velocity or acceleration. Such constraints become an issue if the requested $\tilde{z}$ rapidly changes. In situations where $\tilde{z}$ changes gradually or on systems where the internal dynamics (dependent on actuators) are much faster than the LIPM dynamics (dependent on g/h), this additional constraint can reasonably be ignored.

There are several options for controlling θ depending on the context. If the increased control authority is being used in a planning context for generating trajectories with large accelerations, then the planner will produce a time varying $\tilde{z}(t)$. In this case, optimization (for example, a quadratic program) can be used to determine a time varying $\theta(t)$ which satisfies (4.56) as well as whatever hardware constraints exist on $\dot{\theta}$ or $\ddot{\theta}$. A cost on $\theta(t)$ can be added to the optimization to bias it towards upright postures that do not utilize torso rotation wherever possible. However, if (as we use it) the increased control authority is being used for improved responses to unexpected disturbances, then no expected $\tilde{z}(t)$ will be available (because the disturbances are unexpected).

In this case, we use a fixed relationship between $\tilde{z}$ and the desired torso orientation, θ_{des}, then use a high gain PD servo to track that desired angle. Immediately following a large disturbance, it will not be possible to obtain the requested $\tilde{z}$ until the torso has had time to slew, resulting in (4.55) being inaccurate. However, slewing happens relatively quickly, and rotating the torso at the maximum speed is the best thing to do in this situation. Therefore, the quality of our control does not suffer from neglecting this effect.

We have used a piecewise linear relationship with a dead zone to determine θ_{des} from $\tilde{z}$.

$$\theta_{des}(\tilde{z}) = \begin{cases} \text{if } \tilde{z} > z_{\max}/2 & : \quad (\tilde{z} - z_{\max}/2)mh/I_{\text{eff}} \\ \text{if } \tilde{z} < -z_{\max}/2 & : \quad (\tilde{z} + z_{\max}/2)mh/I_{\text{eff}} \\ \text{else} & : \quad 0 \end{cases} \tag{4.57}$$

The purpose of the dead zone is to prevent the upper body from rotating unnecessarily, which would both look unnatural and run the risk of exciting unmodeled dynamic modes. For ordinary small values of $\tilde{z}$, θ_{des} will remain at 0, and torso rotation will only be utilized when it becomes necessary to achieve large $\tilde{z}$'s. In simulation, the exact shape of this relationship has very little effect on the system robustness. However, smoother functions perform better on real hardware due to decreased jerk and less excitation of unmodeled higher-order dynamics.

4.5.2 Use in Practice

We use the method described here to add torso rotation to our controller without having to add dimensions to or otherwise complicate our DP policies. Many other existing systems that use the LIPM dynamics could use a similar modification. We use it for feedback control where we have a system position and velocity and wish to generate desired accelerations and torques. However, it can also be used with pattern generation methods based on the LIPM dynamics. We first explain how it could be applied to open-loop pattern generation and then torque feedback control.

Pattern Generation

Figure 4.15 shows how you would use this method for augmenting a simple generic motion planning system for a position controlled robot. The upper flow chart depicts the unmodified

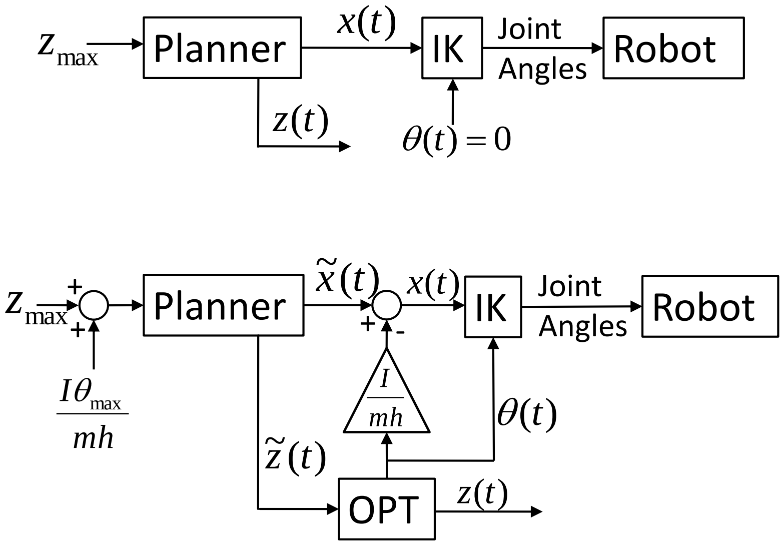

Figure 4.15: A flow chart of a simple generic system for humanoid walking pattern generation using standard LIPM dynamics (top) and a flow chart for that same system modified to use the LIPFM dynamics (bottom). IK refers to inverse kinematics and OPT refers to a trajectory optimization.

system, which uses a planner (e.g. preview control) to generate COM and COP trajectories. Then, an inverse kinematics solver uses the COM trajectory and a fixed upper body orientation, $\theta(t) = 0$, to generate joint angle trajectories, which are used to drive the robot.

The lower flow chart shows how this system would be modified to use the LIPFM dynamics as described in this Section. The same planner and inverse kinematics solver can be used without modification, but the interaction between the two is modified. First, the planner is given the more lenient constraint given by (4.56). Now, the output of the planner is interpreted as $\tilde{x}(t)$ and $\tilde{z}(t)$. A simple trajectory optimization (denoted "OPT" in Figure 4.15) is then used to break $\tilde{z}(t)$ into $z(t)$ and $\theta(t)$ such that (4.56) and any hardware constraints on θ or its derivatives are satisfied at all times. The $\theta(t)$ trajectory is then used in place of the fixed torso orientation given to the inverse kinematics solver. It is also used to compute $x(t)$ from $\tilde{x}(t)$. The inverse kinematics solver then determines joint angle trajectories for driving the robot.

The change of variables allows the planner to look at the extra capability given by upper body rotation as simply a larger virtual foot. Then, we post-process its output to achieve the extra control authority it requests through upper body rotation.

Torque Control

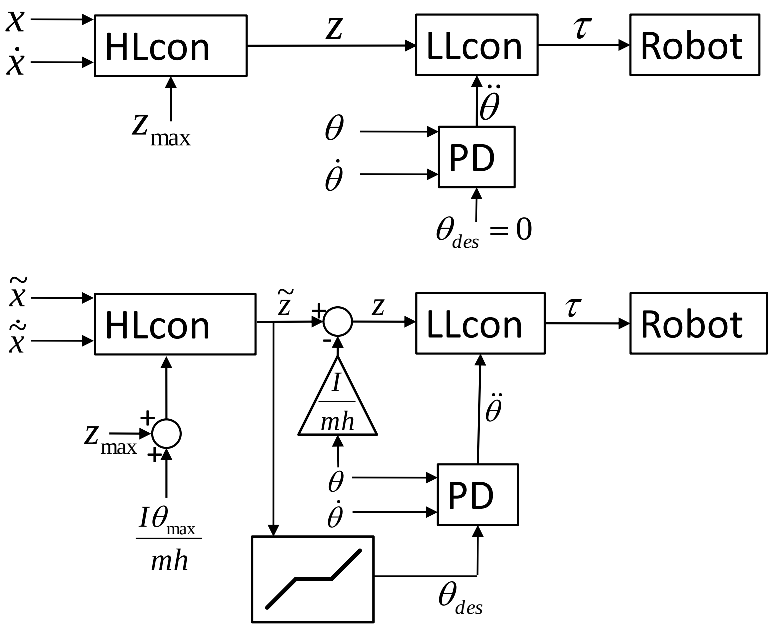

Figure 4.16: A flow chart of a simple generic system for torque control of a humanoid robot using standard LIPM dynamics (top) and a flow chart for that same system modified to use the LIPFM dynamics (bottom). LLcon and HLcon refer to low-level controller and high-level controller respectively.

We use an analogous process to modify a LIPM-based controller for a torque controlled robot, as shown in Figure 4.16. The upper flow chart depicts a generic LIPM-based controller. A high-level controller (coordinated DP policies in our case), denoted “HLcon” in Figure 4.16, uses the current COM state, x and $\dot{x}$ to determine the desired COP, z. A PD servo is used to keep the torso angle at a fixed orientation, and a low-level controller (DBFC or WOID in our case), denoted “LLcon” in Figure 4.16, generates the joint torques necessary to achieve the desired z and $\ddot{\theta}$.

The lower flow chart shows how this system was modified to use the LIPFM dynamics. Again, the primary algorithms, the high-level and low-level controllers, remain unchanged. As for pattern generation, we modify the COP constraint according to (4.56). The inputs to the high level controller are now the augmented COM state, $\tilde{x}$ and $\dot{\tilde{x}}$, and the output is now interpreted as $\tilde{z}$. We use θ and $\tilde{z}$ to determine z by rearranging (4.54), which is passed to the low-level controller. The bottom box represents the function given in (4.57), and is used to pick a θ_{des} that makes $\tilde{z}$ achievable.

4.5.3 Results

Figure 4.17 shows results for a robustness to pushes experiment similar to the one depicted in Figure 4.12 but with the torso rotation control modification described here and fixed footstep locations. We used sagittal and coronal policies computed with the old ZMP constraint replaced by the weaker constraint given by (4.56). The system shows considerably less robustness here than in Figure 4.12 because it is walking with fixed footstep locations. The fact that the recoverable regions are nearly rectangular can be interpreted as indicating that the sagittal and coronal systems really are mostly decoupled. The improvement is larger in the coronal direction because we start to run into kinematic constraints in the sagittal direction even when walking with fixed footstep locations; the hip moves too far for away from the desired touchdown location and the swing foot cannot reach it.

4.6 Walking Faster by Predicting Angular Momentum

The controller described in Section 4.2 works well for walking speeds up to about 0.6 m/s, about half the speed of human walking. However, at higher walking speeds, the system tends

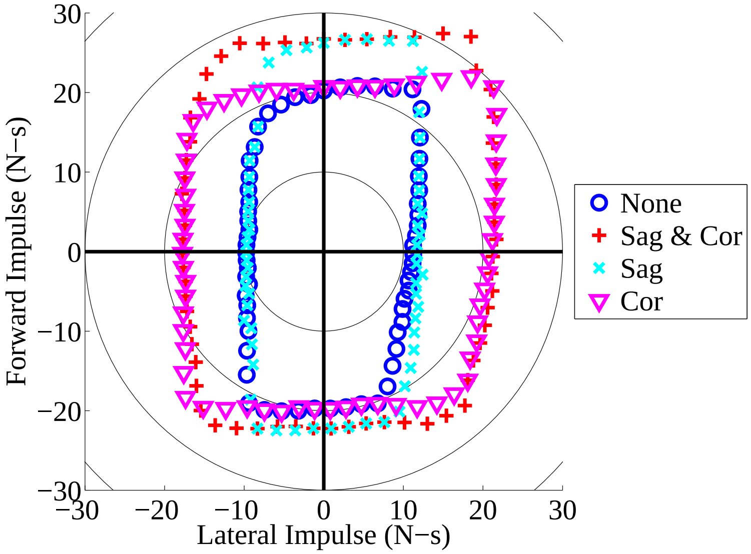

Figure 4.17: Polar plot of the maximum survivable perturbation of our walking simulation as a function of push angle. A point represents the maximum survivable perturbation in a given direction. Pushes occur midway through right single support. Concentric circles are in increments of 10 Newton-seconds. Data is shown for the unmodified system, the system modified in both the sagittal and coronal planes, only the sagittal, and only the coronal plane.

to develop a limp consisting of alternating long and short steps which rapidly grows worse and results in failure within several steps. This failure mode shows up at faster walking speeds because we generated our sagittal plane control policy under the assumption of zero change in angular momentum, but at high walking speeds there is enough change in angular momentum to cause large deviations between the expected and actual forward center of mass acceleration. If we rewrite (4.20) in a different form and do not remove the angular momentum term, we can write the forward center of mass dynamics as

$$\ddot{x} = \frac{\dot{\mathbf{L}}_y/m + (g + \frac{\partial^2 h}{\partial x^2}\dot{x})(x - p_x)}{h(x,d) - \frac{\partial h}{\partial x}(x - p_x)}, \tag{4.58}$$

where p_x is location of the center of pressure in the x direction and $\dot{\mathbf{L}}_y$ is the time derivative of the y component of angular momentum. Note that if we enforce $\dot{\mathbf{L}}_y = 0$ and $h(x, d) = \text{constant}$, this simplifies to the LIPM dynamics. When generating our policy, we did enforce the former condition, but selected a center of mass height function based on the compass gait, but modified to have double support.

Based on this equation, we can see that there are only three possible sources of deviation between the expected and achieved center of mass acceleration: center of pressure, vertical center of mass motion, and change in angular momentum. In simulation, where we have a perfect model and actuators that are pure torque sources, we achieve the desired center of pressure nearly perfectly so CoP error contributes negligibly to the CoM acceleration error. Figure 4.18 shows the effect of the remaining two possible factors.

There is significant deviation between the expected and measured CoM height, but the effect on the forward CoM acceleration is very small. On the other hand, almost the entirety of the error can be explained by unexpected changes in angular momentum. The effect is so pronounced that we see the opposite of the expected pattern of acceleration during single support; rather than decelerating during the first half of single support and accelerating during the second half, we end up accelerating and then decelerating the CoM.

It is also worth noting how comparatively unimportant the effect of vertical motion is. If we had used the same controller, but generated our policy using LIPM dynamics ($\ddot{z} = 0$), the effect would have been approximately 4 times larger, but it still would have been small compared to the effect of angular momentum. This implies that it is reasonable to plan horizontal CoM motion using LIPM dynamics, but actually control the robot so that the vertical CoM motion is more natural than the constant height required by the LIPM dynamics. The effect of this discrepancy on the horizontal motion (or ZMP location for position-controlled schemes) should be much smaller than that of angular momentum.

It is somewhat surprising that the effect of angular momentum is so large given that studies

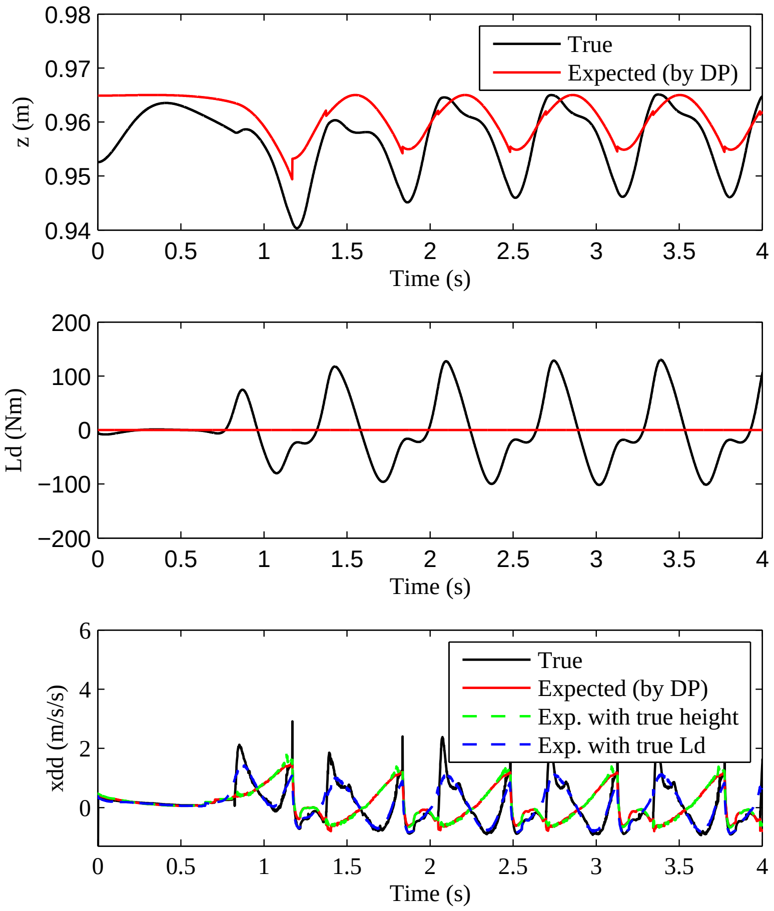

Figure 4.18: The top plot shows expected and actual center of mass height. The middle plot shows expected ($\dot{\mathbf{L}} = 0$) and actual time derivative of angular momentum (y-component). The bottom plot shows the effect of these two errors on the center of mass acceleration. The true and predicted values are shown. Additionally, a trace calculated using the true vertical CoM motion but zero angular momentum change is shown as well as a trace calculated using the expected vertical CoM motion, but the true angular momentum change. Data was taken during simulated walking at $0.45\mathrm{m/s}$.

have shown whole body angular momentum to be relatively small in human walking [35]. We believe there to be several factors contributing to this difference. For simplicity, our simulation has no arms, so we have not incorporated arm swing into the gait. While the effect of arm swing on angular momentum is primarily about the yaw axis, it also has a significant effect in the sagittal plane. Additionally, the kinematics of our gait are not identical to that of human walking. We do not use pushoff and the swing leg path as well as timing may differ significantly from that used by humans. While the above two factors can be regarded as features of our controller, the mass distribution of our simulation (and the robot it is based on) differ from that of a human in such a way as to result in more angular momentum during walking.

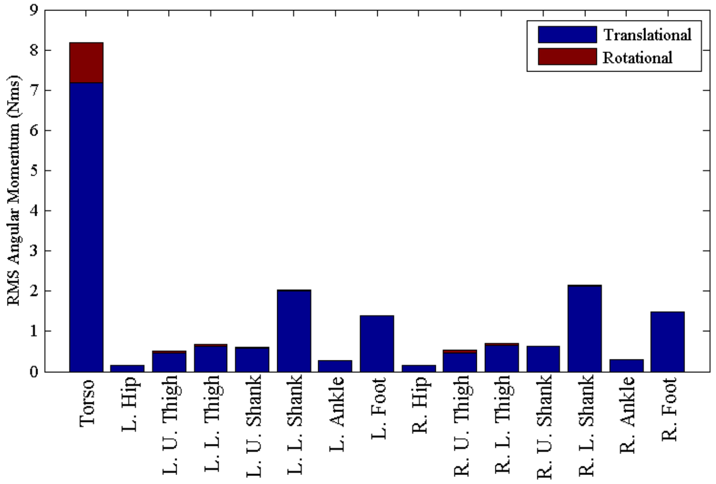

Figure 4.19: RMS angular momentum in each rigid body link during walking at $0.45\mathrm{m/s}$. Translational angular momentum refers to the contribution from the body's mass moving relative to the system CoM, and rotational angular momentum refers to the contribution from the individual body rotating in space.

Figure 4.19 shows the RMS angular momentum in each link of the rigid body system during walking and further differentiates between translational (motion of the body about the system CoM) and rotational (rotation of the body) components. Note that this figure only shows which bodies have large amounts of angular momentum (relative to the system CoM), but does not say anything about how concurrent motion of multiple bodies cancel out to produce less total system angular momentum. In the limit where we describe the robot by an infinite number of infinitesimal particles, all of the angular momentum would be “translational” rather than “rotational”. We subdivide the legs into a large number of bodies, so it is unsurprising that the translational component dominates. Other than the torso, the largest contributions to angular momentum are the foot and lower leg, which move quickly at a large distance from the system CoM. Our system has about the same fraction of its mass in the legs as a human does. However, it has more of this mass below the knees in the lower legs than an average human. This extra mass in precisely the parts of the body that are moving quickest and farthest from the system CoM result in larger system angular momentum. While not universally true, it is common for humanoid robots, especially high degree-of-freedom ones, to have a larger fraction of their mass in their lower legs than humans, meaning that this phenomenon may be common in humanoid robotic walking.

It is possible for our controller to walk with very little angular momentum. Change in angular momentum is equal to the torque about the CoM, so we have complete control of the angular momentum when selecting our contact forces. We normally select contact forces that provide torque sufficient to keep the torso vertical during walking, but if we drastically reduce the PD gains on torso angle, our controller will walk with nearly constant angular momentum comparable to human walking. The effect of this is that the torso flops back and forth to cancel out the angular momentum in the legs and maintain a total system angular momentum near zero. Figure 4.20 compares this to regular walking.

In addition to being aesthetically distasteful, the aggressive torso rotation results in kinematic difficulties making low torso gains an inadequate solution for walking at faster speeds. As the torso rotates, the hip moves forward and backward relative to the system CoM. Combined with the longer step lengths involved in faster walking, this results in violating reachability constraints. As the hip deviates horizontally from the CoM, it can make it impossible for the feet to reach their desired positions on the ground.

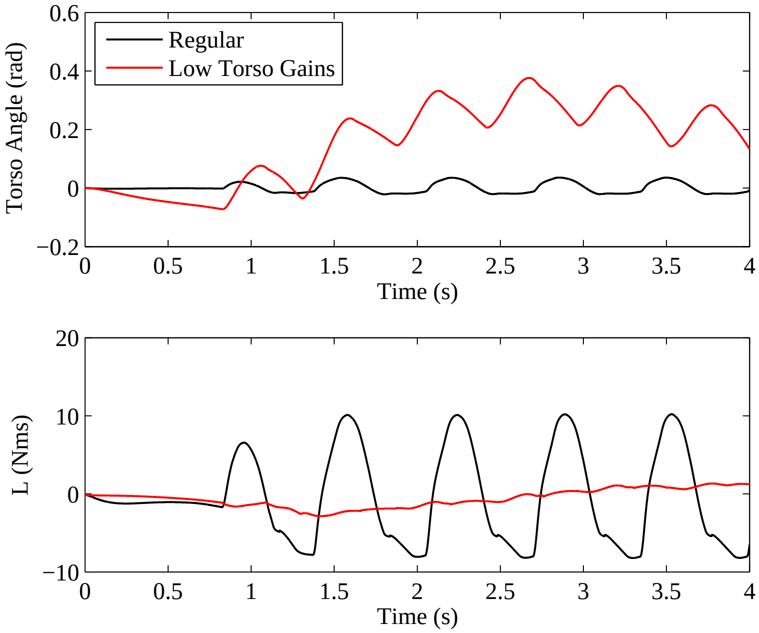

Figure 4.20: Comparison of walking normally and walking with very low torso angle gains. With low torso gains, the angular momentum remains nearly constant, but the torso rotates aggressively to counter the angular momentum in the legs.

4.6.1 Predicting Change in Angular Momentum

To walk at faster speeds, we will need to be able to predict the change in angular momentum $\dot{\mathbf{L}}_y$. At run time, we can use the full forward dynamics to compute $\dot{\mathbf{L}}_y$, but unfortunately, we need it during policy generation, when we only have access to the reduced state used by the DP policy. The simple model used for generating the DP policy has no notion of angular momentum so it can not be used for this purpose. Instead, we learn a function for $\dot{\mathbf{L}}_y$ as a function of the DP states and actions using data generated by our existing walking controller. For this, we use Gaussian Processes (GP) because they handle the inconsistent data produced as a result of using a reduced state well. Additionally, there exists easy to use off-the-shelf software available with automatic hyperparameter selection [85]. However, many other types of function approximator would also work well.

To generate rich data, we modify our walking controller in various ways. We can easily adjust the walking speed by adding an offset to the desired CoP location (adjust backward to walk faster and adjust forward to slow down). We can also modify the walking pattern by adding additional terms to the cost function during the coordination process when footstep timing is determined; we can use this method to increase or decrease the duration of single support or double support. Finally, we can explore more of the state space by perturbing the system with an external push and taking data during the recovery (though not during the push itself). To maintain computational efficiency during the iterative policy generation procedure, we store the function approximated by the GP in a grid similar to that used by the DP algorithm rather than computing the output of the GP repeatedly. Since the states and actions used by DP are different in each, we must handle the data from single support and double support separately.

During double support, we have access to the five DP state variables: x, CoM position, $\dot{x}$, CoM velocity, t_{lo}, time until liftoff, d, the stance width, and w, the fraction of weight on the lead leg as well as the two DP control variables, p, the CoP, and $\dot{w}$. To keep the grid a manageable size (for both time and memory limitations), we want to find a function for $\dot{\mathbf{L}}_y$ of no more than five variables, so we must drop two of the available seven. We choose to ignore $\dot{w}$ because its effect is higher order than the other choices, affecting jerk rather than acceleration. We consider t_{lo}, d, and w as candidates for the 2nd variable to drop. For each of these there candidates, we fit a function for $\dot{\mathbf{L}}_y$ using the remaining five variables, then measure the RMS error when predicting $\dot{\mathbf{L}}_y$ on three new walking trials, one at a new walking speed, one response to a new push, and one with a new alteration to the step timing. The results, shown in Table 4.6.1, indicate that it is best to not include t_{lo} in the fit because it results in the least error, actually improving the prediction on two of the three trials.

During single support, there are only five variables available: the four DP state variables x, $\dot{x}$, t_{td}, time to touchdown, and x_{td}, the location of the impending touchdown as well as the only DP action, p. It is possible to create a grid representing a fit for $\dot{\mathbf{L}}_y$ using all five of these

Table 4.2: RMS error on the double support portion of three test trials when fitting $\dot{\mathbf{L}}_y$ using all six of x, $\dot{x}$, t_{lo}, d, w, and p, as well as the five remaining variables when removing t_{lo}, d, or w. The test data was produced by varying the speed by modifying the CoP, providing an external push, and using additional cost function terms to vary the step timing.

	Speed Test	Push Test	Timing Test
All Variables	10.4	12.5	16.6
Drop t_{lo}	10.5	12.3	13.9
Drop d	12.3	11.3	23.5
Drop w	9.5	14.0	22.9

variables. However, when using this estimate, the system fails to walk. The problem appears to be that the controller believes that t_{td} and x_{td} have a direct effect on $\dot{\mathbf{L}}_y$ and plans accordingly. However, t_{td} and x_{td} are really just internal controller states and will have no direct effect on the system dynamics. To prevent the policy generation and coordination algorithms from attempting to exploit loopholes that do not actually exist, we remove both t_{td} and x_{td} from the fit, learning a function for $\dot{\mathbf{L}}_y$ based on only x, $\dot{x}$, and p. To quantify the effect of fitting $\dot{\mathbf{L}}_y$ from fewer variables, we develop fits with and without them and measure the RMS error. Table 4.6.1 shows results for experiments similar to Table 4.6.1, but using the single support portion of the data rather than the double support portion. Unfortunately, fitting with only the remaining three variables results in significantly more prediction error, but still much less than the magnitude of $\dot{\mathbf{L}}_y$. The controller performs well with this estimate of $\dot{\mathbf{L}}_y$ plugged into the dynamics given by (4.58) during both policy generation and execution, providing a better prediction of $\ddot{x}$, the CoM acceleration.

4.6.2 Bootstrapping to Higher Speeds

With an estimate of $\dot{\mathbf{L}}_y$, we can generate sagittal walking policies based on simple model dynamics that more accurately reflect the dynamics of the true system, which allows us to successfully

Table 4.3: RMS error on the single support portion of three test trials when fitting $\dot{\mathbf{L}}_y$ using all five of x, $\dot{x}$, t_{td}, x_{td}, and p, as well as the remaining variables when removing t_{td}, x_{td}, both, or both and p. The test data was produced by varying the speed by modifying the CoP, providing an external push, and using additional cost function terms to vary the step timing.

	Speed Test	Push Test	Timing Test
All Variables	14.8	17.7	24.5
Drop t_{td}	18.2	21.5	25.4
Drop x_{td}	15.0	17.9	23.3
Drop t_{td} and x_{td}	32.5	32.9	28.4
Drop t_{td} and x_{td} and p	37.3	40.8	28.8

generate policies for faster walking. Unfortunately, our estimate of $\dot{\mathbf{L}}_y$ is only good in the region of state space where we were able to provide the GP with data, so we can only use it for generating policies for walking a little bit (0.1-0.2 m/s) faster than our existing policies. However, we can use this new policy to get new walking data at higher speeds. We can then iterate in this manner, using our estimate of $\dot{\mathbf{L}}_y$ to generate policies for progressively faster walking while using those policies to get data for generating an estimate of $\dot{\mathbf{L}}_y$ that is valid at progressively higher velocities. After four such iterations, we were able to generate a policy that has a desired velocity of $v_{\text{des}} = 1.0\text{m/s}$. Using this policy, the system's actual walking rate was 0.91 m/s, but by adding an offset to the CoP, it could be increased to 1.05 m/s before becoming unstable. It is likely that similar results could have been achieved with fewer iterations if we had used a function approximator for $\dot{\mathbf{L}}_y$ that was better at extrapolation than GP's.

At this point, we appear to have reached a point of diminishing returns where it becomes more difficult to further increase the walking speed. This is still slightly slower than human walking (1.2-1.3 m/s), but it is entirely flat-footed walking with no toe-off. Toe-off allows the rear foot to remain on the ground longer as the hip moves forward, which allows for longer steps while keeping the CoP behind the CoM. In flat-footed walking, we have no way to further extend the rear leg once the knee has become straight so we are forced to lift the foot earlier. While our controller framework can handle toe-off (and heel-strike), implementation in simulation would require a significant upgrade of the simulator itself, which is a promising avenue of future work.

4.7 Conclusion

By avoiding pre-planned trajectories, our controller can react immediately to unexpected perturbations. It selects a new time and location of touchdown at every control cycle (currently 400 Hz). In the sense that it avoids tracking a planned trajectory, our method resembles MPC. However, to compute the control in real time, MPC approaches typically use linear dynamics and fixes the step timing. We, on the other hand, avoid any requirement of linearity by performing our optimization iteratively offline. However, offline computation forces us to consider all possible states, which subjects us to the "Curse of Dimensionality" and constrains the dimensionality of our dynamics. Our coordination scheme does let us handle higher-dimensional systems by breaking them into lower-dimensional subsystems, but it requires that the full system be an ICS. This, in turn, requires that both the dynamics and cost function be capable of being decoupled (except at transitions).

One major advantage of this control framework is its flexibility: Coordination is not dependent on any particular dynamic model, cost function, or constraints. While we must take care to maintain low dimensionality and the ability to decouple our system, we remain free from any further restrictions - for instance, linearity - on the dynamics or cost function. In fact, many of the motion's characteristics can be controlled by adjusting the cost functions, and the variables considered can be changed by combining or adding additional subsystems.

We have defined an Instantaneously Coupled System and demonstrated the equivalence of coordinated policies that are optimal for the subsystems to a single controller that is optimal for the full ICS. We apply this theory to walking and present a walking controller for a simulated biped. Our controller optimizes center of mass motion as well as footstep timing and location, and it can react in real time to perturbations and accumulated modeling error.

Chapter 5

Robotic Walking

5.1 Introduction

Our simulated walking was accomplished on a simulation based on the Sarcos Primus hydraulic robot, and it was designed with the eventual goal of walking on the real robot in mind. However, the simulation controller does not perform well unaltered on the actual robot. The difference in performance is largely due to an inaccurate dynamic model of the robot, inaccurate sensor calibration, and inaccurate state estimation. In this chapter, we will discuss the robot itself, the changes we made to the simulation controller in response to the difficulties associated with working on real hardware, as well as results for walking in place and walking forward.

5.2 Robot Description

All of our hardware experiments are done with the Sarcos Primus humanoid robot [16, 52] shown in Figure 5.1. It is a hydraulically powered, force controlled humanoid robot massing about 95 kg and about 1.7 m tall. When standing straight, its CoM is about 1.0 m above the ground. Its feet are flat rectangular aluminum plates (about 0.25 cm long and 0.12 cm wide) with 1/4 inch thick rubber glued to the bottom.

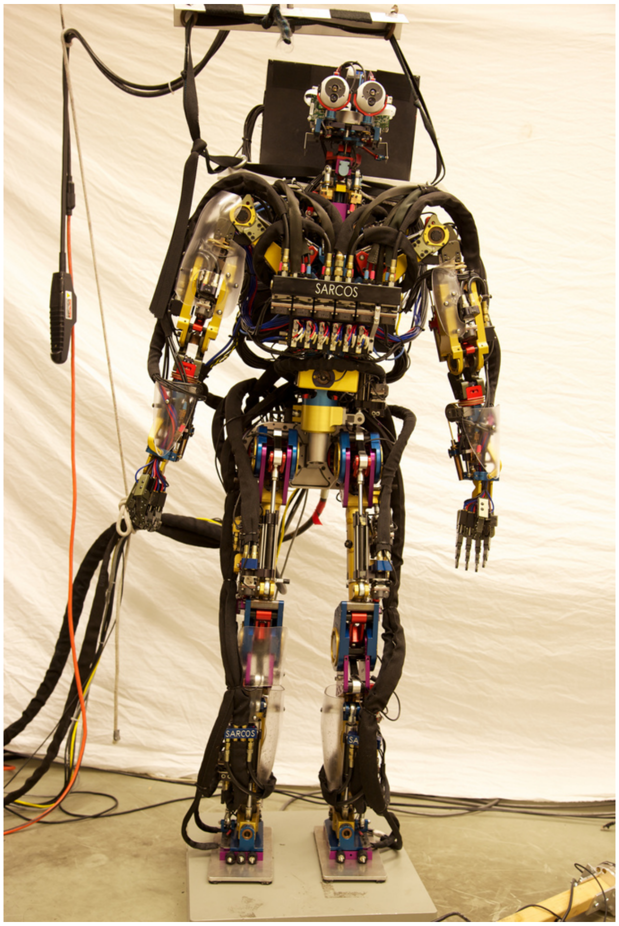

Figure 5.1: The Sarcos Primus humanoid force-controlled robot standing on a force/torque plate.

5.2.1 Mechanical

The primary actuation is a 3000 psi hydraulic system powered by an off-board pump. It has 34 hydraulically actuated joints: 7 in each leg, 7 in each arm, 3 in the torso, and 3 in the neck. We only actively control 12 of those joints for the work in this thesis: 2 at each hip, 1 at each knee, 2 at each ankle, and a shank rotation joint with its axis parallel to the shank. The remaining joints are servoed to fixed angles. Using these joints gives us the ability to position our feet with the full 6 degrees of freedom relative to the torso without having any redundancy. All other joints are servoed with high gains to fixed angles. The robot also has electrically actuated pan and tilt for each eye, and 12 pneumatically actuated joints in the hands, which we do not use in this thesis.

5.2.2 Sensing

Every joint has position sensing by means of an analog potentiometer. We also have force sensing at every joint. For the majority of joints, the force sensing is done by strain gauges glued to a rigid link in the hydraulic drive chain. However, for a few joints, we use commercial load cells to sense force. Each foot has a 6-axis force/torque sensor. We also have an Inertial Measurement Unit (IMU) attached to the pelvis near the right hip. There are cameras in the eyes, but we do not use them in this work.

5.2.3 Computation

Our primary computation is done on an off-board computer with an 8-core Pentium processor. It receives sensor information and transmits joint force commands to the robot at 400 Hz. In the robot's backpack are small micro-computers (one for each joint) that perform the high speed force control on individual joints at 5000 Hz. They also have small position and velocity gains that help to smooth control on the short time scale between new commands from the off-board computer.

5.3 Difficulties

The controller described in Chapter 4 works well in simulation, but fails when run on the robot. The robot has several nonidealities that make control more difficult, and it is useful to understand the nature of these nonidealities when designing a controller.

One of the more obvious differences between our simulation and our robot is that in the simulation, we know the system state perfectly, but on the robot, we must rely on imperfect sensors. We use potentiometers (for joint angles) and an IMU on the pelvis (for orientation in space) to get a fairly accurate measurement of the robot's position. However, to get velocities, we filter and numerically differentiate the position estimates. For CoM velocity, for example, we use a second-order Butterworth filter to filter the CoM position before differentiation. If we use a high enough filter cutoff frequency to sufficiently reduce noise, we end up with a CoM velocity estimate that is approximately 75 ms delayed. A fruitful avenue of future work would be developing a Kalman filter to reduce this delay without adding more noise.

Even at the level of position, error can be problematic for some aspects of the controller. It is difficult to accurately calibrate the potentiometers and many of the joints experience significant (visibly obvious) off-axis flexibility when heavily loaded. Such position errors are most important when looking at the swing foot height and vertical velocity and attempting to determine when touchdown will occur. For this reason, our robot controller uses the desired swing foot height and velocity (Section 5.4.2) when determining the touchdown parameters during policy coordination (Section 4.2.2) rather than the inaccurate measured foot height.

The other large category of nonideality is modeling error. We model the robot as a rigid body system, but have had limited success finding a mass distribution model that accurately matches data and performs well in practice. Boston Dynamics also has trouble creating an accurate rigid body model for their Atlas robot. They find about a 2 cm error in predicting CoM location (personal communication). There are many unmodeled effects such as friction (both stiction and viscous) and hose forces (the hydraulic hoses are quite stiff when pressurized). Additionally, there are higher-order dynamics that can be excited by large jerks. The simulation is a pure second order system, so there is no problem if we instantaneously pick new desired accelerations and have discontinuous torque commands. If the same is done on the robot, however, it will excite the higher order dynamics, resulting in oscillation, so we must take care to make all transitions smoothly.

While the robot is very repeatable on a short time scale, it is not very repeatable on a longer time scale. This means that while it may do nearly the exact same thing on consecutive steps of a single trial, running the same controller another day may have completely different results. In addition to requiring greater robustness to handle the day-to-day variations, it has the specific implication that it is best to make the controller as adaptive as possible. Rather than hand-tuning an offset, it's better to use an integrator to automatically find that offset so that when the system changes, the controller adapts along with it. There are also significant delays in the dynamics, caused in part by state estimation, but also contributed to by the hydraulic valves and by software filters. These software filters are necessary to avoid exciting the higher order dynamics with rapid torque changes.

Another problem with the real hardware is joint torque limits. The Sarcos robots has several joints for which this is particularly problematic. The knee joints are one potential source of trouble. Raising the CoM and therefore keeping the knees straighter limits the torque on the knees, but helps very little with regard to the torque limit because it is actually a limit on the piston force, and the piston leverage changes as the knee angle changes, resulting in an angle dependent torque limit. The straighter the knee, the less leverage the piston has and therefore the lower the torque limit, thus counteracting the reduced torque from maintaining straighter stance knees. However, leaning the torso forward does help because it move the hips and knees rearward relative to the feet and CoM. This puts the knees closer to directly under the CoM, which reduces the knee torque. A forward lean angle of 0.3 radians is enough to make knee failures rare. The other problematic joints are the hip adduct/abduct (roll axis) joints, especially the right one; the asymmetry is caused by the difference in force between pushing and pulling for a hydraulic piston. The right hip adduct/abduct piston has insufficient strength for the robot to stand statically on one leg. The result is a significant, uncontrolled torso roll oscillation during walking as it rolls toward the swing leg on every step. To handle this in future work, it is possible to design a gait that includes an intentional roll oscillation of the torso and include it in the simple dynamics through use of the LIPFM (described in Section 4.5.1).

5.4 Major Controller Changes

We can use the same controller for standing both in simulation and on the real robot. Unfortunately, for more complex activities, such as walking, the nonidealities of the real hardware become more important. While the walking controller described in Section 4.2 was designed with an eye towards coping with an inaccurate dynamics model, there are several nonidealities on the real robot (Section 5.3) that make control difficult, some of which need to be specifically addressed.

5.4.1 Jerk-Based Policies

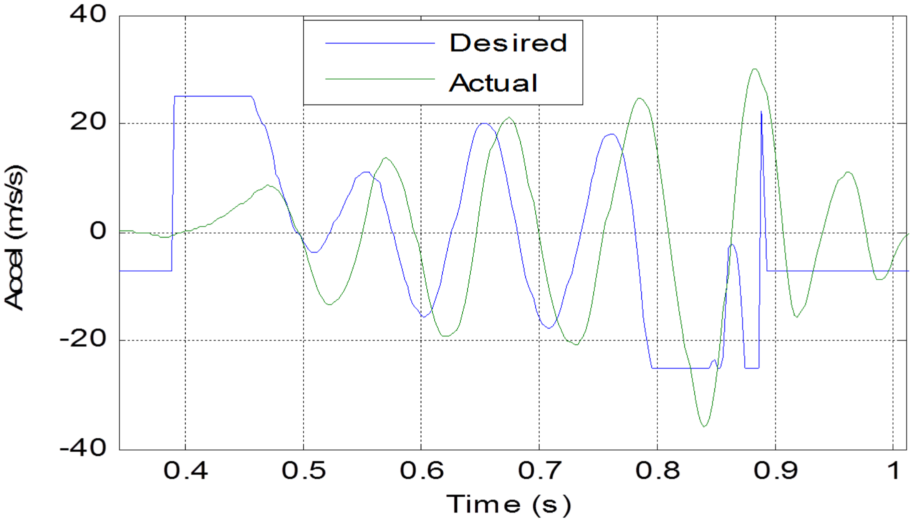

Figure 5.2: Plot of the desired and measured swing foot acceleration in the z direction for one step. Measured accelerations are obtained by filtering and double integrating potentiometer data.

One of the major difficulties with controlling walking on the Sarcos humanoid is control of the swing foot. If we have the robot walk in place using the swing leg policies described in Section 4.2.1, we get oscillation. Figure 5.2 shows the measured and desired acceleration for the swing foot in the z direction. There is a slow initial rise and a large amplitude oscillation at about

9 Hz. Additionally, there is a delay of about 23 ms from the desired to the measured acceleration.

The delay has at least three component causes. There is a slight delay between desired accelerations and desired torques because of a filter on the output torques. The purpose of this filter is to avoid exciting higher order dynamics of the mechanical system. We can increase the cutoff frequency of this filter, but avoiding abrupt changes in desired torque is important. We also have a delay of about 9 ms between desired torques and measured torques caused by the physical characteristics of the hydraulic valves and our low level force control. Finally, we have a significant delay between actual system acceleration and measured acceleration due to state estimation.

Unmodeled delays can cause oscillations, and simple simulations have shown that the oscillations we see in Figure 5.2 are consistent (in frequency and amplitude) with what would be caused by a 23 ms delay. However, changes to the cost function used for policy generation which mitigated the impact of delays in simulation had little effect on the real hardware, indicating that delays are not the sole cause of oscillation.

To prevent large oscillations in commanded swing foot acceleration, we penalize its derivative, jerk. To support this change, we increase the order of our policies by one from acceleration-based to jerk-based. We add acceleration as a 4th state (in addition to the already existing, time until touchdown, position, and velocity), and we replace acceleration as the control input with jerk. Jerk-based swing foot policies can be coordinated in the same way as the original acceleration-based policies within the ICS framework. When running the policy on the robot, we do not attempt to measure the actual acceleration. Instead, we have acceleration as a controller state that starts at 0 m/s/s at the beginning of swing and evolves ideally according to the commanded jerk. We then pass this acceleration on to the inverse dynamics, which considers a second order model of the system. From the point of view of the rest of the system, the jerk-based policies still produce a desired acceleration, but now we penalize its rate of change.

To keep computation reasonable, we must reduce the resolution of the state space grid for the other dimensions when adding an additional state space dimension. We partially mitigate the effect of this by also reducing the range of the grid so that the grid cells are closer to their original size. We now use a grid with minimum states of {0 m, -0.8 m/s, -20 m/s/s 0 s}, maximum states of {0.06 m, 0.8 m/s, 20 m/s/s, 0.6 s}, and resolutions of {61, 81, 61, 121}.

For the swing-Z subsystem, we replace the original cost function, (4.25) with

$$L(\mathbf{x}, \mathbf{u}) = \left(\frac{\ddot{z}}{\ddot{z}_{\max}}\right)^2 + \left(\frac{\dddot{z}}{\dddot{z}_{\max}}\right) + 2\left(\frac{z - z_{\text{nom}}}{z_{\text{nom}}}\right)^2, \tag{5.1}$$

where $\ddot{z}_{\max}$ is still 25 m/s/s and $\dddot{z}_{\max}$ is 300 m/s/s/s. We also add jerk to the analytic controller that takes over for $t_{td} \leq 0.04$s. We increase this limit from 0.03 s in the acceleration-based case because of the more limited control authority and decreased grid resolution. We now replace (4.26) with

$$C = \int_0^T 4\dddot{z} + 4\ddot{z}\, dt + 3000(\dot{z}_f - \dot{z}_{\text{nom}})^2, \tag{5.2}$$

and find the constant jerk, $\dddot{z}$ that minimizes (5.2). The constant coefficients are selected for the same reasons described in Section 4.2.1. We use an integral rather than simply multiplying by T as in (4.26) because acceleration, $\ddot{z}$, is not constant. We also loosen the touchdown timing constraint somewhat and now require that touchdown actually occur ($z = 0$) within 0.003 seconds of the nominal time. The looser constraint is necessary because of both the reduced control authority of the jerk-based policies and the slower time steps when controlling the robot (1 ms for simulation and 2.5 ms for the robot). We make a similar modification to the combined Swing-X/Swing-Y policy.

Figures 5.3 and 5.4 compare the response of the acceleration-based and jerk-based policies to a 20 ms delay in simulation. The jerk-based policy is less affected by the delay. Unlike simply adjusting the cost function, this improvement translated well to operating on hardware. Figure 5.5 compares the performance of the acceleration-based policy and jerk-based policy for walking in place on the Sarcos humanoid robot.

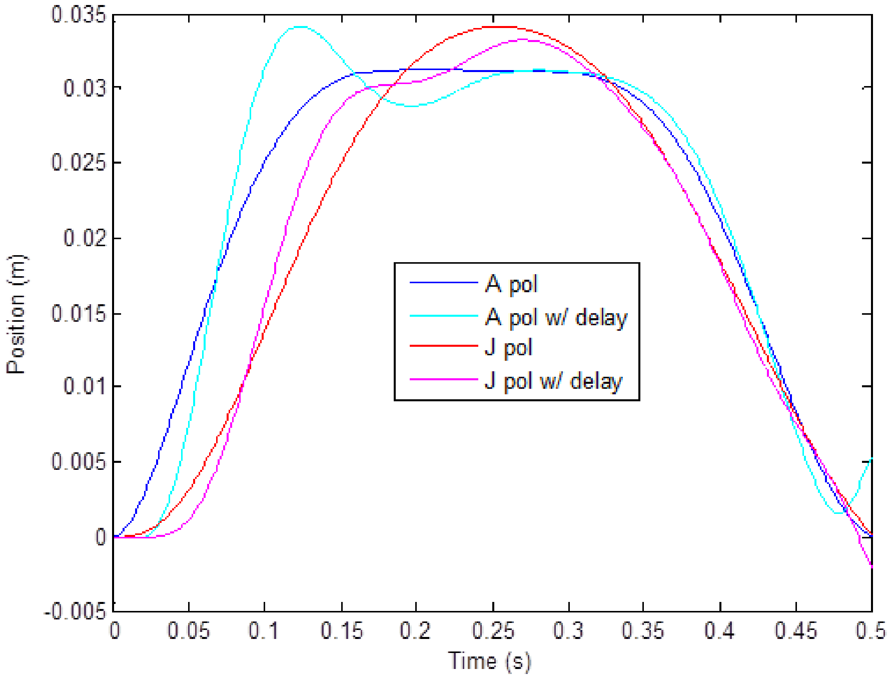

Figure 5.3: Comparison of the acceleration-based and jerk-based swing-z policies showing the position that results with and without a 20 ms delay.

5.4.2 Inverse Kinematics

We have found that in addition to the torques produced by our inverse dynamics, having PD controllers on individual joints with low gains helps to produce smooth motion. This is especially important for the swing leg during walking because we are unable to accurately achieve desired swing foot accelerations from inverse dynamics alone (likely due to an inaccurate mass distribution model). Unfortunately, our walking controller (Section 4.2) goes directly from system state to desired accelerations and torques without producing desired positions and velocities. To get the desired positions and velocities, we define virtual points representing the desired foot and CoM positions and velocities, then use Inverse Kinematics (IK) to find the desired joint positions and velocities.

We use virtual points to represent the desired positions of both feet in the x, y, and z directions

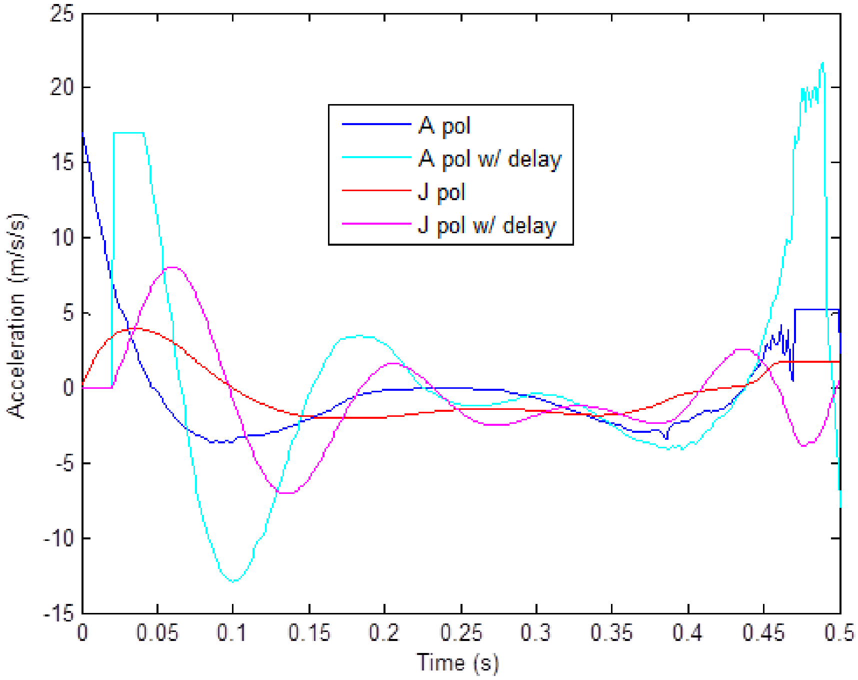

Figure 5.4: Comparison of the acceleration-based and jerk-based swing-z policies showing the acceleration that results with and without a 20 ms delay.

as well as the CoM in only the x and y directions (we use a constant desired CoM height), for 8 total degrees of freedom. For the two CoM degree of freedom, we treat the point as a generic second order system to which we apply the desired acceleration, a_{des} (produced by the high level DP policies using the virtual point state and the main controller's coordination state). To keep the points near the true system in the face of persistent acceleration errors, we also integrate (with gain k_I) them towards the measured position, p_{meas} and measured velocity, p_{meas}. The update equation for the virtual points' position, p, and velocity, v, therefore looks like

$$p_{i+1} = k_I T p_{\text{meas},i} + (1 - k_I T)(p_i + v_i T + 1/2 a_{\text{des},i} T^2) \tag{5.3}$$

$$v_{i+1} = k_I T v_{\text{meas},i} + (1 - k_I T)(v_i + T a_{\text{des},i}), \tag{5.4}$$

where the subscript i indicates the time step and T is the duration of a time step. Using (5.3) and (5.4), v will not actually be the derivative of p. In order to supply the inverse kinematics solver

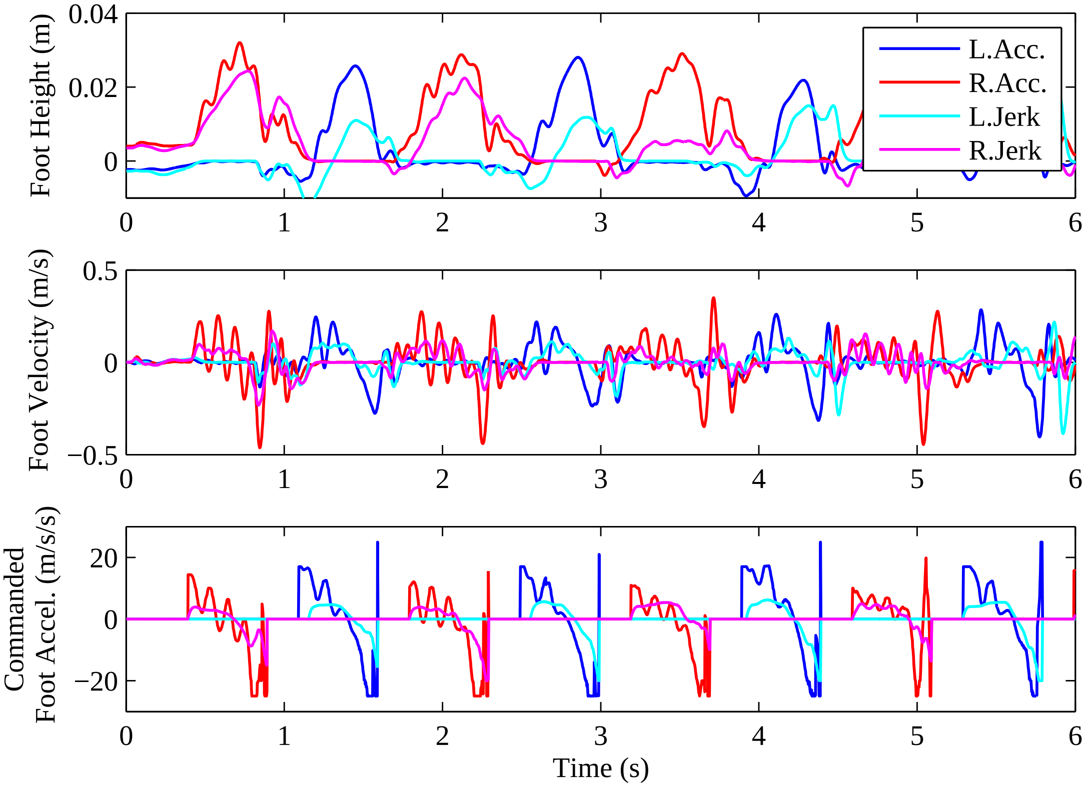

Figure 5.5: Comparison of the acceleration-based and jerk-based swing-z policies for walking in place on the Sarcos humanoid robot.

with something consistent, we therefore use the effective velocity $v_{\text{eff},i+1} = (p_{i+1} - p_i)/T$.

For the six foot degrees of freedom, the policies output desired jerks rather than accelerations, so we treat them as generic third order systems, while still anchoring the position and velocity to the measured values in the same manner. When a foot is in stance, we anchor it to $z = 0$, $\dot{z} = 0$, $\dot{x} = 0$, and $\dot{y} = 0$ instead of the respective measured values. Note that if the robot actually achieves the desired velocity, these virtual points will lie directly on the measured positions and have the measured velocities. Figure 5.6 in Section 5.5.1 shows the motion of the virtual CoM relative to the true CoM for walking in place.

Our inverse kinematics solver (adapted from the one described in [93]) maintains a reference pose, $\mathbf{q}_{\text{IK}}$. During each time step, we take the desired foot and CoM positions and velocities as well as the desired torso orientation and angular velocity and solve for the best velocities, $\dot{q}_{\text{IK}}$. We then update the positions $\mathbf{q}_{\text{IK},i+1} = \mathbf{q}_{\text{IK},i} + T\dot{\mathbf{q}}_{\text{IK},i}$. We can then use $\mathbf{q}_{\text{IK}}$ and $\dot{\mathbf{q}}_{\text{IK}}$ as desired positions and velocities for low gain individual joint PD controllers. We also use it as a reference pose within the inverse dynamics.

To solve for $\dot{\mathbf{q}}_{\text{IK}}$, we set up a quadratic program and use an off-the-shelf QP solver. The setup is similar to that used for the Inverse Dynamics described above, except that the only variables are the generalized velocities (6 base coordinates and 12 joint angular velocities) in $\dot{\mathbf{q}}_{\text{IK}}$. We again set up an overconstrained system with manually selected weights on the equations as in (4.46).

Dual IK

We use relatively low gains for the individual joint PD controllers based on the IK pose, $\mathbf{q}_{\text{IK}}$, and its derivative, $\dot{\mathbf{q}}_{\text{IK}}$, meaning that the actual angles can deviate significantly from $\mathbf{q}_{\text{IK}}$. Once some joints have deviated from the desired angle, it may be preferable for other joints to compensate, resulting in less total error through the kinematic chain, rather than adhering to their portion of the desired pose. Specifically, it makes sense to treat the swing and stance legs differently during single support. Torques in the stance leg are large and dominated by inertial forces, so the IK torques will be more of a subtle correcting factor. For this leg, it makes sense to use the desired torso orientation and CoM height because they are achieved precisely by the stance leg maintaining desired angles. The job of the stance leg is to position the torso relative to the stance foot.

The job of the swing leg, on the other hand, is to position the swing foot relative to the stance foot (not relative to the torso). Total torques in the swing leg are also much lower and motion is more dominated by non-inertial forces such as friction, so the IK torques will be more important. We therefore wish to actively compensate for stance leg errors with the swing leg. To do this, we need an IK solution that puts the two feet in the correct relative solution, but uses the true torso orientation and height. This way, errors in pelvis location do not necessarily translate to errors in swing foot location.

We implement this by maintaining and updating two independent IK poses, one that uses the desired torso orientation and height and one that uses the true torso orientation and height. During double support, we use the solution with desired orientation for both legs, but during single support, we use it only for the stance leg and use the solution with true torso orientation for the swing leg. To avoid discontinuous torques, which would excite higher-order dynamics in the physical system, we transition gradually between the two solutions at touchdown and liftoff for a period of 0.1 seconds. During the transition, we interpolate between the two solutions, shifting between them linearly.

This architecture is especially important because large torso orientation errors are inevitable. The hip adduct/abduct actuators, especially the right one, are insufficiently strong for normal walking motions, resulting in a large semi-controlled oscillation of the torso in the coronal plane (about the x-axis).

5.4.3 Integral Control

Modeling errors can cause persistent errors, especially with our emphasis on low gains. However, our hardware, while difficult to model, is very repeatable on a short time scale. What happened in the past is generally an accurate indicator of what will happen in the future and integral control can be a very effective means of compensating for modeling error.

Despite its usefulness in eliminating steady-state errors, use of integral control in the high-level controller can be problematic. While it will generally eliminate the specific error being integrated over, if not used carefully, it can create entirely new problems. For example, if we have a constant offset in CoM acceleration, we can achieve a stationary CoM by integrating the desired CoM acceleration. However, in the complicated multi-input-multi-output system that is our Weighted Objective Inverse Dynamics, this can lead to errors in other places. In this example, the joint accelerations necessary for a desired foot acceleration will be computed relative to the hip, which the WOID thinks is accelerating, but actually is not.

For this reason, we prefer to instead use integral control to modify the dynamic model. This way, the solved torques, accelerations, and contact forces remain consistent. This requires that we find some parameters to modify in the dynamic model that reasonably capture the type of error that is actually occurring. We choose to add virtual torques, τ_{virt}, on the torso to the dynamic model, changing the dynamics equations used by the low-level controller to

$$\mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} - \mathbf{S}\tau - \mathbf{J}^{\mathsf{T}}\mathbf{F} = -\mathbf{N}(\mathbf{q}, \dot{\mathbf{q}}) + \mathbf{J}_h^{\mathsf{T}}\tau_{\text{virt}}, \tag{5.5}$$

where $\mathbf{J}_h^{\mathsf{T}}$ is the hip Jacobian for the virtual torques. These are identical to the normal dynamic equations in the top row of (4.42) except for the addition of the final term. The virtual torques can directly represent the torques that the tether hoses put on the robot, but they can also generally represent any modeling error that tends to rotate the torso. We compute each component of τ_{virt} individually by integrating the commanded angular acceleration,

$$\tau_{\text{virt},i} = k_I \int \ddot{\theta}_{i,\text{des}}\, dt. \tag{5.6}$$

By imagining a virtual force causing the error, the low level inverse dynamics controller pushes back against it and corrects for it, without expecting unrealistic accelerations.

We correct steady-state translational errors by integrating a CoP offset, $\Delta\mathbf{p}$, based on the difference between the high level CoP command and the CoM location:

$$\Delta\mathbf{p} = k_I \int \mathbf{p}_{\text{HL}} - \mathbf{c}\, dt \tag{5.7}$$

where $\mathbf{p}_{\text{HL}}$ is the desired CoP produced by the high-level controller, without altering the desired CoM acceleration, $\ddot{\mathbf{c}}_{\text{des}}$. This will, on average, move the CoM over the high-level commanded CoP. This makes sense assuming that the high-level controller is not trying to obtain a net CoM acceleration at a time scale longer than the integration time constant.

At reasonable integration gains, k_I, CoP integration in the y direction interacts poorly with the walking controller, so we disable it during walking. Instead, we integrate $\Delta\mathbf{p}_y$ only during standing and then allow it to function as a constant offset during walking. The other offsets (virtual torques and x CoP offset) can all remain operational during walking. Continued integration of $\Delta \mathbf{p}_x$ is actually essential because it compensates for the increasing hose force as the robot walks forward and must drag a progressively longer section of hose behind it.

5.4.4 Additional Coordination Cost Terms

For the robot controller, we add several additional cost terms to the policy coordination step described in Section 4.2.2. It is necessary to add these terms to the primary policy value functions because the policies are based on simple models that do not capture all aspects of walking. These extra terms can be thought of as a value function for those otherwise unrepresented aspects of walking, and they prevent the controller from acting in ways that work for the simple models, but not for the full system (e.g. severe limping). In the simulation controller, during double support (Section 4.2.2), we pick the time until liftoff by minimizing $V(t_{\text{lo}}) = V_c(t_{\text{lo}}) + V_s(t_{\text{lo}})$. For the robot, we add an additional term:

$$V_t(t_{\text{lo}}) = (t_{\text{DS}} + t_{\text{lo}} - t_{\text{DS,NOM}})^2 w_t \tag{5.8}$$

where t_{DS} is the time so far elapsed in this double support phase, $t_{\text{DS,NOM}} = 0.2\text{s}$ is the nominal duration of double support, and w_t is a constant weight determining the importance of this term. The effect of this additional term is to make double support periods last closer to the nominal duration of 0.2 seconds. This helps reduce limping and prevents the controller from getting stuck in endless double support phases or skipping it entirely (except when absolutely necessary). It also encourages smoother walking closer to the nominal walking pattern.

During single support, as described in Section 4.2.2, we use an optimization to minimize $V(t_{\text{td}}, x_{\text{td}}, y_{\text{td}})$, which is a sum of the five individual policy value functions. For the robot, we add a term similar to (5.8) as in double support, and for much the same reasons, but using the nominal single support duration of 0.5 seconds. We also add a smoothing cost term, $V_{\delta t}(t_{\text{td}}) = (t_{\text{td}} - t_{\text{td,prev}})^2 w_{\delta t}$, which smooths out t_{td} and forces it to change gradually. Without the addition of this term, there will be significant noise in the selection of t_{td} because of discritization in the policies, incompletely converged policies, and incomplete convergence of the optimization minimizing $V(t_{\text{td}}, x_{\text{td}}, y_{\text{td}})$. Such noise results in an accompanying noise in the high level commands such as CoP and CoM acceleration, which can excite the higher order unmodeled dynamics of the robot and result in poor performance.

We also add one final term to reduce limping: $V_{sl}(x_{\text{td}}) = (x_{\text{td}} - x_{\text{td,NOM}})^2 w_{sl}$, where the nominal step length is computed from the desired walking speed and nominal step timing as $x_{\text{td,NOM}} = v_{\text{des}}(t_{\text{DS,NOM}} + t_{\text{SS,NOM}})$. This encourages the controller to take all steps of the same, nominal length, rather than alternating between long and short steps in a limping gait. A nearly identical effect could be achieved by modifying the cost function for the sagittal policy, but this implementation allowed greater flexibility for experimentation during development without having to recompute a policy.

5.4.5 Sidestepping Control

The coronal policy generated by DP has no notion of absolute position in the world; it measures CoM location relative to the feet. This makes it only neutrally stable with regard to global lateral motion, which in simulation is fine. Any disturbance in simulation pushes it a bit to the side, and while it will not return to the original path, it will return to the desired heading, only a bit to the side. On the real robot, however, modeling error can result in a constant asymmetric perturbation, which continuously drives the robot to sidestep. Without correction, this results in a constant rate of lateral motion, which can be quite significant. It can cut walking in place trials short by leaving the test area and destabilize forward walking.

We have two simple means of adjusting our control to counter the tendency to sidestep: we can adjust the CoP, or we can add yet another term to the cost function discussed in the previous Section to adjust lateral footstep locations. The former is intuitively like pushing on the CoM, and the later is intuitively like pushing on the feet. Used in a proportional controller, either can arrest the sidestepping on its own, but can destabilize the system. Adjusting the CoP, like pushing on the CoM, can cause it to fall in the direction we are trying to move the robot; adjusting the footsteps, like pushing on the feet, can cause it to fall in the opposite direction. Using the two methods in concert allows these two tendencies to cancel out and is intuitively akin to pushing both the feet and the CoM in the desired direction. We find good weights for the two proportional controllers experimentally.

When adjusting the footsteps laterally, we pick world coordinate touchdown locations for the left and right foot by taking the starting CoM location and moving half the nominal stance width in either direction. Then, we add at term that penalizes the square of the deviation from these touchdown locations to the optimization for single support touchdown parameters.

5.5 Results

5.5.1 Walking in Place

For walking in place, we can use a sagittal policy which is computed with a desired velocity of zero in the cost function, However, we usually use an analytic policy rather than a policy generated by DP for control of the CoM motion in the forward direction. If we do not specifically address it, we have more trouble with movement in world coordinates in this direction because of coupling between the vertical motion of the foot and the forward motion of the foot. If we simply try to stay in one place, the robot will slowly walk forward. A small proportional gain on position added to the CoP is insufficient to keep it in one place. To remain stationary, we select the footstep location such that for the next footstep (in 0.7 seconds), the capture point will be at the desired CoM location, which we assume to be the origin without loss of generality. Note that if the capture point is already at the origin, then placing the foot there will result in the system coming to rest at the origin and all subsequent capture points and steps being there. Therefore, this is theoretically a deadbeat controller. We compute the desired footstep location analytically, by starting with the LIPM dynamic equation

$$\ddot{x} = \frac{g}{h}(x - p) \tag{5.9}$$

where h is the height, x is the CoM location, and p is the CoP location. If we solve the differential equation, we get

$$x(t) = C_1 e^{\sqrt{\frac{g}{h}}t} - C_2 e^{-\sqrt{\frac{g}{h}}t} \tag{5.10}$$

and its derivative

$$\dot{x}(t) = C_1\sqrt{\frac{g}{h}} e^{\sqrt{\frac{g}{h}}t} + C_2\sqrt{\frac{g}{h}} e^{-\sqrt{\frac{g}{h}}t}, \tag{5.11}$$

where C_1 and C_2 are unknown constants depending on the initial conditions. If we add the initial conditions (2 equations) and the constraint that the capture point be at the origin on the next step (1 equation), we get a system of 5 equations and 5 unknowns: C_1, C_2, x_1, $\dot{x}_1$, and p, where x_1 is the CoM position at the next step. We can then solve this system to find where to step now

$$p = \frac{e^{\sqrt{\frac{g}{h}}T}(x_0\sqrt{\frac{g}{h}} + \dot{x}_0)}{\sqrt{\frac{g}{h}}(e^{\sqrt{\frac{g}{h}}T} - 1)}, \tag{5.12}$$

where T is the time until the following touchdown. To coordinate this with the swing foot policy, we need a value function to specify how important it is that we actually put the foot at p. We construct a parabola with its vertex at $\{p, 0\}$ and a constant quadratic term to act as the value function. This analytical sagittal value function has no dependence on footstep timing, meaning that $\partial V_s/\partial t_{\text{td}} = 0$ and $\partial V_s/\partial t_{\text{lo}} = 0$.

As in simulation, we can stop the robot simply by switching to a standing controller from the walking controller. The transition goes much more smoothly if done during double support, especially early in double support. If done during single support, the swing foot stomps violently as the robot tries to abruptly put weight on it. We normally stop walking by switching to the standing controller at the end of a single support phase.

The controller has a nominal desired cadence of one step every 0.7 seconds (0.2 for double support and 0.5 for single support), but in practice steps slightly slower, with a cadence of one step every 0.75 seconds. Figures 5.6 and 5.7 show results for normal, unperturbed walking in place. Ideally, the time until transition trace would always have a slope of -1 (except at discontinuities), with the larger rises indicating single support and the short rises indicating double support. Double support deviates much farther from this ideal "countdown" slope. You can also see that single support tends to end earlier than predicted. Single support ends when a significant force is detected with the foot force sensors and may occur early either because the foot is lower than desired or because it is not flat, resulting in part of the foot touching down early. The foot force/torque sensors are not generally reliably accurate, and this binary touchdown detection is the only place where they are used in the controller. We do, however, use them for analysis, such as plotting the measured CoP, as in Figure 5.6.

Robustness

Walking in place is very robust. Generally, if undisturbed, it will run indefinitely without failure; we have run the walking controller continuously for one hour without human intervention. One way to test its robustness is by perturbing it with an external push. We push the robot either by hand, or with a stick that has a force sensor on the end, which can measure the force (and impulse) of a push. Unfortunately, when using the stick, we are limited to pushing normal to large flat surfaces in order for the force sensor to get a good reading. For lateral pushes, this limits us to pushing on either the hip or the backpack, either of which imparts a significant yaw moment, which the controller is less capable of handling. The largest measured push that the system recovered from was a push of approximately 40 Ns to the hip, peaking at about 150 N and lasting about 0.3 seconds. Figure 5.8 shows a trial in which the system recovers from two pushes to the side of the backpack, of 33 Ns and 25 Ns respectively.

The size of the push that the controller can handle is heavily dependent on the timing of the push, largely because our controller can not cross one foot in front of or behind the other, meaning that it has much more freedom to step to the outside than to the inside. Therefore, it is

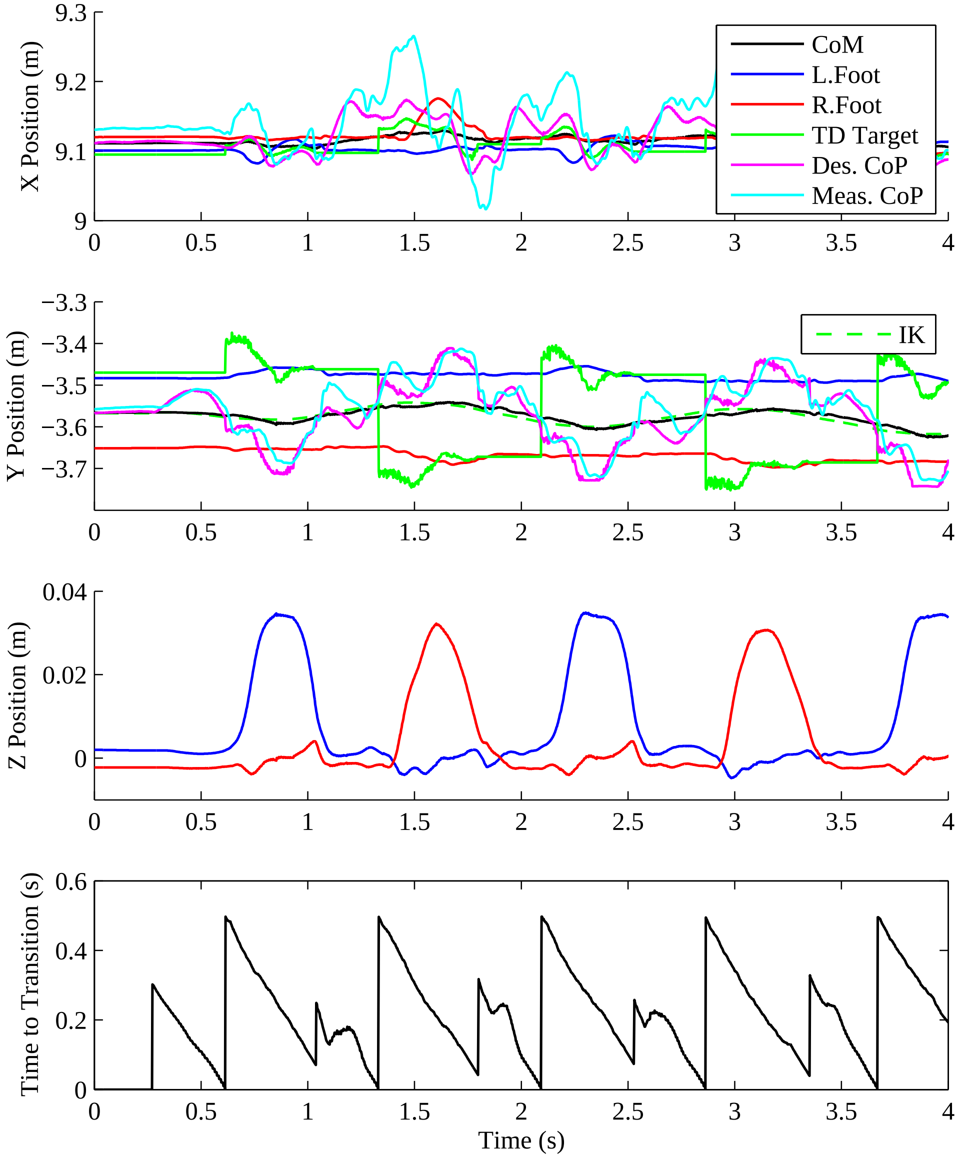

Figure 5.6: Walking in place starting from standing still. Note that the measured CoP is measured using the internal force/torque sensors in the robot's feet and may be inaccurate. The IK trace shows the location of the virtual point used in the inverse kinematics solution as described in Section 5.4.2.

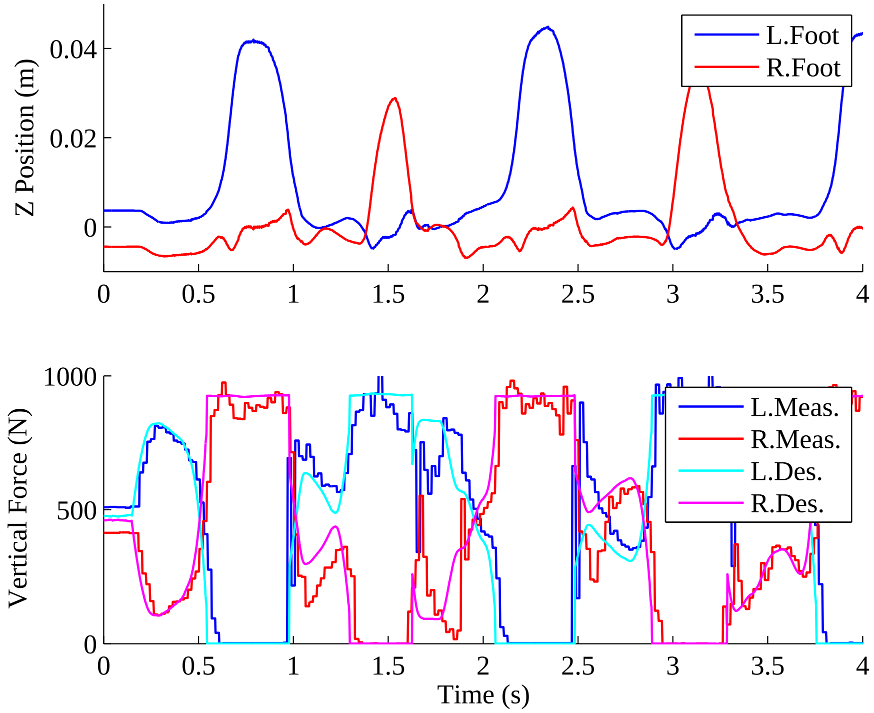

Figure 5.7: Vertical forces during walking in place starting from standing still. Measured vertical forces are measured with external force plates, which are more accurate than the internal force sensors, but have a slow update rate. The robot was limping substantially during this trial, lifting the right foot less than the left.

much more robust to pushes either during double support (when it can delay liftoff and handle the push with both legs) or early in single support when pushed towards the swing foot (when it can adjust the touchdown location outward).

The walking in place controller is also robust to perturbations to the feet. Our controller uses only inverse dynamics plus low PD gains, so the swing leg is quite compliant and can be disturbed without causing the robot to lose balance. It is possible to step on one (or both) of its feet, preventing it from lifting them without causing a failure. We also tested pushing on the foot during swing to alter the touchdown location. We were able to deflect the touchdown location up to about 0.2 m without causing failure. Figure 5.9 shows a photograph of such an experiment.

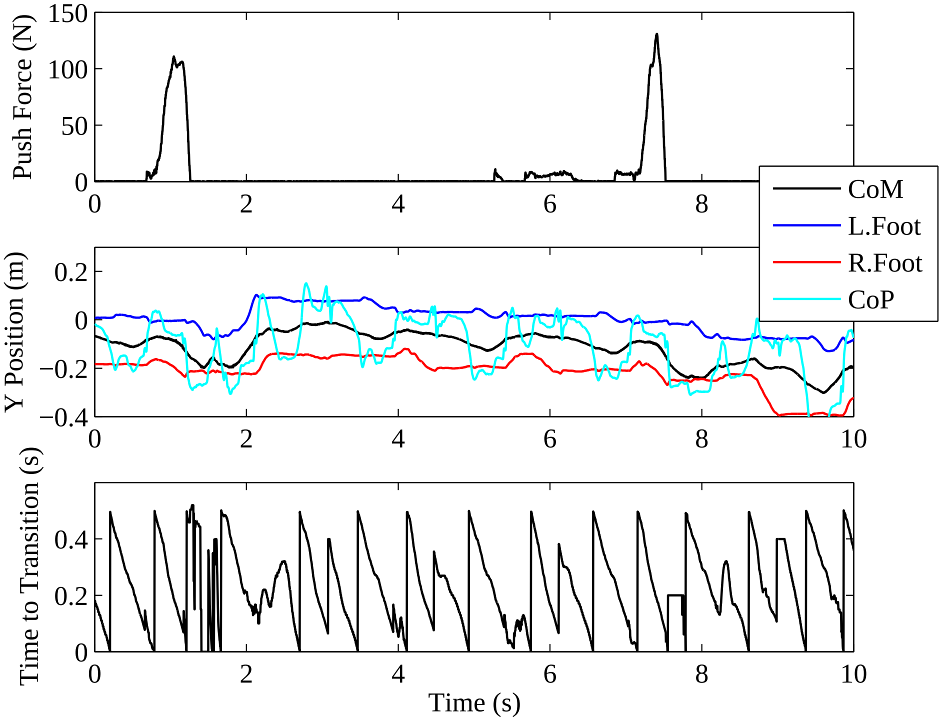

Figure 5.8: Response to two pushes to the right (-Y direction) of 33 and 25 Ns respectively. Both pushes are applied to the side of the backpack with a force-sensor-on-a-stick. CoP is measured by the internal force/torque sensors and inaccurate.

5.5.2 Walking Forward

The only change necessary to the walking in place controller to walk forward is replacing the analytic sagittal policy with one computed by DP. How fast it walks depends on the desired speed in the cost function when computing the sagittal policy, but the faster it goes, the less reliable it becomes. If we use a policy computed with a desired velocity of 0.2 m/s, the robot walks with an actual average speed of 0.17 m/s, as measured by internal odometry. At this pace, the system is quite reliable, and all of our perturbation experiments are done at this speed. If we use a policy computed with a desired velocity of 0.25 m/s, it walks with an average speed of about 0.21 m/s; at this speed it usually walks successfully, but is not very robust to disturbances. The fastest

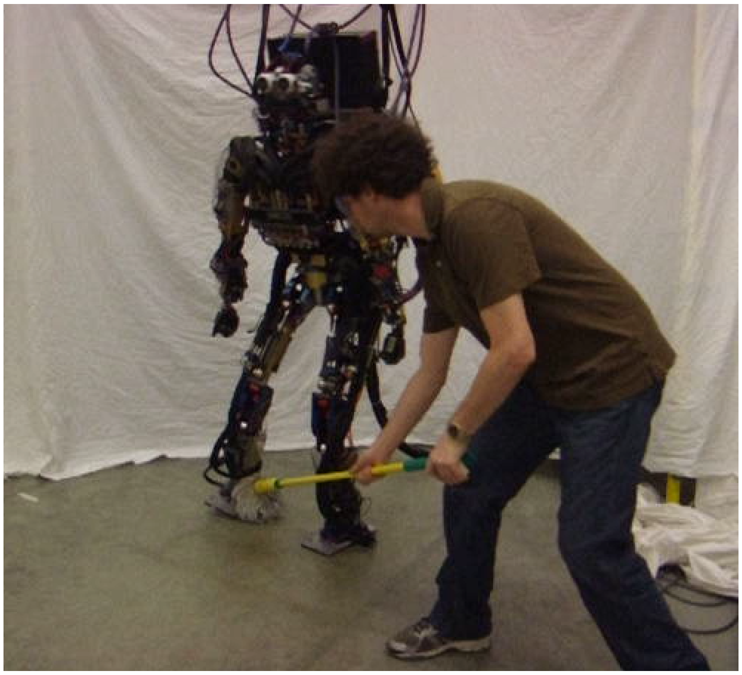

Figure 5.9: A photograph of pushing on the foot during walking in place. We use a mop to displace the foot up to about 0.2 m without causing failure.

walking we were able to achieve had an average speed of 0.25 m/s using a policy computed with a desired velocity of 0.3 m/s, but at this pace, walking was not reliable and often failed within a few steps. Figure 5.10 shows data from unperturbed walking at 0.22 m/s (desired speed 0.25 m/s).

Robustness

We test the robustness of our walking controller with a variety of experiments, all using the 0.2 m/s desired velocity walking policy. Like walking in place, we test our forward walking controller by pushing it. Unsurprisingly, it was somewhat less robust to external pushes while walking forward than while walking in place as well as being more sensitive to the timing and direction of the push. The largest push that we measured with the push stick that it was able to recover from was a rearward push of 29Ns (peaking at around 90 N and lasting about 0.4 seconds) applied to the large black manifold on the robot's chest. A trial containing this push

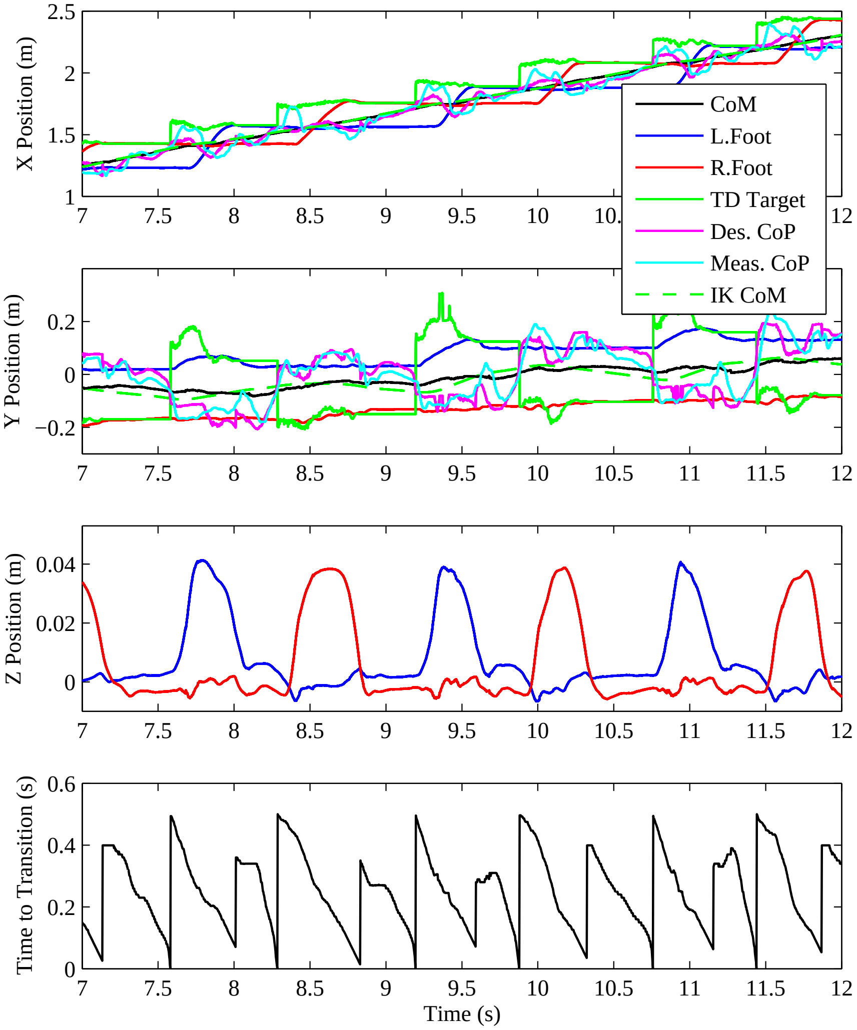

Figure 5.10: Walking forward undisturbed at 0.22 m/s (desired speed 0.25 m/s). TD target refers to the planned location of touchdown for the current step. IK CoM refers to the location of the virtual point used for inverse kinematics as discussed in Section 5.4.2. The measured CoP is measured using the internal force plates and may be inaccurate.

and a second push of 22Ns is shown in Figure 5.11. Pushing the robot with the push stick is pictured in Figure 5.12.

We also test our controller by putting two different types of obstacles in its path (both shown

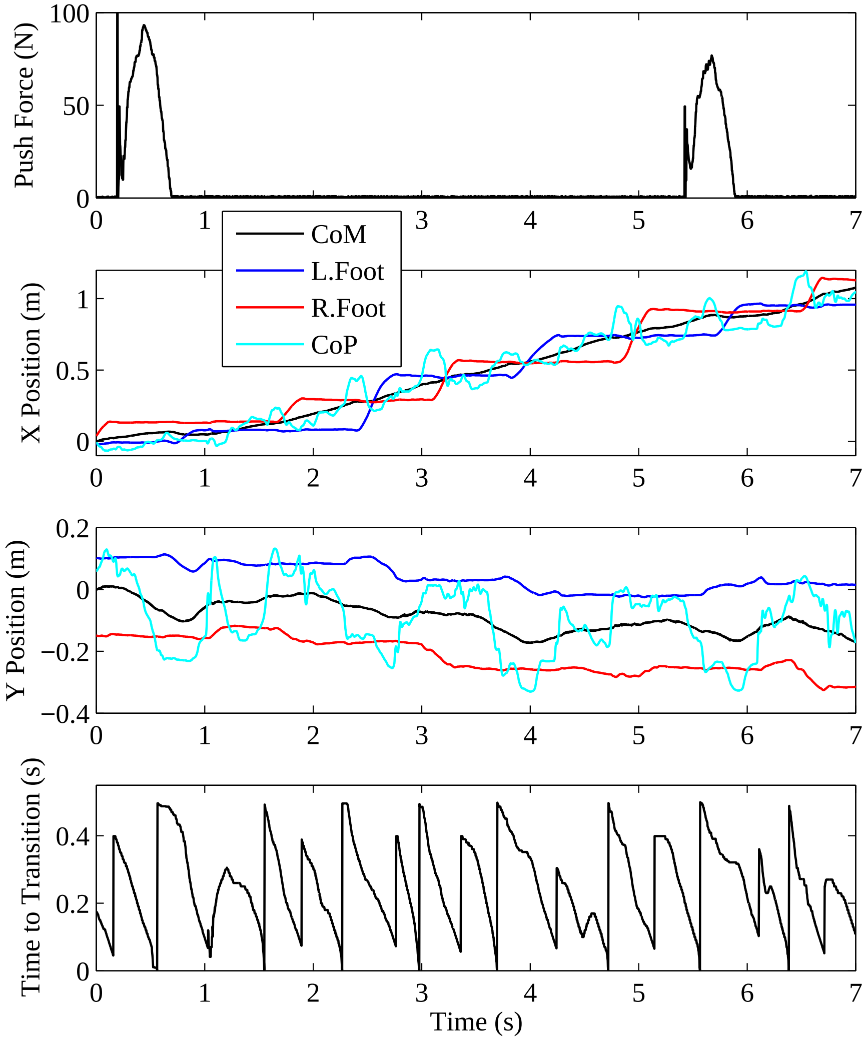

Figure 5.11: Walking forward with two rearward pushes of 29 Ns and 22 Ns respectively applied to the the robot's chest. The measured CoP is measured using the internal force plates and may be inaccurate.

in Figure 5.12. The first type of obstacle we tested was hammers, an irregular obstacle that is small enough for the robot to step onto, but large enough to tilt the foot at a significant angle. Our controller does not require the foot to be flat, and in fact has a compliant ankle, even during stance. This allows the controller to walk successfully even with the foot tilted significantly so long as it is able to achieve approximately the desired CoP. If the obstacle is near the center of the foot, it can even roll from having the foot tilted up to tilted downward as the CoP moves from the back of the foot to the front. The other obstacle type we tested was a standard brick. Since the brick is taller than the robot lifts its feet, it can not step onto the brick. Instead, it walks through the brick, kicking it out of the way over the course of several steps.

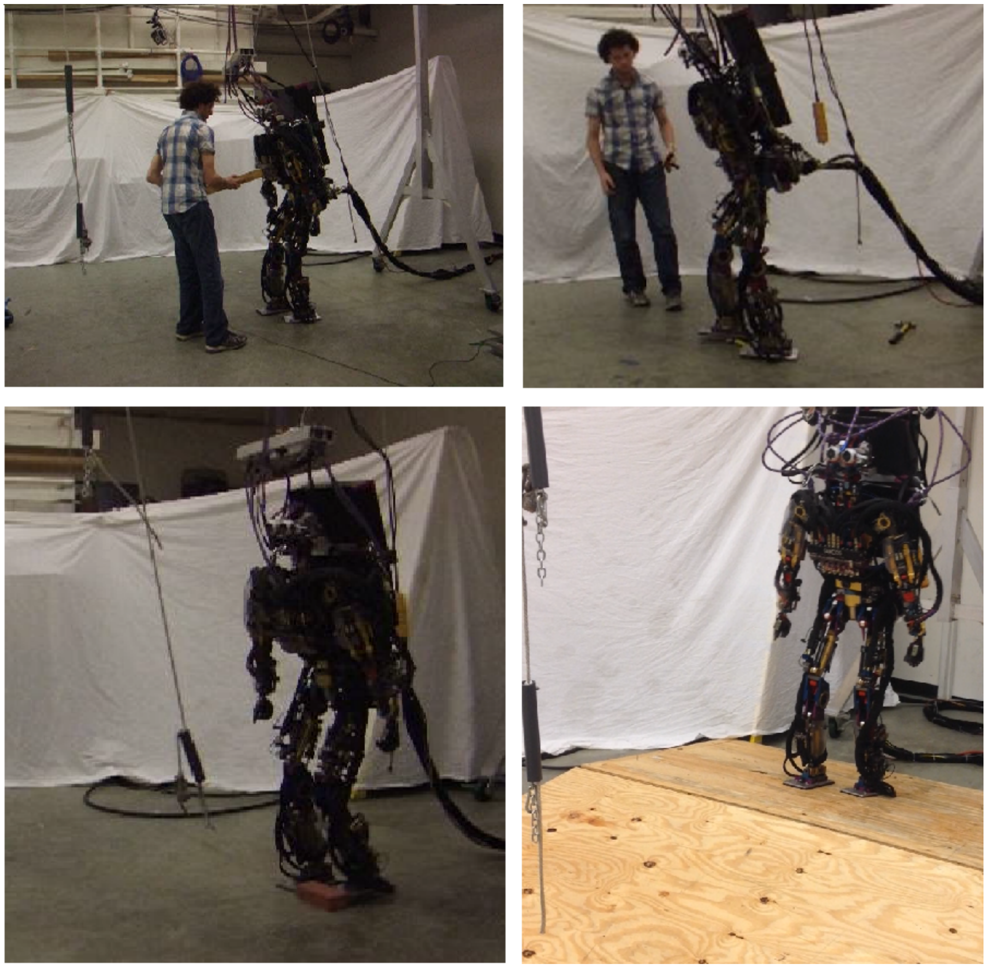

Figure 5.12: Various walking robustness experiments. Pushing with the force stick (upper left), stepping on hammers (upper right), kicking bricks (lower left), walking on the ramp (lower right).

Finally, we tested the ability of our controller to walk on sloped surfaces. We constructed a ramp with a 1.2 m long slope pitched 8 degrees upwards followed by a similar slope pitched downward (shown in Figure 5.12). The robot was stable walking up the ramp, but borderline walking downwards. When walking upwards, the slope causes touchdown to occur earlier than expected, which is not a problem because the foot is already near the desired touchdown location and can simply remain in double support with most of the weight on the rear leg longer to compensate. When walking down the slope however, the slope causes touchdown to occur later than expected. This becomes a problem when the swing knee straightens completely before touchdown occurs, leaving no way to continue lowering the foot while maintaining the desired CoM height. This behavior could be improved by specifically addressing it in the controller; it could bend its stance knee to lower the CoM and swing foot. Additionally, the expected touchdown height could be adjusted based on the slope of the previous step or steps. Interestingly, the 8 degree slope used here is nearly twice as steep as the simulated walker could handle, likely because the simulation was walking nearly 3 times faster and lifting the swing foot less.

Chapter 6

Robust Dynamic Programming

Model based optimal control is a powerful tool that suffers from a major difficulty: it requires a model of the system. Developing such a model can be a difficult task. Even if you can find a good model structure, parameter measurement and sensor calibration can be inaccurate. Even if initially accurate, the model may change over time due to wear, temperature, weather, contact condition, or any number of unforeseen influences. Additionally, you may wish to develop a single controller for a large number of instances of a manufactured product which have variations between them due to manufacturing inconsistencies or unequal change over time. In all of these cases, the controller must cope with a model that is not identical to the true system.

Optimal control methods can often be very sensitive to even small modeling errors. Optimizers are very good at finding and exploiting any possible advantage, even if that advantage is a modeling error. Optimal paths often lie along constraints, so even small errors can make them infeasible [4]. Methods that plan a long way into the future may rely on the first portion of their plan working precisely in order for the latter portion to perform well.

Most methods that aim to reduce the sensitivity of optimal control algorithms to modeling error fall into one of two general categories: (i) minimizing an expected cost over possible outcomes or (ii) minimax formulations that minimize the maximum cost (worst case) of possible outcomes.

For expected cost approaches, a distribution of possible outcomes is required. It can be either a discrete list of possibilities (with associated probabilities) or a continuous distribution. The distribution can take many forms, ranging from additive output noise to sets of models. Some examples include additive or multiplicative state or control process noise on top of the nominal dynamics [51, 100], random variables in the process model [91], arbitrary state-dependent stochastic transition functions [109], and Markov decision processes (MDP) with known or even unknown transition probabilities [73].

Minimax methods are concerned with minimizing the worst case cost, so the probabilities of the potential outcomes are irrelevant and unnecessary. Again, there exists a large variety of ways to describe the set of possible outcomes. An H_∞ controller for linear systems can be found by solving a Riccati equation [25]. These controllers are robust to bounded input and process noise. Some examples for nonlinear systems include additive disturbances [69], discrete sets of models [8], and continuous sets of models [24]. Minimax formulations are generally able to provide stronger theoretical guarantees of robustness because they deal with worst case scenarios.

Most robust control algorithms are modifications of existing algorithms for deterministic problems. Differential Dynamic Programming (DDP) [51, 69, 100] and Model Predictive Control (MPC) [14] are popular techniques based on optimizing trajectories that can be modified to produce robust controllers. Dynamic Programming (DP) is a class of algorithms that rely on the observation that any portion of an optimal trajectory is itself optimal. This leads to a situation where efficient computation can be achieved by reusing the solution to overlapping subproblems [12, 17, 88]. Most versions of DP produce control policies that are valid for large regions of the state space. In this section, we will explore several methods to make DP robust.

A popular approach to robust control law design is to optimize a policy by evaluating its performance in simulation on a distribution of possible models [8, 9, 11, 66, 75, 78, 87, 96, 103]. In this section, we discuss and compare an error-as-noise approach, a mini-max approach and two multiple model approaches to making dynamic programming robust. The first Multiple

Model Dynamic Programming (MMDP) algorithm presented is the heuristic approach published in [106], and the second MMDP algorithm presented is a locally optimal variant.

6.1 Pendulum Swing-Up

We will demonstrate our algorithm on the problem of inverted pendulum swing-up. The controller must apply torques to a rigid pendulum in order to raise it to the inverted position and maintain it there. This is a good test problem because it involves both a dynamic travel component (getting to the inverted position) and a regulation component (remaining at the inverted position once there). The inverted pendulum is also one of the simplest nonlinear systems and a system about which we have good physical intuition, which makes it easier to interpret our results. The policy can also be severely limited by action-space constraints with it still remaining possible to achieve the goal from anywhere in the state space.

The pendulum is a second order system with one degree of freedom, so it has a 2-dimensional state space, $\mathbf{x} = \{\theta, \dot{\theta}\}$, where θ is the pendulum angle ($\theta = 0$ defined as upright), and the dot indicates a derivative with respect to time. It has a one-dimensional action space, $\mathbf{u} = \{\tau\}$, where τ is the control torque. The pendulum dynamics are given by

$$\ddot{\theta} = \frac{mLg\sin(\theta) + \tau}{mL^2}, \tag{6.1}$$

where m is the mass, L is the length, and $g = 9.81\mathrm{m/s}^2$ is the acceleration due to gravity. For the nominal system (we will perturb it later) the mass is 1 kg and the length is 1 m. We also constrain the control torque to $|\tau| \leq 1.5\mathrm{Nm}$, which requires the system to swing back and forth multiple times before reaching the goal.

The swing-up task is formally defined as minimizing the total cost, C, given by the integral,

$$C = \int_0^\infty L(\mathbf{x}, \mathbf{u})\, dt, \tag{6.2}$$

of the one step cost function,

$$L(\mathbf{x}, \mathbf{u}) = \theta^2 + 0.5\dot{\theta}^2 + \tau^2. \tag{6.3}$$

Since angles are topologically circular, we can represent the entire θ direction with a finite grid. However, we must bound the $\dot{\theta}$ dimension to the somewhat arbitrarily selected range of $\pm 10 \mathrm{rad/s}$. A grid resolution of at least about {50, 100} is necessary to achieve swing-up. All experiments in this Section were performed with a resolution of {400, 900} to give a good trade-off between computation time (5 minutes on a PC using a 6-core processor with a clock speed of 3.33 GHz) and policy quality. This gives a resolution of {0.016 radians, 0.022 radians/second} and a grid of 360,000 discretized states. It takes about 500 iterations to generate a controller that can achieve swing-up. We run all of our experiments for 5000 iterations to allow the policy and value function to converge very closely to the optimum.

Note that our DP algorithm requires the discrete time form of the pendulum dynamics, which we obtain by integrating (6.1):

$$\begin{bmatrix} \theta_{k+1} \\ \dot{\theta}_{k+1} \end{bmatrix} = \begin{bmatrix} \theta_k + \dot{\theta}_k T + 0.5\ddot{\theta}_k T^2 \\ \dot{\theta}_k + \ddot{\theta}_k T \end{bmatrix} \tag{6.4}$$

where $T = 0.001 \mathrm{seconds}$ is the time step, and $\ddot{\theta}$ is determined by (6.1).

6.2 Tests of Robustness

The goal is to produce stationary controllers, $\mathbf{u} = \pi(\mathbf{x})$, that are as robust as possible to modeling error. By this, we mean that we wish the controllers to be effective on a set of modified pendulum systems. We measure the effectiveness of a particular policy, $\pi(\mathbf{x})$, on a particular model by starting the system at $\mathbf{x} = \{\pi, 0\}$ (down), simulating it forward in time, and measuring (6.2). If the system does not reach the goal ($\mathbf{x} = \{0, 0\}$) and remain there within 60 seconds of simulation, we assume it never will and assign an infinite cost.

In order to test the robustness of our policies, we will modify the nominal model in four ways:

I. Varying the mass, m.

II. Varying the length, L.

III. Adding varying amounts of viscosity, ν. Equation (6.1) becomes

$$\ddot{\theta} = \frac{mLg\sin(\theta) + \tau - \nu\dot{\theta}}{mL^2}. \tag{6.5}$$

IV. Misaligning the goal with gravitational up, representing an off-center mass or sensor miscalibration. An offset, θ_0 is added to (6.1), changing it to

$$\ddot{\theta} = \frac{mLg\sin(\theta - \theta_0) + \tau}{mL^2}. \tag{6.6}$$

For the final test, there is no state-action pair that results in both no acceleration and no cost, L, so the cost given by (6.2) will always be infinite. To accommodate this, we consider the task complete and stop integrating (6.2) after the system has remained near the goal for 15 seconds.

6.3 Modeling Error as Noise

Much work has been done on generating optimal controllers for stochastic systems by optimizing the expected value of the cost function. Modeling error is not actually stochastic, but these techniques can be used to increase robustness to modeling error by generally increasing robustness to unexpected dynamics.

For DP, we do not have an analytical expression for the value function, so we take the expectation of the stochastic dynamics by sampling from it, turning (2.3) into

$$V(\mathbf{x}) = \min_{\mathbf{u}\in\{\mathbf{u}_0,\mathbf{u}_r\}} \sum_{i=1}^{N} (L(\mathbf{x}, g_i(\mathbf{u})) + V(f_i(\mathbf{x}, g_i(\mathbf{u}))))p_i \tag{6.7}$$

$$\sum_{i=1}^{N} p_i = 1 \tag{6.8}$$

where g_i is an instantiation of the control noise, f_i is an instantiation of the process noise, and p_i is the probability of the ith sample. To avoid optimizing over the instantiations of the random noise, we must fix the g_i's, f_i's, and p_i's for all iterations, which essentially forces us to approximate the noise distribution as a sum of delta functions. In general, this can require a large number of samples (and correspondingly large computational cost) to adequately model the distribution.

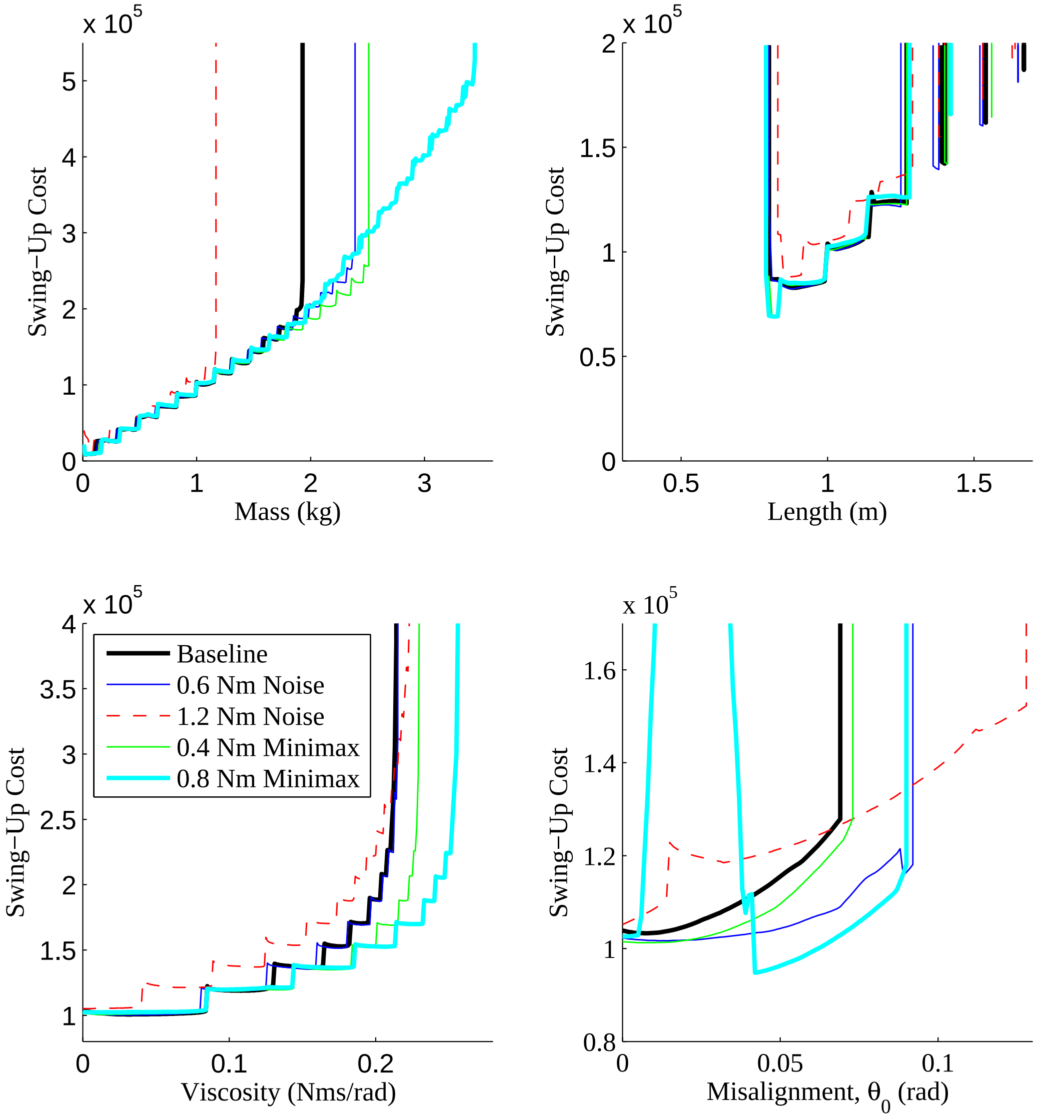

Figure 6.1: Robustness to model variations of mass, length, viscosity, and gravity misalignment. Results are shown for the baseline DP (Section 2.2), DP with small and large amounts of noise (Section 6.3), and DP with small and large minimax perturbations (Section 6.4).

For the case of additive noise, however, the Central Limit Theorem tells us that regardless of what distribution we use, after a large number of time steps, the total noise will have an approximately Gaussian distribution. This frees us to use a small number of samples and rely upon summation over many time steps to produce a fuller distribution. For our swing-up example, we used no process noise ($f_i = f$), and additive control noise ($p_1 = p_2 = 0.5$, $g_1(\tau) = \tau - \Delta$ and $g_2(\tau) = \tau + \Delta$ where Δ is a parameter that controls the size of the noise). Note that additive control noise is identical to additive acceleration noise because $\partial\ddot{\theta}/\partial\tau = 1/mL^2$, which is constant.

Figure 6.1 shows robustness results for stochastic DP compared to the baseline DP implementation. Discontinuities in the cost result from changes in strategy: using more or fewer back and forth swings to reach the upright position. In general, robustness to modeling error increases as the noise level increases. For all four tests, the large noise results in a significantly increased cost of swing-up. This is a general problem for methods aimed at improving robustness; the cost increases when minimizing it is no longer the sole goal of optimization.

6.4 Minimax Formulation

The unexpected dynamics introduced by modeling error are poorly represented by random noise because they are not independent at every time step. The errors introduced by modeling error often consistently push the system in the same direction, a situation which can get ignored by noise-based formulations as being extremely low probability. Another possibility for handling noise is to use a minimax formulation. Rather than minimizing the expected value, minimax algorithms attempt to minimize the maximum value or worst case scenario. In [69], a minimax version of DDP is developed and demonstrated on a walking robot.

We implement minimax DP much like stochastic DP, but instead of taking the expected value, we take the maximum, so the Bellman equation becomes

$$V(\mathbf{x}) = \min_{\mathbf{u}\in\{\mathbf{u}_0,\mathbf{u}_r\}} \max_i L(\mathbf{x}, g_i(\mathbf{u})) + V(f_i(\mathbf{x}, g_i(\mathbf{u}))). \tag{6.9}$$

We no longer require the Central Limit Theorem to approximate the full distribution. Instead, we rely upon the assumption that the worst case disturbance will be an extreme disturbance: either the maximum push in one direction or the maximum push in the other direction. This assumption is an approximation for nonlinear systems, but in the case of short time steps, additive disturbances, and affine in controls, it is very nearly accurate. This again allows us to keep the computation manageable by using only two samples. For the pendulum example, we again use no process noise ($f_i = f$), and additive control noise ($g_1(\tau) = \tau - \Delta$ and $g_2(\tau) = \tau + \Delta$).

Figure 6.1 also shows results for the minimax version of DP. Both methods generally increase robustness to modeling error with increased magnitude of the disturbance until they reach a point of diminishing returns. This point can be quite abrupt, and it can be different for different types of error. For each method, we show results for two different disturbance sizes, one slightly before the point of diminishing returns on any of the tests, and another where performance has decreased on some tests but not on others. As expected, the minimax formulation generally outperforms the noise formulation, but it does well on different tests. Qualitatively, these trends can be summarized by noting that the minimax formulation tends to do better when pushing harder is necessary to overcome the unexpected modeling error (increased mass and viscosity), but the noise formulation deals better with small errors at inopportune times (misaligning the goal with gravitational up). Discontinuities in the cost result from changes in strategy: using more or fewer back and forth swings to reach the upright position.

6.5 Heuristic Multiple Model Dynamic Programming

If the true system is deterministic, but different from your model, neither random nor worst-case disturbances are a good description of the true error. We are interested in the situation where there exists one true, deterministic model, but we do not know what it is. We present two versions of Multiple Model Dynamic Programming (MMDP) to solve this problem. In this Section, we discuss Heuristic MMDP (HMMDP), and in the following Section, we present Locally Optimal

MMDP (LOMMDP).

In MMDP, we wish to generate a single policy, $\mathbf{u} = \pi(\mathbf{x})$, that is optimized to perform well on a set of M candidate models. We denote the dynamics of the mth model as $\mathbf{x}_{k+1} = f_m(\mathbf{x}_k, \mathbf{u}_k)$. This method only directly optimizes performance for the specific models it is given, but we have a reasonable expectation of good performance for a range of model space around each of the candidate models. If we know something about the type of modeling error we expect, we can choose candidate models that cover the range of expected models. Formally, we wish to find the policy, π, that minimizes the total cost criterion,

$$C = \sum_{m=1}^{M} \sum_{x_0} \sum_{t=0}^{\infty} L_m(\mathbf{x}_t, \pi(\mathbf{x}_t)). \tag{6.10}$$

The innermost sum is trajectory cost and is the discrete form of (6.2). The middle sum is over trajectories started at each grid point. The outermost sum is over all M models.

For a single model, we have the convenient property that a single policy is optimal for all start states. Unfortunately, this property does not hold for the multiple model case. To show this, consider a policy that is optimal for a single start state. If we have a single start state, a state-indexed policy can control each of the models independently because they will travel through different states. We can therefore construct an optimal policy for multiple models with a single start state by copying the optimal policy along the optimal trajectory from each of the individual models' optimal policies. The optimal action at a given state will therefore depend on which model passes through it (and which corresponding optimal policy we must copy from). Since which model passes through a given point depends upon the start state, the optimal policy must also depend upon the start state.

We modify our DP algorithm to approximately minimize (6.10) by maintaining individual value functions, $V_m(\mathbf{x})$, for each of the M dynamic models (but only a single policy). To update a grid point, we pick the action by adding the value from each of the models:

$$\pi(\mathbf{x}) = \arg \min_{\mathbf{u} \in \{\mathbf{u}_0, \mathbf{u}_r\}} L_m(\mathbf{x}, \mathbf{u}) + \sum_{i=1}^{N} p(m) V_m(f_m(\mathbf{x}, \mathbf{u})), \tag{6.11}$$

where $p(m)$ is the probability of each model. We then update each value function individually according to

$$V_m(\mathbf{x}) = L_m(\mathbf{x}, \pi(\mathbf{x})) + V_m(f_m(\mathbf{x}, \pi(\mathbf{x}))). \tag{6.12}$$

As in the baseline implementation, we do many iterations of updating every point in the grid this way. This gives us a policy, $\pi(\mathbf{x})$, that is approximately optimized to perform well on all of the models, and a value function, $V_m(\mathbf{x})$, for each of the models given this policy.

To analyze our update rule with respect to (6.10), we note that the value function, V, is exactly equal to the innermost sum, so we can rewrite (6.10) as

$$C = \sum_{i=m}^{M} \sum_{x_0} p(m) V_m(x_0). \tag{6.13}$$

We define $h_m(\mathbf{x})$ as the number of states for which if you start a trajectory there it will pass through state $\mathbf{x}$ for model m (and a given policy). If we make a single change to our policy at state $\mathbf{x}$, it will change the value at that state as well as at any state whose trajectory passes through it. All other values will remain the same. We can write the resulting change in total cost as

$$\Delta C = \sum_{i=m}^{M} p(m) \left[\Delta V_m(\mathbf{x}) + h_m(\mathbf{x}) \Delta V_m(\mathbf{x})\right] \tag{6.14}$$

where ΔV_m is the new value minus the old value for model m. To ensure that this is decreasing, we must assume that

$$h_i(\mathbf{x}) = h_j(\mathbf{x}) \quad \forall i, j, \mathbf{x} \tag{6.15}$$

which means that all models have an equal chance (given a random start state) to get into any given state. If this were not true, it would make sense to focus on minimizing the value for the models that are more likely to end up in that state, which our algorithm does not do. This assumption will not generally be true, but for many problems where the candidate models are similar, it will be approximately true. LOMMDP presented in the next Section addresses this issue.

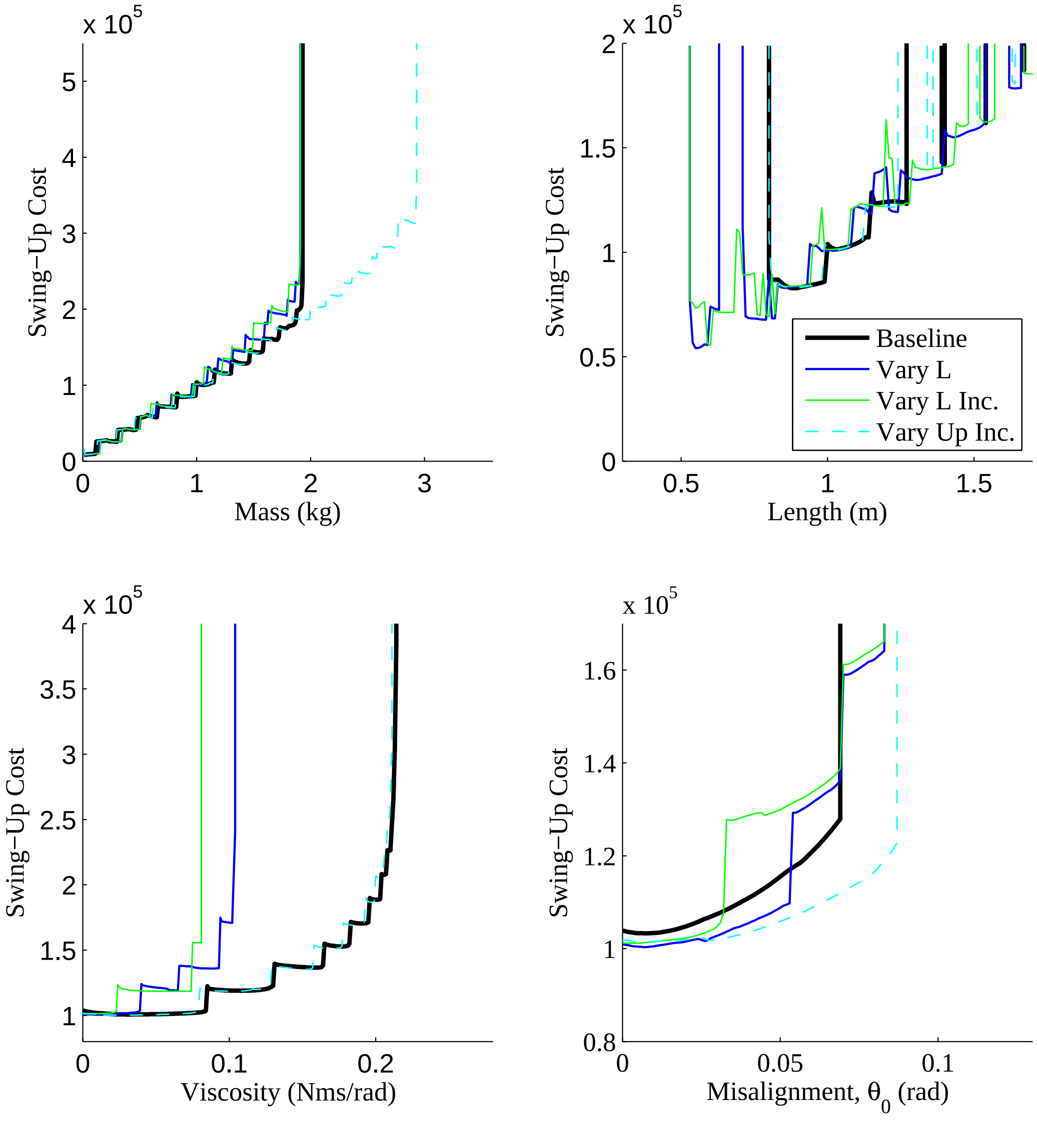

Figure 6.2: Robustness to model variations of mass, length, viscosity, and gravity misalignment. Results are shown for the baseline DP, HMMDP with models of various length added all at once and added incrementally, and HMMDP with models of various θ_0 added incrementally.

Given (6.15), (6.14) simplifies to

$$\Delta C = (1 + h(\mathbf{x})) \sum_{i=m}^{M} p(m) \Delta V_m(\mathbf{x}). \tag{6.16}$$

Given our update rule, we know that $\sum_{m=1}^{M} p(m)\Delta V_m(\mathbf{x}) \leq 0$, so we have $\Delta C \leq 0$.

In practice, we generally have a nominal (or best guess) dynamic system and wish to increase the range of model space surrounding it for which the policy performs well. Either because the convergence rate is very slow or because of nonidealities such as (6.15) not holding, MMDP sometimes finds solutions that work well for the extreme cases of model, but not for the nominal case. A less drastic problem is gaps in the region of model space for which the policy succeeds.

To cope with this, we add models incrementally. This technique is similar to shaping, which was first invented in the field of psychology for training animals [77], but has since been used for machine learning and optimization [54]. Shaping is used to train animals to do complex tasks by first training them to do something simple, then training them to do successively closer approximations of the desired behavior. We start off by running DP on just the nominal model. We then add dynamic models that are very similar to the nominal case, and run a new optimization starting with the previously solved for policy and the previously solved for value function. We can then continue to add additional models progressively further from the nominal dynamics and rerunning the optimization. When new models are added, the value function can be initialized by copying the value function of the most similar model already in the training set.

Figure 6.2 shows results for HMMDP. We took 21 pendulum models ($p(m) = 1/M$) with lengths, L, ranging from 0.5 m to 1.5 m in 0.05 m increments and ran HMMDP on them. The result was a drastically improved robustness to variation in the length. Swing-up was achievable for nearly the entire 0.5 m to 1.5 m range, but with a gap centered around 0.65 m.

We can compare this to the result when we add the same 21 models incrementally. We start with the nominal model ($L = 1.0$ m). After doing 5000 iterations of DP, we add the two adjacent models ($L = 0.95$ m and $L = 1.05$ m), initializing their value functions from the nominal models and then do another 5000 iterations of DP. We then add another 9 pairs of models, initializing each from the previous pair, and doing 5000 iterations of DP between each addition. By shaping the policy in this way and gradually building its robustness, we are able to cover the same range of modeling error, but without the gap centered at $L = 0.65$ m.

We also use the same method to target improving robustness to the gravitational angle by incrementally adding pairs of models with positive and negative θ_0's further and further from 0. The result is an improved robustness to varying this angle while maintaining a low cost. On the other hand, optimizing on this set of models does not improve performance with respect to changing the pendulum length. This illustrates a downside of MMDP: it often does not provide improved robustness for model variations other than the type specifically targeted.

Because (6.15) will generally not hold, in practice HMMDP is a heuristic method. The more qualitatively similar the models, the closer (6.15) will be to holding, and the closer HMMDP will get to truly minimizing (6.10).

6.6 Locally Optimal Multiple Model Controller Synthesis in Acyclic MDPs

The goal of Locally Optimal Multiple Model Dynamic Programming (LOMMDP) is to find the stationary stateless policy, $u = \pi(x)$, that minimizes the expected trajectory cost from (a weighted function of) all start states and a discrete set of M deterministic models. Formally, we wish to minimize the total cost criterion

$$C = \mathop{E}_{m=1}^{M} \left[\sum_{x_0} g_m(x) \sum_{t=0}^{\infty} L_m(x_t, \pi(x_t)) \right], \tag{6.17}$$

where g is a (possibly model dependent) function defining the relative importance of potential start states and L is a (possibly model dependent) one-step cost function that depends on both state and action. The innermost sum gives the cost of an infinite time horizon trajectory, and the outer sum gives the total cost for all potential start states.

Unfortunately, it has been shown that finding a globally optimal solution to this problem is NP-hard [10]. It was shown that this problem is equivalent to 3SAT by constructing a model to represent each 3SAT clause and representing the assignment of boolean states in 3SAT with a policy in our problem. Given this equivalence, if we could solve our problem, we could solve the equivalent 3SAT problem, meaning that our problem must be NP-hard. We will therefore have to be content with finding a locally optimal solution.

The added complexity of MMDP (as compared to single-model DP) arises because each model can have its own state distribution, which is policy dependent. In single-model DP, we do not need to worry about how important each state is because reducing the value at one state does not conflict with reducing the value at any other. However, in MMDP, a policy change may be good for some models and bad for others, making the relative state distribution of each model important.

6.6.1 Inputs

The LOMMDP algorithm takes as input a set of M dynamics functions, $x_{t+1} = f_m(x_t, u_t)$, associated probabilities $p(m)$, a set of M (possibly identical) one-step cost functions, $c = L_m(x, u)$, a set of M (possibly identical) importance functions, $g_m(x)$, and an initial policy, $\pi_0(x)$. The dynamics are discrete time and deterministic. Unfortunately, we must restrict the dynamics by requiring that they can not form loops (i.e. $x_{t_1} \neq x_{t_2} \forall t_1 \neq t_2$). Equivalently, we require that a trajectory starting from any initial condition must eventually terminate in a sink state. In practice, it may be possible to ignore this restriction with little or no effect, but it is necessary to guarantee that the algorithm will converge to a locally optimal solution. If this constraint is ignored, the algorithm may not converge; however, if it does converge, it will still do so to a locally optimal solution.

The importance functions, $g_m(x)$, specify weightings for each starting state. This can be interpreted as a distribution of initial states, in which case $\Sigma_x g(x) = 1$, but it can more generally be seen as part of the cost function, specifying which regions of state space are most important. The importance function can be model dependent ($g_i(x) \neq g_j(x)$), but need not be. Similarly, the one-step cost function, L, can be model dependent but need not be. We expect that for most applications it will make sense for g and L to be model independent ($g_i(x) = g_j(x), L_i(x,u) = L_j(x,u) \forall i,j$). Any initial policy, $\pi_0(x)$, can be used, though an initial policy nearer to the optimal policy likely make the algorithm converge in fewer steps as well as making it more likely to find the globally optimal solution.

6.6.2 Bookkeeping

LOMMDP is a variant of policy iteration that uses additional bookkeeping. We maintain a single policy, $\pi(x)$, but a separate value function for each model, $V_m(x)$. Additionally, we maintain a "density function", $h_m(x)$, for each model as well as well as an ancestor list, $H_m(x)$. The density function, $h_m(x)$, is a weighted (by $g_m(x)$) count of how many trajectories pass through x given that model m is correct. We define $\Psi_m(x)$ as the set of all states from which a trajectory will eventually pass through x, given m and π. The density function is then defined by

$$h_m(x) = \sum_{x_i \in \Psi_m(x)} g_m(x_i). \tag{6.18}$$

If $g_m(x)$ is interpreted as a probability distribution of the initial state, then $h_m(x)$ is the probability that the system will at some point pass through x given model m and the current policy, π. The ancestor list, $H_m(x)$, is simply a means of recording the inverse dynamics and will be helpful for efficiently updating the value function in response to an individual policy change. For every state, we maintain a list of other states whose dynamics lead to the state in question (given m and the current policy). Each state only leads to one other state, so each state can only be recorded in one other state's ancestor list. Therefore, the total storage necessary for the ancestor lists should be proportional to NM (the same as for value functions and density functions), where N is the number of states and M is the number of models.

6.6.3 Initialization

We initialize the policy with $\pi_o(x)$, then we must initialize the bookkeeping for each model independently. For each state, add x to the appropriate ancestor list, $H_m(f(x, \pi(x)))$. If we also have a list of sink states, we can then initialize the value function, $V_m(x)$, with no redundant computation. Starting from the sink states, we can use $H_m(x)$ to search backwards through the dynamics, filling in $V_m(x)$ with the Bellman equation, $V_m(x) = L_m(x, \pi(x)) + V_m(f_m(x, \pi(x)))$. To initialize the density function, $h_m(x)$, we recognize that it should equal the state's own importance plus the sum of the density of its ancestors, or

$$h_m(x) = g_m(x) + \sum_{x_{t-1} \in H_m(x)} h_m(x_{t-1}), \tag{6.19}$$

which can be solved recursively, again starting from the sink states. Alternatively, both $V_m(x)$ and $h_m(x)$ can be computed iteratively. The iterative method for computing $h_m(x)$ is closely related to the Forward Algorithm commonly used in Hidden Markov Model and Conditional Random Field inference [84]. We use deterministic dynamics and have no observations, which is a simplified special case of the common Forward Algorithm. The difference is that we are looking for a time-independent version of the probability of passing through a given state: essentially the cumulative sum of the chance of passing through on each time step.

6.6.4 Iterations

In each iteration, we must cycle through every state and do a policy update, then update the bookkeeping accordingly. The policy update step must consider all of the models and looks like

$$\pi(x) = \arg\min_u + \sum_{m=1}^{M} p(m) h_m(x)(L_m(x, u) + V_m(f_m(x, u))). \tag{6.20}$$

If the new $\pi(x)$ is different from the old $\pi(x)$, then we must update the V's, h's, and H's to match the updated policy. The H update is trivial. For each model, simply remove x from $H(f(x, \pi(x)_{\text{old}}))$ and add it to $H(f(x, \pi(x)_{\text{new}}))$. To update h, for each model, we must follow the forward dynamics from x to the eventual sink state using both the old and new action. We must subtract $h_m(x)$ from each $h_m(x_{\text{future}})$ along the trajectory using the old action and add $h_m(x)$ to each $h_m(x_{\text{future}})$ along the trajectory using the new action. Finally, we use the H_m's to efficiently locally update the V_m's. The new values at x for each model are given by

$$V_m(x) = L_m(x, u) + V_m(f_m(x, u)). \tag{6.21}$$

We can then use H_m to search up the ancestor tree of x to find every state in $\Psi_m(x)$ and updating the value by subtracting the change in value at x:

$$V_m(x_a)_{\text{new}} = V_m(x_a)_{\text{old}} - (V_m(x)_{\text{old}} - V_m(x)_{\text{new}}) \quad \forall x_a \in \Psi_m(x) \tag{6.22}$$

This does require us to update the value function after every policy update rather than at the end of the iteration, but keeping track of the affected values with H allows us to do so efficiently. In the later sweeps, when there are few policy updates per iteration, this may be faster than recomputing the entire value at the end of the iteration. The algorithm terminates when no policy changes are made in an entire iteration.

6.6.5 Proof of Local Optimality

Theorem 1. *LOMMDP is locally optimal when applied to an acyclic MDP in the sense that no single policy change can result in further improvement.*

If we take the total cost criterion given by (6.17) and substitute in the definition of the value function as the total cost to go, we get

$$C = \underset{m=1}{\overset{M}{E}} \left[\sum_x g_m(x) V_m(x) \right]. \tag{6.23}$$

If we make a single policy change at state x, it will change each value function at x by some amount ΔV_m. Additionally, for each model, the value at every state in $\Psi(x)$ will also change by the same value ΔV_m. The effect on the total cost, C, from a single policy change at x is therefore given by

$$\Delta C = \underset{m=1}{\overset{M}{E}} \left[\sum_{x_i \in \Psi_m(x)} g_m(x_i) \Delta V_m \right]. \tag{6.24}$$

We substitute the definition of h_m given in (6.18) to rewrite this as

$$\Delta C = \underset{m=1}{\overset{M}{E}} \left[h_m(x) \Delta V_m \right]. \tag{6.25}$$

If we substitute the value update (6.21) into the policy update (6.20) and rewrite the sum as an expectation, the policy update becomes

$$\pi(x) = \arg\min_u + \underset{m=1}{\overset{M}{E}} \left[h_m(x) V_m(x) \right]. \tag{6.26}$$

We can now see that the policy update step minimizes the total cost criterion, C. Further, when the algorithm terminates, we know that no single policy change can further reduce C. Therefore, the policy produced by LOMMDP is locally optimal in the sense that no single policy change can reduce C as defined in (6.17).

6.6.6 Use in Practice

In practice, we will want to use LOMMDP for periodic problems that have continuous states and actions, so we modify it along the lines of the baseline DP algorithm described in Section 2.2. We represent the state space (a low-dimensional continuous space) with a grid and use multilinear interpolation to find $V(f(\mathbf{x}, \mathbf{u}))$. We handle continuous action spaces by only comparing the current best action to a single randomly selected new action for each state during an iteration. We also switch to modified policy iteration; rather than maintaining consistent $\pi(\mathbf{x})$, $V_m(\mathbf{x})$, and $h_m(\mathbf{x})$, we iteratively solve for all three simultaneously. When we remove the restriction on periodic dynamics, we must add a discount factor slightly less than 1, γ, to ensure that the value converges to a constant. Removing this restriction invalidates our proof of local optimality. However, using interpolation to find the final value in each step drastically reduces the problem. The problem arises in the first place because if a state is downstream of itself, the value will depend on itself. However, with interpolation, at the end of the loop the value will not depend entirely on itself; instead the dependence will have spread out to a significant region of neighbors. Therefore, we can generally expect the effect to be small and the algorithm to be nearly locally optimal.

The effect of switching to continuous states will depend on the accuracy of determining the value by linearly interpolating points on the value function grid, which will depend on the resolution of the grid. Trading off this concern with computation time (and storage space) will generally be what drives selection of the grid resolution. The switch to continuous actions, and the associated change to considering only two potential actions during the policy update, means that the algorithm will never quite converge because it will take infinite time to find the exact best action. The convergence properties of using random actions are discussed in [5]. The switch to modified policy iteration will also negate the guarantee of convergence and local optimality. However, we expect the effect to be small so long as no single policy change has an enormous effect on either V or h, which is generally true for problems based on smooth continuous dynamics such as those we are interested in.

In this version, we can initialize with any policy and $V_m(\mathbf{x}) = 0$. We initialize the density to $h_m(\mathbf{x}) = g_m(\mathbf{x})$ at the beginning of every iteration. The policy update looks like the original LOMMDP policy update, but with only comparing two actions:

$$\pi(\mathbf{x}) = \arg \min_{\mathbf{u} \in \{\mathbf{u}_0, \mathbf{u}_r\}} + \sum_{m=1}^{M} p(m) h_m(\mathbf{x}) (L_m(\mathbf{x}, \mathbf{u}) + \gamma V_m(f_m(\mathbf{x}, \mathbf{u}))), \tag{6.27}$$

where $\mathbf{u}_0$ is the current $\pi(\mathbf{x})$ and $\mathbf{u}_r$ is a randomly selected action. For each model, we then do a value update,

$$V_m(\mathbf{x}) = L_m(\mathbf{x}, \mathbf{u}) + \gamma V_m(f_m(\mathbf{x}, \mathbf{u})), \tag{6.28}$$

and a density update,

$$h_m(f_m(\mathbf{x}, \pi(\mathbf{x}))^t = h_m(f_m(\mathbf{x}, \pi(\mathbf{x}))^t + \gamma h_m(\mathbf{x})^{t-1} \tag{6.29}$$

Due to stochastically searching a continuous action space, this algorithm will never quite finish, so we terminate it when either there are few policy changes per iteration or the total value change is small per iteration.

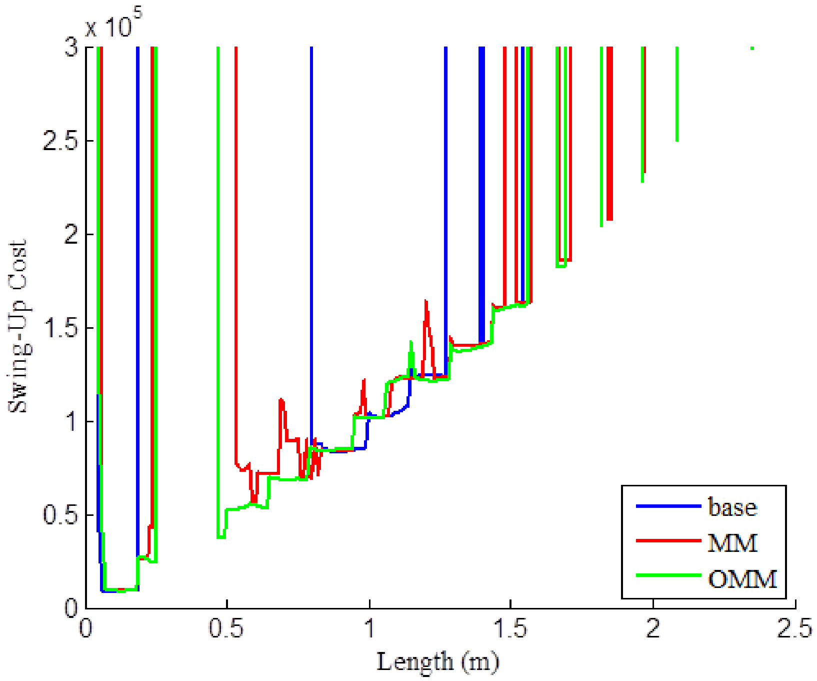

Figure 6.3: Comparison of the baseline DP algorithm (base), the heuristic MMDP algorithm (MM), and the LOMMDP algorithm (OMM) for the pendulum swing up problem. The baseline algorithm is computed with a pendulum length of 1.0 m, and the two MMDP algorithms use a set of 21 models that have pendulum lengths varying (with even spacing) from 0.5 m to 1.5 m. This plot shows the actual swing-up cost as a function of the test model pendulum length.

This version (LOMMDP) performs slightly better and more predictably than the heuristic version in Section 6.5. We test both MMDP algorithms on the pendulum swing-up problem. We run the two algorithms on a set of 21 training models with pendulum lengths spaced from 0.5 m to 1.5 m in even 0.05 m increments. Then, we test the generated policy on a larger set of test models that also have varied pendulum length, but cover a wider range and vary in much smaller increments. Figure 6.3 shows the swing-up cost as a function of the test model's pendulum length for both algorithms. Unlike the heuristic version, which has some gaps near the edges.

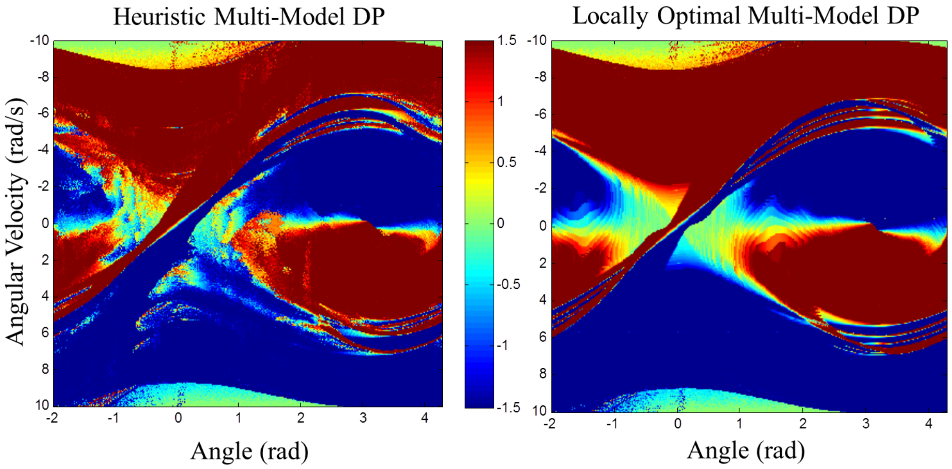

Figure 6.4: Comparison of the policy produced by the HMMDP and LOMMDP algorithms for the pendulum swing up problem.

The locally optimal version is able to stabilize the entire training range. It also has fewer spikes in the value curve. Figure 6.4 compares the two policies. They have the same general shape, but the locally optimal version produces a smoother policy with less noise.

Figure 6.5 shows the convergence of the two algorithms. Note that the plot is not showing value, but the number of test models that are stabilized, which is a measure of the size of the region of model space that is stabilized. Both algorithms take about the same number of sweeps to achieve most of the final value. The Locally Optimal algorithm stabilizes a larger region than the Heuristic algorithm, and perhaps more importantly, it fully converges, whereas the Heuristic algorithm continues to vary up and down without converging.

We applied MMDP to both simulation and the real robot. In simulation, we used MMDP to generate the sagittal stance policy using three different models, having -3 cm 0 cm, and 3 cm offsets to the touchdown location. The effect depended on the timing of the push, but averaging over all push timings, we found a 10% increase in the size of the survivable push as compared to the baseline DP policy. For the Sarcos robot controller, we applied MMDP to the coronal policy using three models that had a -2 cm, 0 cm, and 2 cm offset to the CoP location. The MMDP

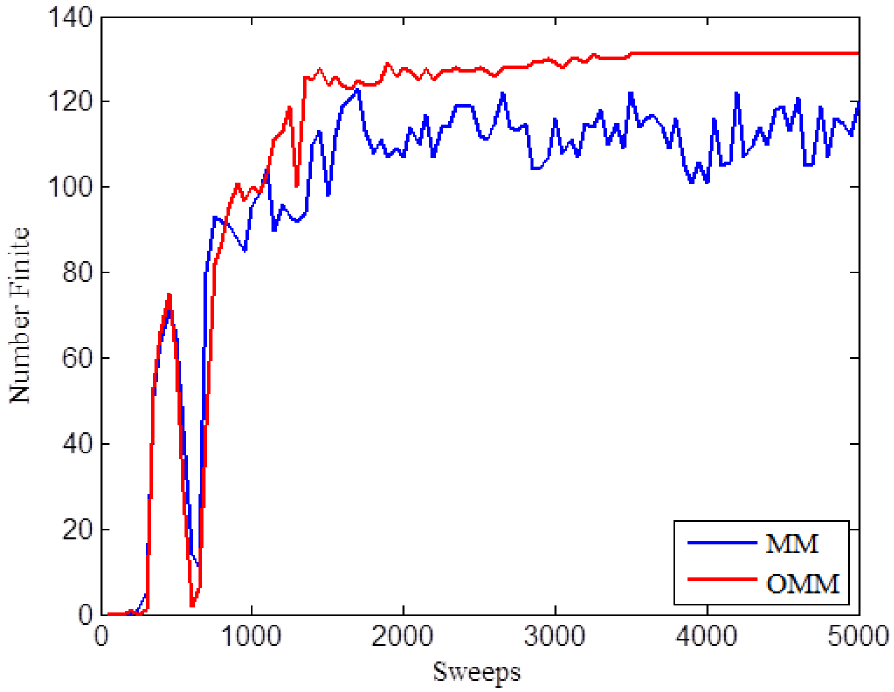

Figure 6.5: How large a region of model space is stabilized as a function of sweeps for Heuristic MMDP (MM) and Locally Optimal MMDP (OMM). "Sweeps" refers to the number of iterations of updating every state. "Number Finite" refers to the number of test models that are stabilized by the policy (achieve swing-up with finite cost). Both algorithms compute their policy with a set of 21 models that have pendulum lengths varying (with even spacing) from 0.5 m to 1.5 m. The large drop at 700 (and the smaller one at 1400) sweeps likely occurs when the policy switches the number of swings used to achieve swing-up.

policy performed comparably to the baseline DP policy. A more thorough attempt at applying MMDP to the walking controller may be a useful area of future research.

6.7 Discussion of MMDP

Both version of MMDP work well when you know something about the type of modeling error you have. It works best when you are able to provide it with models that span the range of possible systems. Attempts to generally increase robustness by providing models with multiple types of changes performed less well. This was at least partially due to the fact that in the higher dimensional model space, our sampling was much more sparse.

Another limitation of the MMDP approach is that it is computationally expensive to handle types of error that increase the dimensionality of the state space (e.g. time delays, filters, structural flexibility) even if the policy does not depend on these extra dimensions. MMDP must include these extra states in the grid and therefore pay the exponential computational and memory cost of the extra states. Methods that do not explicitly model the modeling error (such as the noise and minimax formulations) need not pay this cost. Methods that simulate entire trajectories (instead of single time steps as in DP) can avoid it as well because they need only fully explore the space that the controller depends on.

For a known, deterministic system, the optimal controller for any cost function where failure has an infinite cost will have the same basin of attraction, which is identical to the maximum feasible region. However, this says nothing about what happens when the system is different from what was expected. Robustness to modeling error can be very sensitive to the choice of cost function. Experiments support our intuition that in MMDP, the robustness to the type of model variation is relatively insensitive to choice of cost function because robustness is being directly optimized for. However, for types of modeling error other than what we were directly optimizing for, we see just as much sensitivity to cost function choice as we see in the stochastic and minimax formulations.

The pendulum swing-up problem is an easy task, so even the baseline control algorithm achieves a moderate level of robustness to modeling error. In this case, the robust algorithm provides extreme robustness, which may seem unnecessary or excessive. In some cases, such as contact (e.g. shaking hands) or walking in sand or surf at the beach, this extreme level of robustness may be necessary. Additionally, for harder problems, the baseline algorithm may provide minimal robustness to modeling error, so the additional robustness provided by the robust versions of the algorithm will be necessary for controlling a real system.

As mentioned at the end of Section 6.3, there is a performance cost to achieving robustness. The optimal deterministic controller achieves the lowest possible cost if the system does follow the expected deterministic dynamics. Any other controller will therefore result in a higher cost, so if the robust algorithm generates a more robust controller, it must necessarily be different, and must necessarily have a higher cost when applied to the nominal plant. For example, note that in Figure 6.3 the baseline algorithm has a lower value near the nominal model even though it stabilizes a smaller region of model space. A downside of MMDP is that it does not necessarily provide benefits for model variations other than the type specifically targeted. For example, while the range of gravity misalignment angles that can be handled increases, the amount of viscosity that can be tolerated actually decreases.

The HMMDP algorithm reduces to the baseline DP implementation when the number of models, M, is one, and has computational cost per iteration proportional to M. Additionally, it may take more iterations for the HMMDP algorithm to converge than the baseline algorithm, and if we add models incrementally, that further increases the computation time. It has a storage cost proportional to $d + M$, where d is the dimensionality of the continuous action space. The LOMMDP algorithm has a similar computation cost per iteration to the HMMDP algorithm since the density update can be efficiently done concurrently with the value update. It does, however, have a reduced need to add models incrementally. Its storage cost is proportional to $d + 2M$ because of the need to store the h_m functions in addition to the V_m functions.

The problem we are solving here is a special case of that solved by Iterated Policy Search by Dynamic Programming (IPSDP) in [10]. IPSDP differs in that it finds non-stationary policies and fully optimizes the policy between updates of the state distribution (analogous to a non-stationary version of h_m in LOMMDP). Non-stationary policies have the potential to be better than the best stationary policy, but can be inconvenient in practice. They require vastly more storage space (they store a policy for every time step), which would be limiting for our intended applications. Additionally, it is not always obvious how to use a non-stationary policy. There may not naturally be any notion of a clock or an initial time step. Non-stationary policies may be more sensitive to types of modeling error that make the dynamics evolve at an unexpected rate, especially when modified to be used in a continuous time context.

A major advantage of LOMMDP is that it produces a locally optimal stationary policy, which is easier to use than a non-stationary policy. The two algorithms also have a different type of local optimality. LOMMDP is locally optimal in the sense that no single policy change can be an improvement; IPSDP is locally optimal in the sense that the policy is globally optimal given the estimate of the state distribution. Unfortunately, LOMMDP requires acyclic dynamics to guarantee convergence.

We test both IPSDP and LOMMDP on the simple problem of pendulum swing-up. We wish to investigate convergence, and an algorithm that randomly samples actions from a continuous action space will never converge, so we use a version of the pendulum dynamics with seven discrete action choices $u \in \{-1.5, -1.0, -0.5, 0.0, 0.5, 1.0, 1.5\}$ (but still using a continuous state space). As expected, single-model DP as described in Chapter 2 converges in finite time when using discrete actions. However, due to approximations, neither IPSDP nor LOMMDP converged (in reasonable time) when used on a three-model problem consisting of three pendulums with lengths of 0.8, 1.0, and 1.2 m.

In the case of IPSDP, the only approximation is the use of continuous state spaces and the associated interpolation, and it very nearly converged reasonably quickly. After a few hundred iterations of optimizing the entire non-stationary policy, then finding a new state distribution, only about 5-10 grid points (of the 90,000 in a 200 by 450 grid) in π_0, the policy for the first time step, changed per iteration. However, this level of slight non-convergence remained constant for many iterations with no sign of further change. The policy it generates is extremely non-stationary, by which we mean that the policy varies considerably between consecutive time steps. Specifically, the ripples shown in Figure 6.7 flow towards the origin along the path of the dynamics.

In the case of LOMMDP, we also have the approximation due to continuous state spaces, but there are other additional approximations as well. The pendulum dynamics are acyclic, though we expect this effect to be at least somewhat smoothed by interpolation. We also use a version that is a compromise between the modified policy iteration we normally use in practice and the version discussed in Sections 6.6.1 through 6.6.6. We fully update the value and density between every iteration, but not after every single grid point policy change. The result of these approximations is that it never converges past the point of several thousand policy changes (from that same 90,000 point grid) in each iteration.

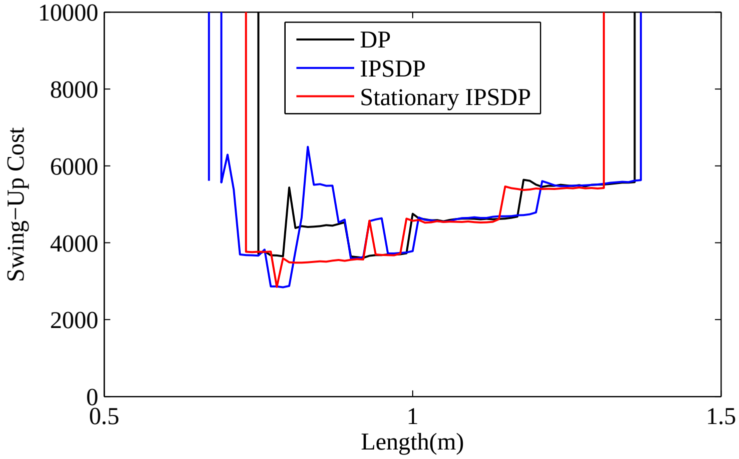

Figure 6.6: The cost of swing up as a function of pendulum length for IPSDP and DP computed with three pendulums with lengths 0.8, 1.0, and 1.2 m. Stationary IPSDP refers to using the first time step of the non-stationary IPSDP policy as a stationary policy.

Figure 6.6 shows results for both algorithms on the three-model pendulum swing-up case. IPSDP stabilizes a larger region of model space than LOMMDP does. However, if rather than using the full non-stationary IPSDP policy, we use the first time step as a stationary version, it does worse than LOMMDP. Figure 6.7 shows the policies produced by LOMMDP and IPSDP. Both algorithms produce policies that contain ripples (i.e. bands of alternating commands), but they are more pronounced in the policy produced by IPSDP. On real hardware, such ripples would likely excite higher-order unmodeled dynamics, but they are not particularly problematic in simulation. IPSDP is a very powerful, but generic, algorithm. LOMMDP has advantages (e.g. stationary policies) for the types of problems we are interested in, but has its own problems (e.g. requires cyclic dynamics to guarantee convergence). Further experiments would be necessary to get a good feel for when either algorithm is likely to outperform the other in practice.

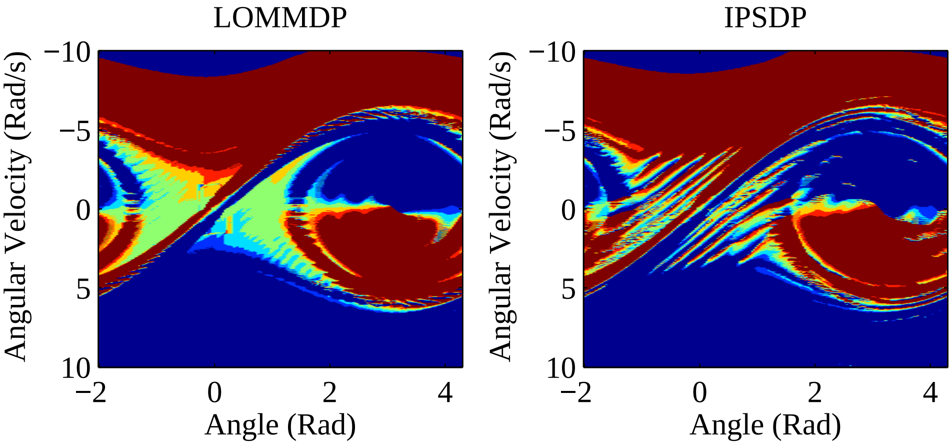

Figure 6.7: The policies produced by LOMMDP (left) and IPSDP (right). Red represents the maximum command (1.5 Nm), and blue represents the minimum command(-1.5 Nm).

Chapter 7

Future Work

Our simulation controller is severely limited by the topology of our chosen simulator. The most important improvement to make in future work is to add true feet to the simulation (ours simulates feet by allowing the shank to apply limited torque directly to the ground) and adding toe off and heel strike to the walking gait. Preliminary attempts at upgrading the simulation to use true feet required either prohibitively slow simulation speeds or vastly increased rotational inertia of the feet. Development of a simulator that does not have these limitations will be the first hurdle. While our control framework can handle toe off and heel strike, we anticipate that there will be considerable engineering challenges during implementation. Another worthwhile improvement would be the addition of arms. Controlled arm swing would both make the gait look more human-like and reduce yaw moment on the foot, which has been somewhat limiting on the robot.

Future work on walking with the Sarcos robot should focus on walking faster and more robustly. We hope that the upgrades that the Sarcos robot is currently undergoing, especially strengthening the right hip adduct/abduct actuator will ease some of the difficulties we encountered. State estimation and dynamic modeling are areas of our control system that would particularly benefit from further research.

A more comprehensive attempt to use Multiple Model Dynamic Programming or other forms of robust DP might also be a promising area of endeavor. If the nonidealities of the real robot can not be eliminated or compensated for (by improved state estimation, dynamic modeling, etc.), then robust control techniques may be a good way to succeed despite them.

Bibliography

[1] Yeuhi Abe, C Karen Liu, and Z Popovic. Momentum-based Parameterization of Dynamic Character Motion. *ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, pages 194–211, 2004. 4

[2] K. Akachi, K. Kaneko, N. Kanehira, S. Ota, G. Miyamori, M. Hirata, S. Kajita, and F. Kanehiro. Development of humanoid robot HRP-3P. In *Humanoid Robots, 2005 5th IEEE-RAS International Conference on*, pages 50–55, 2005. 1.2

[3] Brian Anderson and J Moore. Time-varying feedback laws for decentralized control. *Automatic Control, IEEE Transactions on*, 26(5):1133–1139, 1981. 3

[4] Yaman Arkun and George Stephanopoulos. Studies in the synthesis of control structures for chemical processes: Part IV. design of steady-state optimizing control structures for chemical process units. *AIChE Journal*, 26:975–991, 1980. 6

[5] Christopher G. Atkeson. Randomly sampling actions in dynamic programming. In *Proceedings of the 2007 IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning*, 2007. 2.1, 2.2, 2.2, 6.6.6

[6] Christopher G. Atkeson and Jun Morimoto. Nonparametric representation of policies and value functions: A trajectory-based approach. In *NIPS*, pages 1611–1618, 2003. 2.1

[7] Christopher G. Atkeson and Benjamin J. Stephens. Random sampling of states in dynamic programming. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 38(4):924–929, 2008. 2.1

[8] V. Azhmyakov, V.G. Boltyanski, and A.S. Poznyak. On the dynamic programming approach to multi-model robust optimal control problems. *American Control Conference*, pages 4468–4473, 2008. 6

[9] D. Bagnell and J. Schneider. Autonomous helicopter control using reinforcement learning policy search methods. In *IEEE International Conference on Robotics and Automation*, 2001. 6

[10] J. Andrew (Drew) Bagnell. *Learning Decisions: Robustness, Uncertainty, and Approximation.* PhD thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, August 2004. 6.6, 6.7

[11] F. J. Bejarano, A. Poznyak, and L. Fridman. Min-max output integral sliding mode control for multiplant linear uncertain systems. In *American Control Conference*, pages 5875–5880, 2007. 6

[12] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957. 1.3, 2.1, 2.1, 6

[13] Richard Bellman. A markovian decision process. *Indiana Univ. Math. J.*, 6(4):679–684, 1957. 2.1

[14] Alberto Bemporad and Manfred Morari. Robust model predictive control: A survey. In *Robustness in identification and control*, volume 245 of *Lecture Notes in Control and Information Sciences*, pages 207–226. Springer Berlin / Heidelberg, 1999. 6

[15] Abder Rezak Benaskeur and Andr Desbiens. Decentralized control: a modified lyapunov function scheme. In *American Control Conference, 1999. Proceedings of the 1999*, volume 3, pages 2087–2091. IEEE, 1999. 3

[16] Darrin C. Bentivegna, Christopher G. Atkeson, and Jung-Yup Kim. Compliant control of a compliant humanoid joint. In *Proceedings of the IEEE-RAS International Conference on Humanoid Robots*, 2007. 1.2, 4.2, 5.2

[17] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, 1995. ISBN 1886529132. 2.1, 6

[18] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm intelligence: from natural to artificial systems*, volume 4. Oxford university press New York, 1999. 3

[19] John Camp. Powered "passive" dynamic walking. Master's thesis, Cornell University, August 1997. 1.2

[20] Steven H. Collins, Martijn Wisse, and Andy Ruina. A three-dimensional passive-dynamic walking robot with two legs and knees. *I. J. Robotic Res.*, 20(7):607–615, 2001. 1.2

[21] Scott Davies. Multidimensional triangulation and interpolation for reinforcement learning. In *NIPS*, pages 1005–1011, 1996. 2.2

[22] EJ Davison and TN Chang. Decentralized stabilization and pole assignment for general proper systems. *Automatic Control, IEEE Transactions on*, 35(6):652–664, 1990. 3

[23] Holger Diedam, Dimitar Dimitrov, Pierre-Brice Wieber, Katja Mombaur, and Moritz Diehl. Online walking gait generation with adaptive foot positioning through linear model predictive control. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2008. 2.1, 4

[24] Moritz Diehl and Jakob Bjornberg. Robust dynamic programming for min-max model predictive control of constrained uncertain systems. *Automatic Control, IEEE Transactions on*, 49(12):2253–2257, dec. 2004. 6

[25] Geir E. Dullerud and Fernando Paganini. *A course in robust control theory: A convex approach*. Texts in Applied Mathematics. Springer, New York, NY, 2000. 6

[26] Tom Erez, Yuval Tassa, and Emanuel Todorov. Infinite horizon model predictive control for nonlinear periodic tasks. *Manuscript under review*, 2011. 4

[27] E. Frazzoli, M. A. Dahleh, and E. Feron. Maneuver-based motion planning for nonlinear systems with symmetries. *IEEE Trans. on Robotics*, 21(6):1077–1091, December 2005. 2.1

[28] Y. Fujimoto and A. Kawamura. Proposal of biped walking control based on robust hybrid position/force control. In *Proceedings of IEEE International Conference on Robotics and Automation*, volume 3, pages 2724–2730. IEEE, 1996. 4

[29] R.J. Full and D.E. Koditschek. Templates and anchors: Neuromechanical hypotheses of legged locomotion on land. In *The Journal of Experimental Biology*, volume 202, pages 3325–3332, 1999. 1.3

[30] Luca Di Gaspero. Quadprog++. 4.2.3

[31] Zhiming Gong and Mohammad Aldeen. Stabilization of decentralized control systems. *Journal of Mathematical Systems Estimation and Control*, 7:111–114, 1997. 3

[32] A Goswami and V Kallem. Rate of change of angular momentum and balance maintenance of biped robots. In *Proceedings of the 2004 IEEE International Conference on Robotics and Automation*, volume 4, pages 3785–3790, Honda Res. Inst., Mountain View, CA, USA, 2004. 4

[33] Ambarish Goswami, Bernard Espiau, and Ahmed Keramane. Limit cycles in a passive compass gaitbiped and passivity-mimicking control laws. *Auton. Robots*, 4(3):273–286, July 1997. 1.3

[34] Jesse A. Grimes and Jonathan W. Hurst. The design of atrias 1.0 a unique monopod, hopping robot. In *Proceedings of the 2012 International Conference on Climbing and walking Robots and the Support Technologies for Mobile Machines*, pages 548–554, 2012. 1.2

[35] Hugh Herr and Marko Popovic. Angular momentum in human walking. *Journal of Experimental Biology*, 211(4):467–481, February 2008. 4.6

[36] K. Hirai, M. Hirose, Y. Haikawa, and T. Takenaka. The development of honda humanoid robot. *IEEE International Conference on Robotics and Automation Proceedings*, 2:1321–1326, 1998. 1.2

[37] Daan G. E. Hobbelen and Martijn Wisse. A disturbance rejection measure for limit cycle walkers: The gait sensitivity norm. *IEEE Transactions on Robotics*, 23(6):1213–1224, 2007. 1.2

[38] A. Hofmann, M. Popovic, and H. Herr. Exploiting angular momentum to enhance bipedal center-of-mass control. In *2009 IEEE International Conference on Robotics and Automation*, pages 4423–4429. IEEE, May 2009. ISBN 978-1-4244-2788-8. doi: 10.1109/ROBOT.2009.5152573. 4

[39] R.A. Howard. *Dynamic programming and Markov processes*. Technology Press of Massachusetts Institute of Technology, 1960. 2.1

[40] Qiang Huang, Kazuhito Yokoi, Shuuji Kajita, Kenji Kaneko, Hirohiko Arai, Noriho Koyachi, and Kazuo Tanie. Planning walking patterns for a biped robot. *IEEE Transactions on Robotics and Automation*, 17:280–289, 2001. 1.2

[41] Fumiya Iida and Russ Tedrake. Minimalistic control of a compass gait robot in rough terrain. In *Proceedings of the 2009 IEEE international conference on Robotics and Automation*, ICRA'09, pages 3246–3251. IEEE Press, 2009. 1.3

[42] Eng J. J., Winter D. A., and Patla A. E. Strategies for recovery from a trip in early and late swing during human walking. *Experimental Brain Research*, 102:339–349, 1994. 4.4.3

[43] D. H. Jacobson and D. Q. Mayne. *Differential Dynamic Prgogramming*. American Elsevier Pub. Co., New York, 1970. 2.1, 2.1

[44] S. Kajita and K. Tani. Study of dynamic biped locomotion on rugged terrain-derivation and application of the linear inverted pendulum mode. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 2, pages 1405–1411, April 1991. 4.1

[45] S. Kajita, F. Kanehiro, K. Kaneko, K. Fujiwara, K. Harada, K. Yokoi, and H. Hirukawa. Resolved momentum control: humanoid motion planning based on the linear and angular momentum. In *Intelligent Robots and Systems, 2003. (IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, volume 2, 2003. 4

[46] Shuuji Kajita, Fumio Kanehiro, Kenji Kaneko, Kazuhito Yokoi, and Hirohisa Hirukawa. The 3d Linear Inverted Pendulum Model: A simple modeling for biped walking pattern generation. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 240–246, November 2001. 4.1

[47] Shuuji Kajita, Fumio Kanehiro, Kenji Kaneko, Kiyoshi Fujiwara, Kensuke Harada, Kazuhito Yokoi, and Hirohisa Hirukawa. Biped walking pattern generation by using preview control of zero-moment point. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1620–1626, September 2003. 1.2, 1.3, 4

[48] K. Kaneko, F. Kanehiro, S. Kajita, K. Yokoyama, K. Akachi, T. Kawasaki, S. Ota, and T. Isozumi. Design of prototype humanoid robotics platform for HRP. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2431–2436, 2002. 1.2

[49] Kamal Kant and Steven W. Zucker. Toward efficient trajectory planning: The path-velocity decomposition. *The International Journal of Robotics Research*, 5(3):72–89, 1986. 3

[50] D. Kaynov, P. Soueres, P. Pierro, and C. Balaguer. A practical decoupled stabilizer for joint-position controlled humanoid robots. *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3392–3397, Oct. 2009. doi: 10.1109/IROS.2009.5354431. 4

[51] David Andrew Kendrick. *Stochastic control for economic models*. McGraw-Hill, second edition, 2002. 6

[52] Jung-Yup Kim, Christopher Atkeson, Jessica Hodgins, Darrin Bentivegna, and Sung Ju Cho. Online gain switching algorithm for joint position control of a hydraulic humanoid robot. In *Proceedings of the IEEE-RAS International Conference on Humanoid Robots*, 2007. 1.2, 4.2, 5.2

[53] Jelle R Kok, Pieter Jan't Hoen, Bram Bakker, and Nikos A Vlassis. Utile coordination: Learning interdependencies among cooperative agents. In *CIG*, 2005. 3

[54] George Konidaris and Andrew Barto. Autonomous shaping: knowledge transfer in reinforcement learning. In *Proceedings of the 23rd International Conference on Machine*

learning, ICML '06, pages 489–496, 2006. 6.5

[55] K Madhava Krishna, Henry Hexmoor, and Srinivas Chellappa. Reactive navigation of multiple moving agents by collaborative resolution of conflicts. *Journal of Robotic Systems*, 22(5):249–269, 2005. 3

[56] R.E. Larson. *State increment dynamic programming*. American Elsevier Pub. Co., 1968. 2.2

[57] Steven M. Lavalle. From dynamic programming to rrts: Algorithmic design of feasible trajectories. In *Control Problems in Robotics*. Springer-Verlag, 2002. 2.1

[58] Sung-hee Lee and Ambarish Goswami. Ground reaction force control at each foot : A momentum-based humanoid balance controller for non-level and non-stationary ground. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3157–3162, 2010. 4

[59] Stephane Leroy, Jean-Paul Laumond, and Thierry Siméon. Multiple path coordination for mobile robots: A geometric algorithm. In *IJCAI*, volume 99, pages 1118–1123, 1999. 3

[60] Z. Li, W. W. Melek, and C. Clark. Decentralized robust control of robot manipulators with harmonic drive transmission and application to modular and reconfigurable serial arms. *Robotica*, 27(2):291–302, March 2009. 1.2

[61] Chenggang Liu and Christopher G. Atkeson. Standing balance control using a trajectory library. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2009. 2.1, 4

[62] Schillings A. M., Van Wezel B.M. H., Mulder T. H., and Duysens J. Muscular responses and movement strategies during stumbling over obstacles. *Journal of Neurophysiology*, 83:2093–2102, 2000. 4.4.3

[63] Adriano Macchietto, Victor Zordan, and Christian R. Shelton. Momentum control for balance. *ACM Transactions on Graphics*, 28(3), 2009. ISSN 07300301. doi: 10.1145/1531326.1531386. 4

[64] Thijs Mandersloot, Martijn Wisse, and Christopher G. Atkeson. Controlling velocity in bipedal walking: A dynamic programming approach. In *Proceedings of the IEEE-RAS International Conference on Humanoid Robots*, 2006. 2.1

[65] Tad McGeer. Passive dynamic walking. *Int. J. Rob. Res.*, 9(2):62–82, mar 1990. 1.2

[66] M. McNaughton. CASTRO: robust nonlinear trajectory optimization using multiple models. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 177–182, 2007. 6

[67] Francisco S Melo and Manuela Veloso. Learning of coordination: Exploiting sparse interactions in multiagent systems. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 773–780. International Foundation for Autonomous Agents and Multiagent Systems, 2009. 3

[68] K. Mombaur, J.P. Laumond, and E. Yoshida. An optimal control model unifying holonomic and nonholonomic walking. In *Proceedings of the IEEE International Conference on Humanoid Robots*, Daejon, Korea, 2008. 2.1

[69] Jun Morimoto, Garth Zeiglin, and Christopher G. Atkeson. Minimax differential dynamic programming: Application to a biped walking robot. In *Proceedings of Intl. Conference on Intelligent Robots and Systems*, 2003. 6, 6.4

[70] Mitsuharu Morisawa, Kensuke Harada, Shuuji Kajita, Shinichiro Nakaoka, Kiyoshi Fujiwara, Fumio Kanehiro, Kenji Kaneko, and Hirohisa Hirukawa. Experimentation of humanoid walking allowing immediate modification of foot place based on analytical solution. In *Proceedings of IEEE International Conference on Robotics and Automation*, pages 3989–3994, October 2007. 4

[71] Remi Munos and Andrew Moore. Variable resolution discretization in optimal control. *Machine Learning*, 49, Numbers 2/3:291–323, 2002. 2.1

[72] J. A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, 1965. 4.2.2

[73] Arnab Nilim, Laurent El Ghaoui, and Vu Duong. Robust dynamic routing of aircraft under uncertainty. In *Proceedings of Digital Avionics Systems Conference*, 2002. 6

[74] Koichi Nishiwaki and Satoshi Kagami. High frequency walking pattern generation based on preview control of ZMP. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2667–2672, May 2006. 2.1, 4

[75] R. H. Nyström, J. M. Böling, J. M. Ramstedt, H. T. Toivonen, and K. E. Häggblom. Application of robust and multimodel control methods to an ill-conditioned distillation column. *Journal of Process Control*, 12:39–53, 2002. 6

[76] Ill-Woo Park, Jung-Yup Kim, Jungho Lee, and Jun-Ho Oh. Mechanical design of humanoid robot platform khr-3 (kaist humanoid robot-3: Hubo). In *Proc. IEEE/RAS Int. Conf. on Humanoid Robots*, pages 321–326, 2005. 1.2

[77] Gail B Peterson. A day of great illumination: B. F. Skinner's discovery of shaping. *Journal of the Experimental Analysis of Behavior*, 82:317–328, 2004. 6.5

[78] Y. Piguet, U. Holmberg, and R. Longchamp. A minimax approach for multi-objective controller design using multiple models. *International Journal of Control*, 72(7-8):716–726, 1999. 6

[79] G. Pratt and M. Williamson. Series elastic actuators. In *1995 IEEE/RSJ International Conference on Intelligent Robots and Systems. Human Robot Interaction and Cooperative Robots*, volume 1, pages 399–406, Los Alamitos, CA, USA, 1995. IEEE Comput. Soc. Press. 1.2

[80] Jerry Pratt, John Carff, Sergey Drakunov, and Ambarish Goswami. Capture Point: A Step toward Humanoid Push Recovery. In *Proceedings of the International Conference on Humanoid Robots*, pages 200–207. IEEE, Dec. 2006. ISBN 1-4244-0199-2. doi: 10.1109/ICHR.2006.321385. 4, 4.5.1

[81] Jerry E. Pratt. *Virtual model control of a biped walking robot*. PhD thesis, Massachusetts Institute of Technology, 1995. 1.3

[82] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley and Sons, New York, 1994. 2.1

[83] Martin L. Puterman and Moon Chirl Shin. Modified policy iteration algorithms for discounted markov decision problems. *Management Science*, 24(11):1127–1137, 1978. 2.1

[84] Lawrence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989. 6.6.3

[85] Carl Edward Rasmussen and Malte Kuss. Gaussian processes in reinforcement learning. In *NIPS*, 2003. 4.6.1

[86] John Rust. Using randomization to break the curse of dimensionality. *Econometrica*, pages 487–516, 1997. 2.1

[87] J. Shinar, V. Glizer, and V. Turetsky. Solution of a linear pursuit-evasion game with variable structure and uncertain dynamics. In S. Jorgensen, M. Quincampoix, and T. Vincent, editors, *Advances in Dynamic Game Theory - Numerical Methods, Algorithms, and Applications to Ecology and Economics*, volume 9, pages 195–222. Birkhauser, 2007. 6

[88] Jennie Si, Andrew G. Barto, Warren B. Powell, and Don Wunsch, editors. *Handbook of Learning and Approximate Dynamic Programming*. Wiley-IEEE Press, 2004. 2.1, 6

[89] Dragoslav D Siljak. *Decentralized control of complex systems*. DoverPublications. com, 2012. 3

[90] William M Spears, Diana F Spears, Jerry C Hamann, and Rodney Heil. Distributed, physics-based control of swarms of vehicles. *Autonomous Robots*, 17(2-3):137–162, 2004. 3

[91] Marc C. Steinbach. Robust process control by dynamic stochastic programming. *Proceedings in Applied Mathematics and Mechanics (PAMM)*, 4(1):11–14, 2004. 6

[92] Benjamin Stephens. Humanoid push recovery. In *Proceedings of the IEEE-RAS International Conference on Humanoid Robots*, 2007. 4

[93] Benjamin Stephens. *Push REcovery Control for Force-Controlled Humanoid Robots*. PhD thesis, CMU, 2011. 4.2.3, 5.4.2

[94] Benjamin J. Stephens. Dynamic balance force control for compliant humanoid robots. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2010. Available online at www.cs.cmu.edu/˜bstephe1/papers/iros10.pdf. 4.2.3

[95] Mike Stilman, Christopher G. Atkeson, James J. Kuffner, and Garth Zeglin. Dynamic programming in reduced dimensional spaces: Dynamic planning for robust biped locomotion. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2399–2404, 2005. 2.1

[96] H. T. Su and T. Samad. Neuro-control design: Optimization aspects. In O. Omidvar and D. L. Elliott, editors, *Neural Systems For Control*, pages 259–288. Academic Press, 1997. 6

[97] A. Takanishi, T. Takeya, H. Karaki, and I. Kato. A control method for dynamic biped walking under unknown external force. In *IEEE International Workshop on Intelligent Robots and Systems, Towards a New Frontier of Applications*, volume 29, pages 795–801. IEEE, 1990. 1.2, 1.3, 4

[98] Toru Takenaka, Takashi Matsumoto, Takahide Yoshiike, Tadaaki Hasegawa, Shinya Shirokura, Hiroyuki Kaneko, and Atsuo Orita. Real time motion generation and control for biped robot -4th report: Integrated balance control. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1601–1608. IEEE, Oct. 2009. ISBN 978-1-4244-3803-7. doi: 10.1109/IROS.2009.5354522. 4

[99] Russ Tedrake, Ian R. Manchester, Mark Tobenkin, and John W. Roberts. Lqr-trees: Feedback motion planning via sums-of-squares verification. *Int. J. Rob. Res.*, 29(8):1038–1052, July 2010. 2.1

[100] Evangelos Theodorou, Yuval Tassa, and Emo Todorov. Stochastic differential dynamic programming. In *Proceedings of American Control Conference*, 2010. 6

[101] Emanuel Todorov and Yuval Tassa. Iterative local dynamic programming. In *IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, pages 90–95, 2009. 2.1

[102] Michiel van de Panne, Eugene Fiume, and Zvonko G. Vranesic. A controller for the dynamic walk of a biped across variable terrain. In *Proceedings of the 31st Conference on Decision and Control*, pages pp. 2668–2673, December 1992. 2.1, 3

[103] A. Varga. Optimal output feedback control: a multi-model approach. In *IEEE International Symposium on Computer-Aided Control System Design*, pages 327–332, 1996. 6

[104] Glenn Wagner and Howie Choset. M*: A complete multirobot path planning algorithm with performance bounds. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 3260–3267. IEEE, 2011. 3

[105] Shih-Ho Wang and E Davison. On the stabilization of decentralized control systems. *Automatic Control, IEEE Transactions on*, 18(5):473–478, 1973. 3

[106] Eric C. Whitman and Christopher G. Atkeson. Multiple model robust dynamic programming. In *American Control Conference (ACC), 2012*, pages 5998–6004. IEEE, 2012. 6

[107] Pierre-Brice Wieber. Trajectory free linear model predictive control for stable walking in the presence of strong perturbations. In *Proceedings of the IEEE-RAS International Conference on Humanoid Robots*, 2006. 2.1, 4

[108] Pierre-Brice Wieber and Christine Chevallereau. Online adaptation of reference trajectories for the control of walking systems. *Robotics and Autonomous Systems*, 54(7):559–566, 2006. 2.1, 4

[109] Wee Chin Wong and Jay H. Lee. Postdecision-state-based approximate dynamic programming for robust predictive control of constrained stochastic processes. *Industrial & Engineering Chemistry Research*, 50(3):1389–1399, 2011. 6

[110] KangKang Yin, Kevin Loken, and Michiel van de Panne. Simbicon: simple biped locomotion control. In *SIGGRAPH '07: ACM SIGGRAPH 2007 papers*, page 105, New York, NY, USA, 2007. ACM. doi: http://doi.acm.org/10.1145/1275808.1276509. 3

[111] Matt Zucker, J. Andrew Bagnell, Christopher G. Atkeson, and James Kuffner. An optimization approach to rough terrain locomotion. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 2010. 4.3.3