Stride-Level Control of Quadrupedal Runners through Optimal Scaling of Hip-Force Profiles

by

Andrés Klee Valenzuela

Submitted to the Department of Mechanical Engineering
in partial fulfillment of the requirements for the degree of

Master of Science in Mechanical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2011

Author .

Department of Mechanical Engineering
August 18, 2011

Certified by.....

Sangbae Kim
Esther and Harold E. Edgerton Assistant Professor of Mechanical Engineering
Thesis Supervisor

Accepted by.....

David E. Hardt
Ralph E. and Eloise F. Cross Professor of Mechanical Engineering
Chairman, Committee on Graduate Students

Stride-Level Control of Quadrupedal Runners through Optimal Scaling of Hip-Force Profiles

by

Andrés Klee Valenzuela

Submitted to the Department of Mechanical Engineering
on August 18, 2011, in partial fulfillment of the
requirements for the degree of
Master of Science in Mechanical Engineering

Abstract

This thesis presents Optimally Scaled Hip-Force Planning (OSHP), a novel approach to controlling the body dynamics of running robots. Controllers based on this approach form the high-level component of a hierarchical control scheme in which they direct lower level controllers, each responsible for coordinating the motion of a single leg. An OSHP controller takes in the state of the runner at the apex of its primary aerial phase and returns desired profiles for the vertical and horizontal forces to be exerted at each hip during the subsequent stride. Controlling the legs so as to match these profiles is left to the lower level leg controllers. The hip force profiles returned by OSHP are scaled variants of nominal force profiles based on biological ground reaction force data. The OSHP controller determines the scaling parameters for these profiles through constrained nonlinear optimization on an approximate model of the runner's body dynamics. Additionally this thesis presents an implementation of an OSHP controller for a simple quadruped model. Evaluation of the controller in simulation shows that even with very simple leg controllers, the OSHP controller can produce bounding and pronking gaits in that model. These gaits emerge as the controller attempts to match particular targets for the runners' states at the apex of their strides. The order in which the feet make contact with ground is not pre-specified. That evaluation also shows that the OSHP controller can compensate for errors introduced by the leg controllers to match given target values for the runners' height, pitch, and pitch rate at the apex of their strides.

Thesis Supervisor: Sangbae Kim
Title: Esther and Harold E. Edgerton Assistant Professor of Mechanical Engineering

Acknowledgments

I'd like to begin by thanking my parents. Without their love, help, and support I might not be an engineer, I probably wouldn't be here at MIT, and I definitely wouldn't be the man I am today. Also their generosity and hospitality have been instrumental in Christina's and my getting through our masters programs while raising our now fourteen-month-old daughter.

A special thanks to Professor Sangbae Kim for giving me the opportunity and freedom to work on such fun problems over the last two years. I also greatly appreciate all of the comments, suggestions, and support I've received from the other members of the Biomimetic Robotics Lab (Arvind Ananthanarayanan, Michael Chuah, Matt Haberland, Jake McKenzie, and Sang Ok Seok). And thanks, of course, to the DARPA M3 program for paying the bills.

Finally, and most importantly I want to thank my wife, Christina, and my daughter, Sofía. Not only have they put up with the many long days of work that went into this research and thesis, but their love provides me with the foundation from which I can do all my work.

+ AMDG +

Contents

List of Figures

List of Tables

Nomenclature

Acronyms

BRL Biomimetic Robotics Laboratory, page 17

DARPA Defense Advanced Research Projects Agency, page 17

MIT Massachusetts Institute of Technology, page 17

OSHP Optimally Scaled Hip-force Planning, page 28

SLIP Spring-Loaded Inverted Pendulum, page 24

Constants

d_i Distance from the runners center of mass to the i-th hip, page 31

I Moment of inertia of the two-legged runner, page 33

l_0 Maximum leg length, page 33

m Mass of the two-legged runner, page 33

μ Static friction coefficient of the ground/foot interaction, page 49

n_{legs} Number of legs of a runner, page 35

n_u Number of scaling parameters for each leg's hip-forces, page 39

r Location of the middle root of $f_x^{asym}(t)$, page 42

T Stride period, page 36

U Constraint set for the OSHP nonlinear program, page 39

w_i Weight for the i-th term of the objective function, $J(\mathbf{u})$, page 51

X Constraint set for a nonlinear program, page 38

Super-Subscript

asym Asymmetric but otherwise unscaled force profile, page 42

d Indicates a desired value for the given variable, page 52

f Value at the end of the current stride, page 45

lb Lower bound, page 48

mid Value at mid-stride, page 45

nom Indicates an unscaled (nominal) force profile, page 42

$'$ Integral controlled value, page 53

r Pertaining to leg return, page 49

rel Indicates a relative distance from the hip to the foot, page 52

$*$ Dimensionless variable, page 33

sw Pertaining to a leg's swing phase, page 49

ub Upper bound, page 48

x Pertaining to the x-axis (horizontal), page 42

y Pertaining to the y-axis (vertical), page 42

Variables

E Binary representation of a phase transition event, page 35

f Force exerted by a leg's prismatic actuator, page 31

f Force exerted by a leg on a runner's body, page 42

$\mathbf{f}_i$ Vector of the horizontal force, vertical force, and torque exerted on the body by the i-th leg, page 39

$f_{i,x}$ Horizontal force exerted on the body by the i-th leg, page 32

$f_{i,y}$ Vertical force exerted on the body by the i-th leg, page 32

γ_i Absolute angle of i-th leg relative to the vertical, page 31

J Cost function for the OSHP nonlinear program, page 39

l_i Length of the i-th leg, page 31

Φ Binary representation of a runner's phase, page 35

$\mathfrak{p}$ Parameterization that returns a force function for a given parameter vector $\mathbf{u}_i$, page 39

P Anti-derivative of the polynomial describing the active (non-zero) portion of a force profile, page 47

p Polynomial describing the active (non-zero) portion of a force profile, page 47

$\mathbb{P}$ Second anti-derivative of the polynomial describing the active (non-zero) portion of a force profile, page 47

$\mathbf{q}$ Vector of generalized coordinates for the two-legged quadruped, page 32

t Time, page 32

τ_i Torque exerted by a leg's revolute actuator, page 31

$\tilde{\tau}$ Approximate hip-torque exerted on the body by a leg, page 53

θ Body pitch relative to the horizontal, page 31

u_1 Time-offset for the leg-forces of the front leg, page 42

u_2 Scaling factor for the duration of the forces exerted by the front leg, page 42

u_3 Scaling factor for the magnitude of the vertical force exerted by the front leg, page 42

u_4 Scaling factor for the magnitude of the horizontal force exerted by the front leg, page 43

u_5 Parameter controlling the asymmetry of the front leg's horizontal force profile, page 43

u_6 Time-offset for the leg-forces of the hind leg, page 42

u_7 Scaling factor for the duration of the forces exerted by the hind leg, page 42

u_8 Scaling factor for the magnitude of the vertical force exerted by the hind leg, page 42

u_9 Scaling factor for the magnitude of the horizontal force exerted by the hind leg, page 43

u_{10} Parameter controlling the asymmetry of the hind leg's horizontal force profile, page 43

$\mathbf{u}_i$ Parameter vector for the i-th leg, page 39

$\mathbf{u}$ Parameter vector for an entire runner, comprised of the parameters for each leg, page 39

x Horizontal coordinate of the runner's center of mass, page 31

$\mathbf{x}$ State vector of the two-legged quadruped, page 32

y Vertical coordinate of the runner's center of mass, page 31

Chapter 1

Introduction

1.1 Motivation

1.1.1 Context: The MIT Cheetah Project

The research presented in this thesis forms a part of the MIT Biomimetic Robotics Laboratory's (BRL) Cheetah project, part of the Defense Advanced Research Projects Agency's (DARPA) Maximum Mobility and Manipulation (M3) program. The goal of the Cheetah project is the creation of a high-speed quadrupedal robot. The BRL plans to have a quadruped capable of running at 30 mph by 2014. A conceptual model of the MIT Cheetah is shown in Figure 1-1. This endeavor requires significant research efforts in mechanical design, motor design, manufacturing, and controls. This thesis presents research aimed at providing a high-level controller for the MIT Cheetah; however, the control approach presented here is not restricted to this particular robot, nor even to quadrupeds. Rather, it is a general approach that can hopefully be of use in the control of any running robot. With that hope in mind, it seems appropriate to begin by considering some of the other strategies that have been used to control robotic runners.

Figure 1-1: Conceptual model of the MIT Biomimetic Robotics Laboratory Cheetah, a high-speed running robot. This thesis presents a high-level control approach for this and other robotic runners

1.1.2 Terminology

Before we turn to the discussion of controllers for running robots, however, let us briefly consider some of the terms and concepts used in describing, modelling, and simulating running systems and quadrupeds in particular. This section introduces the hybrid nature of running systems and then describes some of the terminology used to classify different forms of quadrupedal running.

Hybrid Systems

Running models belong to a class of dynamic systems known as *hybrid systems.* While a thorough definition of hybrid systems is beyond the scope of this thesis (the reader is referred to [37]), this section will give a brief discussion of those aspects of hybrid system theory that are important for the simulation and control of quadrupedal runners.

A *hybrid system* can be loosely defined as a combination of continuous dynamics and discrete events which interact with each other. Running models fit this description in that the system evolves according to one set of continuous dynamics until it encounters a particular event, at which point it begins to evolve according to a differ-

ent set of dynamics [37]. For example, a runner with no feet on the ground follows a ballistic trajectory until a foot makes contact with the ground, after which it follows a trajectory determined by a new set of dynamics that include the forces exerted by the leg. We call the realms in which particular sets of continuous dynamics apply *modes* or *phases* and the mapping from one phase to another the *transition map.* Dynamic systems also contain a set of *reset functions* that convert the system's state immediately before an event to its state immediately after. In the case of running, this reset function is usually an expression of impact losses due to legs making contact with the ground. To be complete, models of running systems must define a transition map and reset functions in addition to the equations of motion for every phase.

Quadrupedal Gaits

Bipedal runners are hybrid systems with only four phases: two single-stance phases, a double-stance phase, and an aerial phase. As a result, there are relatively few period-one bipedal gaits. Quadrupedal runners, in contrast, are hybrid systems with sixteen phases, which leads to a wide variety of gaits. Since this work is concerned with quadrupedal *running*, we will leave aside walking gaits and describe only those gaits that may contain one or more aerial phases. For a more detailed discussion of the gaits of quadrupeds see [17], from which much of the information below is drawn.

Trot In a trot, diagonal pairs of legs make contact together, and the pairs make contact with 180° phase difference between them. An aerial phase may separate the stance phases of the two pairs. The trot is the standard running gait for many animals at low speeds.

Pace In a pace, the front and hind legs on the same side make contact together. The left and right pairs make contact with a 180° phase difference between them. There may be an aerial phase between each double stance phase. The pace is the typical running gait for the camel at low speeds.

Bound and Half-Bound In a bound, the font legs make contact together, as do the hind legs. A half-bound differs in that the front legs do not come into contact with the ground simultaneously. Rather, one leg leads the other by a small amount. A bound may have one aerial phase and a quadruple-stance phase, or it may have two aerial phases. A half-bound usually has only one aerial phase. These gaits are used by small mammals like mice, weasels, and squirrels, and also by some larger mammals during acceleration.

Transverse Gallop In a gallop, one leg leads the other in both the front and hind pairs of legs. In a transverse gallop, the leading legs for both the front and hind pairs of legs are on the same side of the body. Transverse gallops generally have an aerial phase between the stance phases of the front legs and those of the hind legs. The transverse gallop is the standard high-speed gait for many animals, including the horse.

Rotary Gallop In a rotary gallop, the leading legs are on opposite sides for the front and hind pairs of legs. Rotary gallops can have a single aerial phase between the stance phases of the hind legs and those of the front legs or it can have two aerial phases. The rotary gallop with two aerial phases is exemplified by the high-speed running of the cheetah.

1.2 Approaches To Control Of Legged Robots

Armed with the concepts presented in § 1.1.2, this section presents a sampling of the techniques that have been used to control legged robots. Particular attention is paid to controllers for running robots and controllers of quadrupeds.

1.2.1 Heuristic Controllers

Heuristics-based controllers are designed based on empirical or intuitive knowledge of the robot's performance. Certain measured quantities or behaviors are identified

as important to the robot's locomotion, and controllers are implemented that drive those quantities to desired values and force the execution of those behaviors. Such controllers have the advantage of not requiring a detailed model of the robot. It can be difficult, however, to know under which conditions a heuristic controller will operate successfully. Additionally, implementation of such controllers often requires substantial tuning of parameters.

Raibert Controllers

In a series of papers published in the mid-1980s, Raibert et al. set out a class of heuristic controllers for monopods [28, 29], bipeds [19], and quadrupeds [30]. The robots treated in these papers have legs with revolute hip joints and prismatic "knee" joints, both of which are actuated. The control-schemes developed in these papers consist of controllers for three quantities critical to running performance and a finite state-machine that governs when each of these controllers is active. The three controllers address the robot's hopping height, body attitude, and forward velocity. The hopping height controller adjusts the timing and amplitude of a thrust delivered by the prismatic actuator during stance. The body attitude controller also acts during stance by using the hip-torque to servo-control the body attitude. The forward velocity controller acts by setting the leg angle during the flight phase which, in turn, determines the position of the foot relative to the hip at touch-down. The controller contains a term proportional to the current error in velocity; if the robot is moving too quickly, the foot is placed farther forwards, if too slowly, farther back. Over a series of strides this brings the robot to the desired running speed.

The three-controller system described above was first developed for a one-legged robot [28], but was later extended to control two- and four-legged robots thanks to two key observations. The first observation is that as long as only one leg is in contact with the ground at a time, each stance phase of bipedal running is equivalent to a stance phase of one-legged running. Thus, the controllers developed for one-leg are directly transferable to two-legged robots [19]. The second observation is that pairs of legs that contact the ground simultaneously can be treated as a single virtual leg.

This means that the trot, pace, and bound gaits of a quadruped can be treated as bipedal gaits [30]. For those gaits, four legs can be seen as two legs, and each of those two legs can be controlled by the one-leg controller.

An extension of this class of controller can be found in BigDog, a quadrupedal robot developed by Boston Dynamics (a company founded by M. Raibert)[6]. BigDog is a power-autonomous robot with hydraulically-actuated, articulated legs. As an industrial project rather than an academic one, less information is available on BigDog's control algorithms. The company's publications do indicate, however, that, like its predecessors, BigDog uses a state-machine to control transitions between flight and stance phases of each leg, and a control scheme that decomposes locomotion into maintenance of certain quantities—orientation, attitude, and forward velocity—each with its own controller. BigDog demonstrates that the relatively simple heuristic techniques described above can be used, with some modification, to produce robust locomotion over varied terrain.

Reflex Controllers

While heuristic controllers can provide full control of a robot's locomotion, as described above, they more often supplement other classes of controllers. Robots whose primary controllers give only general commands, like the oscillatory outputs of central pattern generators (See § 1.2.3), sometimes use heuristic "reflexes" to fine tune the behavior of the robot. From the point of view of the primary controller, these reflexes are part of the plant.

Since animals are the most successful examples of legged locomotors, it is logical that many heuristic methods take their cues from biological phenomena. This is especially true for reflex controllers, as reflexes themselves are biological phenomena. Kimura, Fukuoka, and Cohen include several biologically inspired reflexes on their robots, Tekken and Tekken2 [13, 21]. These include the vestibulospinal reflex, which adjusts the hip angles to decrease body inclination due to ground slope in the sagital plane, the flexor reflex, which decreases the leg length when an obstacle is encountered during protraction to help avoid stumbling, and a corrective stepping reflex to prevent

falls when encountering a step down. Similarly Lewis and Bekey implemented a paw extension reflex on their robot, Geo-II, that inhibits leg flexion while a leg is loaded [22]. All of these reflexes have counterparts described in biological research.

Other reflexes stem from observation of the robot's failure modes. Tekken2 includes a sideways stepping reflex to counteract its tendency to fall on ground that slopes in the frontal plane and a crossed flexor reflex to prevent the contralateral foot from scuffing when a stance leg flexes excessively [21]. Geo-II uses a postural control method based on equalizing foot pressure. The intuition informing this reflexive controller stems not from biology, but from the author's observations of the relationship between foot pressures and the robot's axes of symmetry.

1.2.2 Model-Based Controllers

In contrast with heuristic controllers, model-based controllers begin with a mathematical model of the robot's dynamics. This model may be a drastically simplified model or one that attempts to incorporate as much detail as possible. The resulting controllers are as varied: they may command only leg touch-down angles, or command the joint torques at all times.

Approaches Based on Passive Dynamics

At one extreme of the continuum of model-based controllers, lie the so-called passive dynamic methods. *Passive dynamics* refers to the movement of a mechanical system without the application of actuator torques. The simplest of passive dynamic walkers are childrens toys [42]. In [24], McGeer presents the first contemporary example of a passive-dynamic walker. The bipedal mechanism he describes is capable of walking down a ramp without any actuation; rather, the pendulum-like motion of the legs returns them to the necessary position at the start of the next stride. The ramp is required to replace the energy lost in the impact of the feet. Collins presents a three-dimensional version of the bipedal passive-dynamic walker in [11]. Later, an actuator is added to the ankle of a similar walker to produce a high-efficiency bipedal robot

[10]. The development of these walkers is based on analyzing a mathematical model of their dynamics and adjusting their design so that their passive-dynamics exhibit stable periodic limit cycles.

Passive dynamic running has been extensively studied in the context of one and two legged systems. Inspired by the elastic energy storage observed in running animals [15], the spring-loaded inverted pendulum (SLIP) model treats the runner as a point mass connected to one massless, compliant leg [3]. The SLIP model displays stable passive running for appropriate fixed attack angles of the leg [33, 14]. Passive dynamic running for a biped with massive, compliant legs is proposed by McGeer in [23].

This approach is extended to a bounding quadrupedal robot in [27]. Poulakakis et al. study the dynamics of a quadrupedal model based on the robot SCOUT II. Their model has a rigid back and massless, linearly-compliant legs. They find that for a given velocity, they can induce a stable bounding gait in the model simply by fixing the angle of the legs relative to the ground at touchdown. This shows the feasibility of applying assisted passive-dynamic controllers like the one used in the Cornell biped [10] to quadrupedal robots. The motion is passive in that it requires no energy to move the massless legs, but is assisted, rather than truly passive, because the model controls the position of its legs during flight. Remy et al. go further in [31] when they show that, in simulation at least, true passive-dynamic walking is possible for a more realistic quadrupedal model whose legs have mass.

Approaches Based on Inverse Dynamics/Hybrid-Zero-Dynamics

The inverse dynamics of a robot are a transformation that, given the robot's state, maps desired accelerations to the joint torques required to produce them. If the inverse dynamics of a robot are known, the robot can be made to track a reference trajectory in the state-space. Computing the inverse dynamics of fixed-base robots is a solved problem [35]. Finding the inverse dynamics of a "floating-base" robot (one that has no single continuous connection to the ground) is much more challenging. A characteristic example of an inverse dynamics based controller is found in [5], in which Buchli et al. present an inverse dynamics controller for Boston Dynamics' LittleDog

robot . They find a formulation of the floating-base inverse dynamics for their model of the robot and then use it to follow center of gravity paths supplied by a higher-level path planning module.

Controllers based on the concept of *zero dynamics* are closely related to those based on inverse dynamics. Indeed, all inverse dynamics methods can be seen as implicitly enforcing a set of zero dynamics on a robot [39]. *Zero dynamics* are those dynamics that a robot can display while satisfying the condition that certain functions of its state, called outputs, are identically zero. A controller based on zero dynamics controls the joint torques of the robot so as to drive the outputs to zero. In the case of the inverse dynamics controller described above, the output functions combine the difference between the robot's current state and acceleration (as predicted by the model) and the desired state and acceleration supplied by the path planning module. When dealing with legged robots, we use the term *hybrid zero dynamics* [39] to indicate that we are referring to the zero dynamics of a hybrid system, a system with both continuous and discrete components (See § 1.1.2). Inverse dynamics based controllers specify zero dynamics indirectly, but controllers can also be designed to specify the desired hybrid zero dynamics directly. This technique is used by Westervelt, Grizzle, and Koditschek to produce stable walking in a simulated biped [39], by Westervelt et al. to produce stable walking in the bipedal robots RABBIT and ERNIE [40], and by Sreenath et al. to produce stable walking in the bipedal robot MABEL [36]. Poulakakis and Grizzle open another intriguing avenue for research - they use the dynamics of a simpler system, the SLIP, as the desired zero dynamics for the more complex monopod, Thumper [26, 25]. This approach allows any of the various existing control laws for the SLIP to be applied directly to the more complex robot [26].

Approaches based on Trajectory Planning, Optimization, and Stabilization

The most heavily model based approaches use forward simulation of a physical model to plan the runner's trajectory. This planning may take the form of optimization, as when Coros et al. present a controller for a simulated quadruped based on opti-

mization of the quadruped's motion [12]. They take the parameters of a closed-loop controller as their optimization variables and a weighted sum of terms assessing the runner's performance as their objective function. Alternatively, the planning may use techniques such as Rapidly Exploring Random Trees (RRTs) to find an open loop "tape" of control actions that lead the simulated system to a desired state. In this case, the trajectory must then be stabilized in order to account for differences between the model and the actual system. Shkolnik et al. employ this approach to control bounding over rough terrain with LittleDog [34]. That paper also demonstrates the usefulness of trajectory planning in incorporating knowledge of upcoming terrain.

1.2.3 Central Pattern Generator Based Controllers

Central pattern generators (CPG) are neural networks that control oscillatory motion, including locomotion, in animals [9]. This idea of oscillatory neural networks providing the control signals for legged locomotion, has been embraced by many robotics researchers. The details of their implementations vary, but the common core is a network of neural oscillators. Each oscillator controls the motion of one joint. The connections between oscillators allows the state of one oscillator to affect that of another. With the proper pattern of interconnections, this can produce a variety of gaits [20]. This approach to the control of quadrupedal robots has produced robots that can trot and bound [20], robots that can walk on natural outdoor terrain [21], robots that can adaptively exploit their own natural dynamics [4], and even robots that can learn to walk in minutes [22]. CPG based controllers have the advantage of not requiring a detailed model, or, indeed, any model at all, of the robot's dynamics.

1.3 A New Approach

1.3.1 Controller Requirements for the MIT Cheetah

Each of the above approaches to controlling legged robots has its own capabilities and limitations. The MIT Cheetah project requires a particular combination of con-

troller capabilities. Its controllers should be able to regulate acceleration as well as quadrupedal running at a variety of different speeds. They should also be able to easily accommodate changes in mechanical design. This is especially important, as the MIT Cheetah is still under development.

The first of these requirements entails either switching between controllers, each designed for a certain range of speeds, or using a single controller that can operate at any of the desired speeds. The former approach requires that the designer choose the conditions at which transitions will occur. In order to avoid imposing such a constraint unnecessarily, each controller should be able to accommodate as large a range of speeds as possible, tending toward a controller that fits the latter approach.

To enable rapid adjustment to mechanical design changes, the MIT Cheetah project has adopted a hierarchical control scheme. In this scheme, a body controller issues commands to individual controllers for each leg. The leg controllers use knowledge of the legs' structure and dynamics to apply forces and torques to the body that approximately match those specified by the body controller's commands. Leg controllers are therefore specific to a particular leg design, but the body controller should be able to work with a wide array of legs.

1.3.2 Design Goals for a Body-Level Control Approach

The requirements listed in § 1.3.1, suggest two main goals for approaches to body-level control of the MIT Cheetah. The first of these goals is modularity. Leg controllers for the MIT Cheetah's legs are being developed in parallel with the research presented here. Approaches to body control should therefore be modular, not relying on the dynamics of any particular leg design.

The second goal suggested by the requirements of the MIT Cheetah project is generality. Control approaches should yield controllers capable of handling many of the different tasks performed by running robots, such as acceleration, steady-state running, and traversing obstacles. To be valuable for future projects and for the rest of the research community, control approaches should also be general in the sense of being easily applicable to a wide range of running systems.

These goals pose problems for many of the control approaches described in § 1.2. Controllers based on either passive dynamics or inverse dynamics are infeasible because they would require knowledge of the leg dynamics. Traditional trajectory optimization, performed with detailed models, presents the same problem. CPG based approaches typically apply to a particular gait and speed, which goes against the goal of generality.

1.3.3 Optimally Scaled Hip-Force Planning

The following chapters develop Optimally Scaled Hip-force Planning (OSHP), a new approach to controlling the body dynamics of running robots, that combines heuristics and trajectory optimization. An OSHP controller uses an approximation of the runner's body dynamics to prescribe the forces and torques the legs should exert at the hips (and shoulders) during the following stride. It does this by selecting scaling parameters for nominal hip-force profiles via constrained nonlinear optimization. This approach offers two advantages over traditional trajectory-optimization. First, the use of approximate dynamics allows the formulation of simpler constraints and objective functions for the optimization. Second, taking the hip-forces as control outputs makes the body controller independent of the particular type of leg used in the runner. This satisfies the goal of modularity mentioned above, allowing the same OSHP controller to be used for systems with different leg dynamics.

OSHP also offers a great deal of generality, both in terms of its utility for diverse tasks and in terms of its applicability to diverse running systems. OSHP controllers could be designed for runners with any number of legs. However, this thesis develops OSHP in the context of quadrupeds. This is primarily because this research is being carried out in the context of the MIT Cheetah Project, but also because it is easier to describe OSHP with reference to a single type of runner.

The remainder of the thesis is organized as follows: Chapter 2 presents the quadrupedal model used in this research and outlines the control approach, OSHP, that is the major contribution of this work. Chapter 3 gives a full description of OSHP and details the specific implementation used to validate OSHP in simulation. Chap-

ter 4 presents the results of this evaluation and provides discussion and directions for future research. Chapter 5 concludes the work.

Chapter 2

Defining the Plant: A Simple Quadrupedal Running Model

This chapter presents the model of quadrupedal running used in the description of OSHP (Chapter 3) and in the simulated evaluation of a specific implementation of OSHP (Chapter 4). The model assumes that the runner's front and hind pairs of legs each act as a unit. The result is a two-legged "quadruped" with one leg in the front and one in the rear. This model can only portray bounding and pronking gaits, but its simplicity makes it an excellent vehicle for presenting OSHP and a useful test-case for implementation of an OSHP controller.

2.1 Equations of Motion

Figure 2-1 shows a schematic of the two-legged quadruped. The body of the runner is modeled as a single rigid body with an asymmetric mass distribution. The legs are modeled as being massless, therefore this model is suitable only for systems in which the mass of the legs is very small relative to that of the body. Each leg has two actuators: a prismatic actuator that acts along the leg and a rotary actuator that acts about the hip or shoulder joint (for the sake of simplicity, I will hereafter refer to the proximal joints of both front and hind legs as "hips"). Contact between the feet and the ground is modeled as being instantaneous, that is, the foot velocity goes to

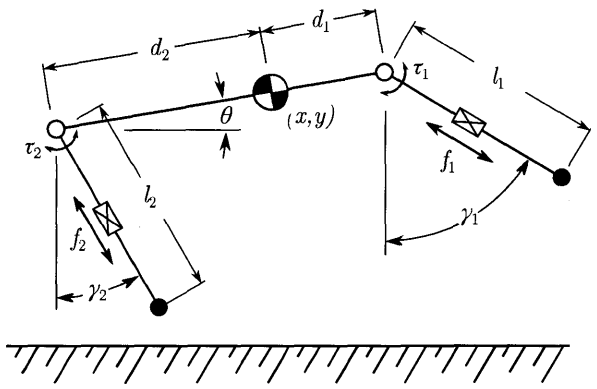


Figure 2-1: Two-legged "quadruped." This model treats the front and back pairs of legs as one leg each. It provides a simple platform for designing and testing controllers while incorporating many of the characteristics of quadrupedal running

zero as soon as the foot touches the ground. However, because the legs are massless, there are no impacts associated with these contact events.

Take the generalized coordinates of the body to be

$$\mathbf{q}(t) = \begin{pmatrix} x(t) \\ y(t) \\ \theta(t) \end{pmatrix}.$$

The state of the body is

$$\mathbf{x}(t) = \begin{pmatrix} \mathbf{q}(t) \\ \dot{\mathbf{q}}(t) \end{pmatrix}.$$

Let the forces and torque exerted on the body by leg i be given by $f_{i,x}$, $f_{i,y}$, and τ_i respectively. Note that these quantities are not independent but are related by

$$\tau_i(t) = l(t)\left[f_{i,x}\cos\gamma(t) + f_{i,y}\sin\gamma(t)\right].,$$

where l and γ are the length and angle of the leg as shown in Figure 2-1

With these definitions, the equations of motion for the body are

$$\ddot{x}(t) = \frac{1}{m}\left[f_{1,x}(t) + f_{2,x}(t)\right] \tag{2.1a}$$

$$\ddot{y}(t) = \frac{1}{m}\left[f_{1,y}(t) + f_{2,y}(t)\right] - g \tag{2.1b}$$

$$\begin{aligned}\ddot{\theta}(t) = &\frac{d_1}{I}\left[-\sin\theta(t)\cdot f_{1,x}(t) + \cos\theta(t)\cdot f_{1,y}(t)\right] \\ &- \frac{d_2}{I}\left[-\sin\theta(t)\cdot f_{2,x}(t) + \cos\theta(t)\cdot f_{2,y}(t)\right] \\ &+ \frac{1}{I}\left[\tau_1(t) + \tau_2(t)\right]\end{aligned} \tag{2.1c}$$

2.1.1 Dimensionless Equations of Motion

To make the results of this research as broadly applicable as possible, we recast (2.1) in terms of dimensionless characteristics and use the resulting dimensionless equations of motion in all following sections. The results obtained in subsequent sections are therefore applicable to any system having the same values for those dimensionless characteristics.

Let the characteristic length of the system be the maximum leg-length, l_0, the characteristic mass be the mass of the body, m, and the characteristic force be the weight of the body mg. The dimensionless time is therefore

$$t^* = \sqrt{\frac{g}{l_0}}t\,.$$

The dimensionless state is

$$\mathbf{x}^* = \begin{pmatrix} x^* \\ y^* \\ \theta^* \\ \frac{dx^*}{dt^*} \\ \frac{dy^*}{dt^*} \\ \frac{d\theta^*}{dt^*} \end{pmatrix} = \begin{pmatrix} \frac{1}{l_0}x \\ \frac{1}{l_0}y \\ \theta \\ \sqrt{\frac{1}{l_0 g}}\dot{x} \\ \sqrt{\frac{1}{l_0 g}}\dot{y} \\ \sqrt{\frac{l_0}{g}}\dot{\theta} \end{pmatrix}. \tag{2.2}$$

The geometric and physical properties can likewise be transformed to yield dimensionless analogs. The dimensionless versions quantities along with the hip forces and torques are given in Table 2.1 With the definitions given in (2.2) and Table 2.1, (2.1) simplifies to the following dimensionless equations of motion

$$\ddot{x}^*(t) = f^*_{1,x}(t) + f^*_{2,x}(t) \tag{2.3a}$$

Dimensionless Quantity	Symbol	Definition
Distance from CoM to hip	d^*	$d_i = l_0 d_i^*$
Moment of inertia	I^*	$I = ml_0^2 I^*$
Hip-force	f_i^*	$f_i = mgf_i^*$
Hip-torque	τ^*	$\tau = mgl_0\tau^*$

Table 2.1: Dimensionless analogs for properties, forces and torques in (2.1)

$$\ddot{y}^*(t) = f_{1,y}^*(t) + f_{2,y}^*(t) - 1 \tag{2.3b}$$

$$\begin{aligned}\ddot{\theta}^*(t) = &\frac{d_1^*}{I^*}\left[-\sin\theta(t)\cdot f_{1,x}^*(t) + \cos\theta(t)\cdot f_{1,y}^*(t)\right] \\ &- \frac{d_2^*}{I^*}\left[-\sin\theta(t)\cdot f_{2,x}^*(t) + \cos\theta(t)\cdot f_{2,y}^*(t)\right] \\ &+ \frac{1}{I^*}\left[\tau_1^*(t) + \tau_2^*(t)\right]\end{aligned} \tag{2.3c}$$

From this point forward, all quantities used in this thesis are dimensionless. To avoid visual clutter, therefore, we will omit the star notation used here in all subsequent sections. Also, throughout the rest of this thesis, the expressions $\dot{(\cdot)}$ and $\ddot{(\cdot)}$, refer to derivatives with respect to *dimensionless* time. Thus, when angular quantities are given in terms of degrees (as they will be in later sections), angular velocities have units of degrees, not degrees per second.

2.2 Transition Map

As stated in § 1.1.2, to simulate a quadrupedal runner, one needs not only the equations of motion given by (2.3) but also a mapping of the possible transitions between phases and the events that cause them. Figure 2-2 shows such a transition map for the two legged model shown in Figure 2-1. These maps could be implemented as a set of instructions for what the new phase should be for every combination of current phase and event, but this would be cumbersome, especially in the four-legged case. However, a simple approach borrowed from computer science allows the transition between any two phases to be accomplished with a single bitwise operation. More-

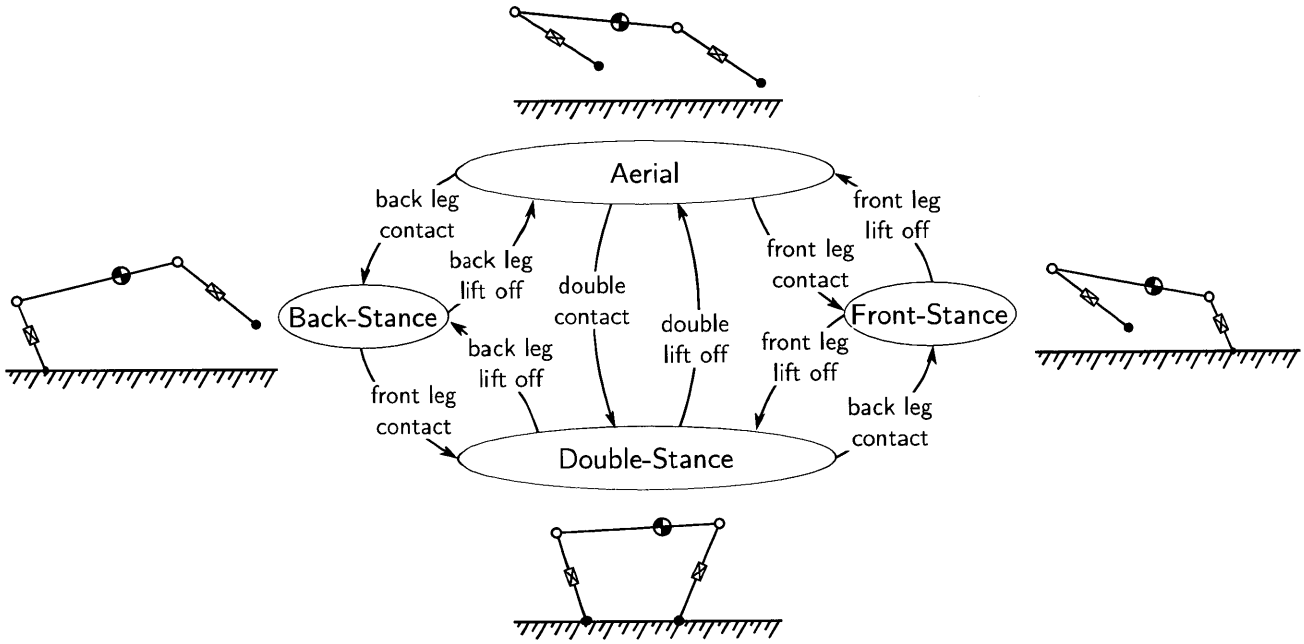


Figure 2-2: Transition map for the two-legged quadruped. Phases are represented by ellipses. Transitions are represented by arrows and labeled with the event that triggers the transition.

over, that operation is independent of the number of legs, allowing the same code to be used in simulating models with different numbers of legs.

To implement this approach we let the n_{legs}-bit binary number Φ represent the phase of the runner. Each bit of Φ is 1 if the corresponding leg is in contact with the ground and 0 otherwise. We also express the event that has necessitated the phase transition as another n_{legs}-bit binary number, E. Each bit of E is 1 if the corresponding leg changes state (stance $\rightarrow$ swing or swing $\rightarrow$ stance) and 0 otherwise. The event can then be used as a bitmask to toggle the appropriate bits of the phase, which simply requires applying the bitwise exclusive OR operation (XOR) to Φ and E.

2.3 Apex-to-Apex Model

All of the gaits described in § 1.1.2 are periodic: the runner's state at time t is the same as its state at time $t + T$, where T is the stride period. This can be used to

analyze the performance of a given running controller: if the controller is inducing a periodic gait, the state of the runner at a given point in the stride will converge to a fixed-point over multiple strides. The mapping of the runner's state at a certain point in the current stride to its state at the same point in the next stride is a type of mapping called a *Poincaré return map* or simply a *return map.* In the analysis of running models and controllers, it is often useful to consider the apex-to-apex return map rather than return maps based on other points in the stride. The apex of a stride occurs when the runner's center of mass reaches its highest point during the aerial phase (or principal aerial phase if there are two). The apex-to-apex return map offers two advantages over other possible return maps, such as one between touch-down times of a given leg. First, the apex occurs in every stride, while it is possible that other conditions may not. Second, the vertical velocity of the runner is zero at apex by definition, which decreases the dimension of the return map by one.

2.4 Problem Statement

The previous sections of this chapter provide all of the components necessary to present the full problem statement that forms the starting point for the development of OSHP:

> Given a desired behavior and a runner defined by the equations of motion in (2.3) and the transition map in Figure 2-2, set $(f_i(t), \tau_i(t))$ whenever leg i is in stance and $(l_i(t), \gamma(t))$ whenever it is not, such that the runner displays the desired behavior.

Chapter 3 presents an approach to the control of a runner's body dynamics that, in conjunction with appropriate controllers of the runner's leg dynamics, provides a solution to this problem for many desired behaviors, including acceleration and steady-state running.

Chapter 3

Optimally Scaled Hip-Force Planning

3.1 Body Control and Leg Control

3.1.1 Different Controllers for Disparate Tasks

Successful running, as laid out in § 2.4, requires an appropriate combination of the runner's configuration at the touch-down of each leg, and the behavior of each leg during stance. The control of a running robot, therefore, consists of two primary tasks:

Task 1: Choose appropriate touch-down configurations and leg behaviors

Task 2: Ensure that the robot's touch-down configurations and leg behaviors match those chosen in Task 1

In Raibert's one legged hopping machine, for instance, one part of the controller chooses the desired touch-down angle of the leg, while another servo-controls the hip actuator to attain that angle [28].

These two tasks are inherently different. Task 1 is really a planning task that may need to consider the behavior of the robot over an extended period of time. Task 2, in contrast, is a true control problem that can typically be expressed as a function of

the robot's current state. It makes sense, therefore, to view the controller design for a running robot as two distinct, but interconnected problems. We will refer to the planning problem of Task 1 as *body control*, since it involves the dynamics of the entire robot, and the control problem of Task 2 as *leg control*, as it is primarily concerned with the action of the legs.

The remainder of this section describes a particular approach to body control. This approach considers the motion of the runner on the stride level, that is, from one apex to the next. A stride-level approach takes advantage of the fact that running can be viewed as a discrete process in the state of the runner at apex. Periodic gaits correspond to fixed-points on the apex-to-apex return map. Controlling the state of the runner at apex, therefore, can provide a means to produce such gaits. This does not mean that the behavior of the robot during the stride is ignored, but rather that the stride is the basic unit over which the controller acts.

3.1.2 Stride-level Body Control: A Constrained Nonlinear Program

In its most general form, stride-level control of the body dynamics consists of choosing the forces and torques at the hips as functions of time, $\mathbf{f}_i(t)$, $i = 1, \ldots, n_{legs}$, such that the body trajectory, $\mathbf{x}(t)$ satisfies some criteria. That choice fits quite neatly under the umbrella of nonlinear programming. Nonlinear programming is the solution of the optimization problem

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & J(x) \\ \text{subject to} \quad & x \in X, \end{aligned}$$

where the constraint set, X, is continuous and either the objective function, J, or the constraint set X is defined by nonlinear equations or inequalities [2]. In the case of stride-level control, the transformation that maps the current apex state, $\mathbf{x}_0$, and the hip-forces and torques to the body trajectory, together with the criteria on that trajectory, serves to define both the cost function and the constraint set.

Although it is possible to define a nonlinear program for stride-level body control

in its most general form, the resulting program has the disadvantage of involving the arbitrary functions $\mathbf{f}_i(t)$ which makes it a problem in infinite dimensions. We must reduce it to a problem in a finite number of dimensions if we want it to be tractable. One way to do this is by parameterizing $\mathbf{f_i(t)}$ by a vector of n_u parameters, $\mathbf{u}_i \in \mathbb{R}^{n_u}$, and a function $\mathfrak{p} : \mathbb{R}^{n_u} \to [\mathbb{R} \to \mathbb{R}^3]$, such that

$$\mathbf{f}_i(t) = \mathfrak{p}(\mathbf{u}_i), \quad \text{for } i = 1, \ldots, n_{legs}\,. \tag{3.1}$$

Time-discretization falls into this form of parameterization and is used extensively in trajectory optimization. In a time-discretization, $\mathbf{u}_i$ is a "tape" of force values at n_u points in time and the range of $\mathfrak{p}$ is a set of functions that interpolate between those values. Common interpolation schemes include the zero-order hold, piecewise linear interpolation, and splines of various orders. In the following section we will describe a different parameterization scheme, one based on scaling certain nominal force profiles.

If we let $\mathbf{u} \in \mathbb{R}^{n_u n_{legs}}$ be a single vector composed of all of the elements of $\mathbf{u}_1$ through $\mathbf{u}_{n_{legs}}$, and let (3.1) define our force vectors, we can write the following nonlinear program:

$$\begin{aligned} \underset{\mathbf{u}}{\text{minimize}} \quad & J(\mathbf{u}) \\ \text{subject to} \quad & \mathbf{u} \in U, \end{aligned} \tag{3.2}$$

where J and U are the same cost function and constraint set as for infinite dimensional problem, re-expressed in terms of u. The goal in designing a stride-level body controller is to choose the objective function, $J(\mathbf{u})$, and constraint set, U, such that the solution of (3.2) yields a body trajectory $\mathbf{x}(t)$ that satisfies the criteria for running.

3.2 Scaled Hip-Force Profiles

3.2.1 Motivation

We can convert the problem of choosing the function $\mathbf{f}_i(t)$ to that of choosing a finite number of parameters $\mathbf{u}_i$ by making the elements of $\mathbf{u}_i$ scaling factors for certain

nominal force profiles. Equation (3.3) shows the application of some possible scaling factors to a function $f^{nom}(t)$

$$f(t) = u_3 f^{nom}(u_2 t - u_1), \tag{3.3}$$

where u_1 is a time offset that shifts $f^{nom}(t)$ along the time-axis, u_2 is a scaling factor for duration, and u_3 is a scaling factor for magnitude. In general, the nominal force profiles could be defined by any function. However, the choice of f^{nom} provides an opportunity to use insights from the most successful running systems: animals. If we use nominal force profiles that have a similar shape to the corresponding force profiles for animals, the resulting force profiles will have more structure than a time-discretization for the same number of parameters. In the case of controlling a quadruped, insight can be drawn from bio-mechanical investigations of galloping. Walter and Carrier present the ground reaction forces exerted by galloping dogs in [38]. The results shown in Figure 3-1 provide a starting-point for choosing nominal force-profiles for an OSHP controller of the two-legged quadruped model.

3.2.2 Selection of Force-Profiles for an OSHP Controller

Criteria for Force-Profiles

The previous section mentioned one possible consideration in designing a parameterized force-profile: Matching the nominal profile to observed profiles. Additionally, the parameterization should require as few parameters as possible. The former makes it more likely that the parameterized force profiles can produce the desired behavior. The latter decreases the computational time needed to solve (3.2), which is critical if the planner is to be run online.

Vertical Force

In Figure 3-1 we see that each leg's vertical force profile starts and ends at zero and is concave down everywhere, except for an initial spike. This spike is most likely due to the impact with the ground. Since the model described in Chapter 2 has massless legs, it experiences no impact and the spike can be disregarded. A first pass

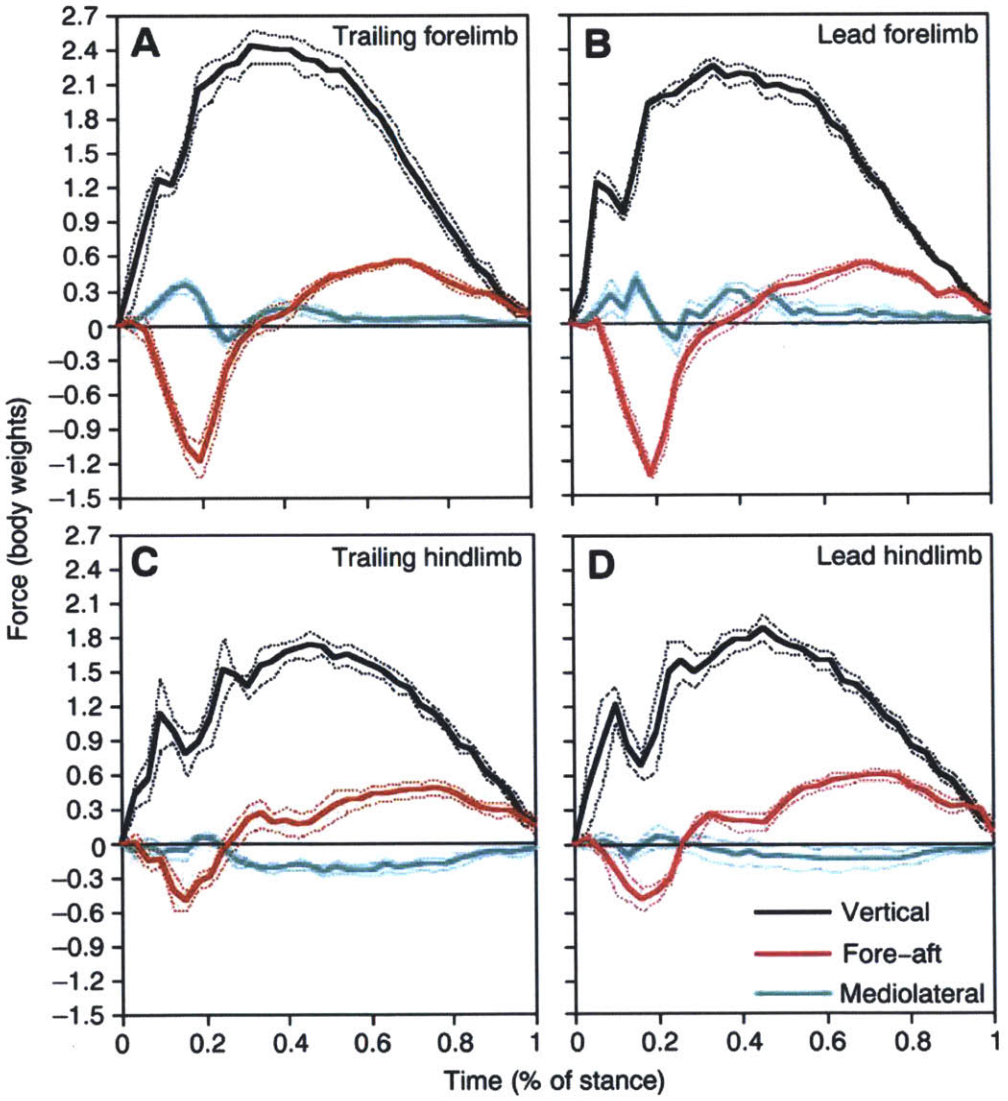


Figure 3-1: Ground reaction force curves for a galloping dog as presented by Walter and Carrier [38]. Reproduced with permission from *The Journal of Experimental Biology*.

at a nominal force profile could therefore be a quadratic with a negative coefficient of the squared term and $f_y^{nom}(0) = 0$. If the stance phase has unit duration, then $f_y^{nom}(1) = 0$, and we have

$$f_y^{nom} = \left(-ct^2 + ct\right)\left[H\left(t\right) - H\left(t-1\right)\right], \tag{3.4}$$

where H is the Heaviside step function and c is an arbitrary constant. To assign a value to the constant c, assume that the nominal force applies a unit impulse in unit time. The nominal force is then,

$$f_y^{nom} = \left(-6t^2 + 6t\right)\left[H\left(t\right) - H\left(t-1\right)\right]$$

This can be parameterized by a time-offset, u_1, a scaling factor for duration, u_2, and a scaling factor for magnitude, u_3, to yield

$$\begin{aligned} f_y\left(t\right) &= u_3\left[-6u_2^2(t-u_1)^2 + 6u_2(t-u_1)\right]\left[H\left(t-u_1\right) - H\left(t-u_1-u_2^{-1}\right)\right] \\ &= \mathfrak{p}_y\left(\mathbf{u}\right). \end{aligned} \tag{3.5}$$

Horizontal Force

The fore-aft forces shown in Figure 3-1 also start and end at zero, but change sign at some point during the stride. The negative portion is concave up, while the positive portion is concave down. A cubic polynomial provides a simple approximation of such a profile. Consider a cubic with roots at 0, ½, and 1

$$f_x^{nom}\left(t\right) = c\left(-2t^3 + 3t^2 - t\right)\left[H\left(t\right) - H\left(t-1\right)\right]. \tag{3.6}$$

This force profile fits the description above, but has a significant flaw: it maintains odd symmetry about mid-stance under all of the forms of scaling presented in (3.3). A horizontal force profile with odd symmetry about mid-stance exerts zero net impulse and therefore cannot accelerate the runner. To allow for asymmetric horizontal force profiles, let the intermediate root of the cubic be located not at ½, but at $r \in [0, 1]$. This gives

$$f_x^{asym}\left(t\right) = c\left(-t^3 + \left(1+r\right)t^2 - rt\right)\left[H\left(t\right) - H\left(t-1\right)\right]. \tag{3.7}$$

Define a new parameter, $u_5 \in [-1, 1]$, as a measure of the asymmetry of the x-force profile:

$$u_5 = 2r - 1\,.$$

Thus, when $u_5 = -1$, the intermediate root falls at the beginning of the force-profile, when $u_5 = 0$, it falls at the center, making the force-profile symmetric, and when $u_5 = 1$, the intermediate root falls at the end of the force-profile. Substituting the definition of u_5 into (3.7) and incorporating a factor of ½ into c yields

$$f_x^{asym}(t) = c\left(-2t^3 + (3+u_5)\,t^3 - (1+u_5)\,t\right)\left[H(t) - H(t-1)\right].$$

This suggests a succinct way of presenting the asymmetric force-profile as a modification of (3.6)

$$f_x^{asym}(t) = f_x^{nom}(t) + \left(cu_5t^2 - cu_5t\right)\left[H(t) - H(t-1)\right].$$

For the purpose of simplifying the expression of some constraints (See §3.3.2), 6 is a convenient choice for the constant, c. Taking $c = 6$ gives

$$f_x^{nom}(t) = -12t^3 + 18t^2 - 6t\left[H(t) - H(t-1)\right]$$

and

$$f_x^{asym}(t) = f_x^{nom}(t) + 6u_5\left(t^2 - t\right)\left[H(t) - H(t-1)\right].$$

It is reasonable to assume that the vertical and horizontal forces will begin and end at the same times. Therefore, the parameters u_1 and u_2 can also be applied to the horizontal force. Let u_4 be a scaling factor for the magnitude of f_x. The expression for the parameterized horizontal force is therefore

$$\begin{aligned} f_x(t) &= u_4 f_x^{asym}\left(u_2\left(t - u_1\right)\right) \\ &= 6u_4\left[-2\left(t-u_1\right)^3 + 3\left(t-u_1\right)^2 - 6\left(t-u_1\right)\right. \\ &\qquad \left. + u_5\left(t^2 + u_1^2 - 2tu_1 - t + u_1\right)\right]\left[H\left(t-u_1\right) - H\left(t - u_1 - u_2^{-1}\right)\right] \\ &= \mathfrak{p}_x(\mathbf{u})\,. \end{aligned} \tag{3.8}$$

Hip-Torque

Since the model described in Chapter 2 has only two actuated degrees of freedom in each leg, no leg controller will be able to match arbitrary vertical and horizontal forces as well as an arbitrary hip-torque. Multiple studies have shown, however, that idealized quadrupedal runners can run without any hip-torques [27, 8]. Therefore, while the planner will not attempt to prescribe a hip-torque profile, it will, in most cases, assume the hip torques to be zero throughout the stride:

$$\mathfrak{p}_\theta(\mathbf{u}) \equiv 0\,. \tag{3.9}$$

An exception to this rule is described in § 3.3.3.

3.3 Control Algorithm

3.3.1 Philosophy and Approximate Dynamics

We return now to the construction of a constrained nonlinear program for the problem of stride-level control of a running quadruped. With the parameterizations given in (3.5) through (3.9), (3.1) becomes

$$\mathfrak{p}(\mathbf{u}) = \begin{pmatrix} \mathbf{f}_1(t) \\ \mathbf{f}_2(t) \end{pmatrix} = \begin{pmatrix} \mathfrak{p}_x(\mathbf{u}_1) \\ \mathfrak{p}_y(\mathbf{u}_1) \\ \mathfrak{p}_\theta(\mathbf{u}_1) \\ \mathfrak{p}_x(\mathbf{u}_2) \\ \mathfrak{p}_y(\mathbf{u}_2) \\ \mathfrak{p}_\theta(\mathbf{u}_2) \end{pmatrix} \tag{3.10}$$

For the two-legged quadruped described in Chapter 2, the variable to be optimized, $\mathbf{u}$, is a ten-element vector.

Many algorithms for solving (3.2) numerically require the computation of gradients, and possibly Hessians, of both $J(\mathbf{u})$ and the constraint functions that define U. There are many methods for calculating the gradients of functions whose values are determined by numerical simulation [32, 7, 18], such an approach could be used to

define gradients for constraints on the body trajectory $\mathbf{x}(t)$. However, the simplicity of the parameterizations (3.5), (3.8), and (3.9), opens the possibility of constructing analytical expressions for constraints and objective functions related to the body trajectory. Such expressions can then be differentiated analytically to yield the required gradients and Hessians.

The objective function and constraints presented in §§ 3.3.2 and 3.3.3 require expressions for the state of the runner at two times: the time when the runner reaches its next apex, t_f and the time at mid-stride, $t_{mid} = {}^{t_f}/_2$. These times can be found by noting that $\dot{y}(0) = 0 = \dot{y}(t_f)$. This requires that

$$\int_0^{t_f} (f_{1,y}(t) + f_{2,y}(t) - 1)\, dt = 0$$

Integrating and solving for t_f, yields

$$t_f = \frac{u_3}{u_2} + \frac{u_8}{u_7}.$$

In order to solve the system of differential equations defined by (2.3) and (3.5) through (3.10) in closed form, we make the simplifying assumption that the pitch, θ, is small. This decouples (2.3) yielding

$$\begin{pmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{\theta} \end{pmatrix} = \begin{bmatrix} f_{1,x} + f_{2,x} \\ f_{1,y} + f_{2,y} - 1 \\ \frac{d_1}{I} f_{1,y} + \frac{d_2}{I} f_{2,y} \end{bmatrix}, \tag{3.11}$$

where d_1 and d_2 are the distances from the runner's center of mass to the anterior and posterior hips respectively. (3.11) can be integrated twice to yield expressions for x, y, and θ at specific times.

From (3.11), it is apparent that the state trajectories are the sum of four or five terms:

- Constant term due to the initial state
- Linear term due to initial velocity
- Quadratic gravitational term (y only)

- One term for each applied force

  This is the trajectory that the state would follow if only that force were applied to it.

Finding the state at t_f is simply a matter of adding the final values of each of these terms. The constraints in §3.3.2 require the y, θ, $\dot{x}$, and $\dot{\theta}$ components of the state. A full derivation of these quantities is given in the appendix. The results of that derivation are:

$$y_f = y_0 \overbrace{- \frac{1}{2}\left(\frac{u_3}{u_2} + \frac{u_8}{u_7}\right)^2}^{\text{gravitational term}} \overbrace{+ u_3\left(\frac{u_8}{u_2 u_7} - \frac{2u_1u_2 - 2u_3 + 1}{2u_2^2}\right)}^{\text{term from } f_{1,y}} \underbrace{+ u_8\left(\frac{u_3}{u_7 u_2} - \frac{2u_6u_7 - 2u_8 + 1}{2u_7^2}\right)}_{\text{term from } f_{2,y}} \tag{3.12a}$$

$$\theta_f = \theta_0 + \dot{\theta}_0\left(\frac{u_3}{u_2} + \frac{u_8}{u_7}\right) \overbrace{- \frac{r_1}{2I^*}\left(\frac{u_3}{u_2^2}\right)\left(1 - \frac{2u_2u_8}{u_7} - 2u_3 + 2u_1u_2\right)}^{\text{term from } f_{1,y}} \underbrace{- \frac{r_1}{2I^*}\left(\frac{u_8}{u_7^2}\right)\left(1 - \frac{2u_7u_3}{u_2} - 2u_8 + 2u_6u_7\right)}_{\text{term from } f_{2,y}} \tag{3.12b}$$

$$\dot{x}_f = \dot{x}_0 - \frac{u_4u_5}{u_2} - \frac{u_{10}u_9}{u_7} \tag{3.12c}$$

$$\dot{\theta}_f = \dot{\theta}_0 + \frac{d_1}{I}\left(\frac{u_3}{u_2}\right) + \frac{d_2}{I}\left(\frac{u_8}{u_7}\right) \tag{3.12d}$$

Finding expressions for the necessary components of the state at mid-stride, y_{mid} and θ_{mid}, is more involved. By definition, t_f is reached after both legs have finished their stance phases. Mid-stride, however, can fall during the stance phase of either leg or of both. As a result, the state at t_{mid} must be defined piecewise or with step functions. Since $f_{1,x}$, $f_{1,y}$, $f_{2,x}$, and $f_{2,y}$ are defined using step functions, we will do the same here. For notational convenience we define the following functions of t in terms of the parameters of $\mathbf{f}_1$:

$$p_{1,x}(t) = 6u_4\left[-2t^3 + 3t^2 - 6t + u_5\left(t^2 - t\right)\right] \tag{3.13a}$$

$$p_{1,y}(t) = u_3\left[-6u_2^2(t-u_1)^2 + 6u_2(t-u_1)\right], \tag{3.13b}$$

$$P_{1,x}(t) = \int_{u_1}^{t} p_{1,x}(\tau)\, d\tau, \tag{3.13c}$$

$$P_{1,y}(t) = \int_{u_1}^{t} p_{1,y}(\tau)\, d\tau, \tag{3.13d}$$

$$\mathbb{P}_{1,x}(t) = \int_{u_1}^{t} P_{1,x}(\tau)\, d\tau, \tag{3.13e}$$

and

$$\mathbb{P}_{1,y}(t) = \int_{u_1}^{t} P_{1,y}(\tau)\, d\tau. \tag{3.13f}$$

Analogous terms, $p_{2,x}$, $p_{2,y}$, etc., can be defined using the parameters of $\mathbf{f}_2$. With this notation,

$$f_{1,x}(t) = p_{1,x}(t)\, H(t-u_1) - p_{1,x}(t)\, H\left(t-u_1-u_2^{-1}\right)$$

and

$$f_{1,y}(t) = p_{1,y}(t)\, H(t-u_1) - p_{1,y}(t)\, H\left(t-u_1-u_2^{-1}\right)$$

As shown in the appendix, the height at mid-stride is given by

$$\begin{aligned} y_{mid}(\mathbf{u}) = y_0 - \frac{t_{mid}^2}{2} &+ \mathbb{P}_{1,y}(t_{mid})\, H(t_{mid}-u_1) \\ &+\left[-\mathbb{P}_{1,y}(t_{mid})+\mathbb{P}_{1,y}\left(u_1+u_2^{-1}\right)+P_{1,y}\left(u_1+u_2^{-1}\right)\left(t_{mid}-u_1-u_2^{-1}\right)\right] H\left(t_{mid}-u_1-u_2^{-1}\right) \\ &+ \mathbb{P}_{2,y}(t_{mid})\, H(t_{mid}-u_6) \\ &+\left[-\mathbb{P}_{2,y}(t_{mid})+\mathbb{P}_{2,y}\left(u_6+u_7^{-1}\right)+P_{2,y}\left(u_6+u_7^{-1}\right)\left(t_{mid}-u_6-u_7^{-1}\right)\right] H\left(t_{mid}-u_6-u_7^{-1}\right) \end{aligned} \tag{3.14}$$

and the pitch at mid-stride is given by

$$\begin{aligned} \theta_{mid}(\mathbf{u}) = \theta_0 + \dot{\theta}_0 t_{mid} &+ \frac{d_1}{I}\mathbb{P}_{1,y}(t_{mid})\, H(t_{mid}-u_1) \\ &+\frac{d_1}{I}\left[-\mathbb{P}_{1,y}(t_{mid})+\mathbb{P}_{1,y}\left(u_1+u_2^{-1}\right)+P_{1,y}\left(u_1+u_2^{-1}\right)\left(t_{mid}-u_1-u_2^{-1}\right)\right] H\left(t_{mid}-u_1-u_2^{-1}\right) \\ &+ \frac{d_2}{I}\mathbb{P}_{2,y}(t_{mid})\, H(t_{mid}-u_6) \\ &+\frac{d_2}{I}\left[-\mathbb{P}_{2,y}(t_{mid})+\mathbb{P}_{2,y}\left(u_6+u_7^{-1}\right)+P_{2,y}\left(u_6+u_7^{-1}\right)\left(t_{mid}-u_6-u_7^{-1}\right)\right] H\left(t_{mid}-u_6-u_7^{-1}\right). \end{aligned} \tag{3.15}$$

Despite the fact that y_{mid} and θ_{mid} are defined in terms of step functions, they are, in fact, continuous and twice-differentiable. They can be differentiated simply by treating the step functions as constants and proceeding as usual. This is not the case in general; it is possible only because of the nature of the expressions for y_{mid} and θ_{mid}.

Table 3.1: Upper and lower bounds on the elements of $\mathbf{u}$

Parameter	Lower Bound	Upper Bound
u_1	0	2
u_2	0	20
u_3	0.5	4
u_4	-0.2	2
u_5	-1	1

3.3.2 Constraints

For appropriate initial conditions, the constraints described in this section, together with the objective function described in § 3.3.3, make the solution of (3.2) a valid body planner.

Bounds on Parameters

Let $\mathbf{u}_{lb}, \mathbf{u}_{ub} \in \mathbb{R}^{n_u n_{legs}}$ be vectors containing lower and upper bounds on the elements of $\mathbf{u}$ respectively. The bounds used in simulation are shown in Table 3.1. Only u_1 through u_5 are shown, as the bounds for u_6 through u_{10} are the same. The lower bounds on u_2 and u_3 are required for the parameters to retain their meaning: the stance phase cannot have negative duration and the leg cannot pull up on the ground. A loose upper bound on u_3 can be derived from the plots in Figure 3-1, which show that the vertical force exerted by one leg never exceeds three times the body weight. The maximum value of f_y^{nom} is one and a half times the body weight; therefore a single leg should never exert a vertical force greater than $2f_y^{nom}$. Since each leg of the two-legged quadruped model represents a combination of two legs, this limit is doubled to give the result shown in the table. The remaining bounds were chosen by hand and do not appear to affect the solutions of (3.2).

Return Time Constraint

Since the legs of our model are massless, they can, in principle, return to a forward position arbitrarily quickly after the end of stance. Allowing this would, however,

make the planner unusable for any real system. Physiological research shows that, for humans at least, there is a minimum time required to return each leg that does not differ significantly over runners whose maximum attainable speeds vary widely [41]. We therefore impose a return time constraint on u_1 and u_6

$$u_1 \geq t_r - t_{1,sw}$$
$$u_6 \geq t_r - t_{2,sw}\,,$$

where t_r is a minimum return time and $t_{1,sw}$ and $t_{2,sw}$ are the elapsed times since the front and hind legs respectively finished stance in the previous stride. For the implementation presented here, the minimum (dimensionless) return time was set to be 1.4. For a runner the size of the MIT Cheetah, this corresponds to a return time of 0.35 s. This is a conservative value for the minimum return time as it is slightly longer than return times reported for galloping cheetahs [16].

Constraints on the Direction of Time

The time-at-next-apex, t_f, is the first time that the vertical velocity equals zero, *after both legs have finished their stance phases.* To ensure that both stance phases end before t_f, we impose the constraints

$$u_1 + \frac{1}{u_2} - t_f = u_1 + \frac{1 - u_3}{u_2} - \frac{u_8}{u_7} \leq 0$$

and

$$u_7 + \frac{1}{u_8} - t_f = u_6 + \frac{1 - u_8}{u_7} - \frac{u_3}{u_2} \leq 0\,.$$

Friction Cone Constraints

The friction cone of the front leg is defined by

$$\left|\frac{f_x(t)}{f_y(t)}\right| \leq \mu \text{ for all } t \in \left[u_1, u_1 + \frac{1}{u_2}\right].$$

Where μ is the static friction coefficient between the foot and the ground. When combined with (3.8) and (3.5) this becomes

$$\left|\frac{2u_2u_4}{u_3}t - \frac{(1 + u_5 + 2u_1u_2)\,u_4}{u_3}\right| \leq \mu\,. \tag{3.16}$$

Because the term between the absolute value bars is linear in t, the end-points of the stance phase are the only candidates for extrema. When $t = u_1$, (3.16) evaluates to

$$\left|\frac{u_4(u_5+1)}{u_3}\right| \leq \mu\,. \tag{3.17}$$

When $t = u_1 + 1/u_2$, it evaluates to

$$\left|\frac{u_4(u_5-1)}{u_3}\right| \leq \mu\,. \tag{3.18}$$

To avoid using the absolute value function, which is non-smooth, we rewrite (3.17) and (3.18) as four constraints. Thus, the friction cone constraints for both legs are:

$$\begin{aligned}
&\frac{u_4(u_5+1)}{u_3} - \mu \leq 0 && \frac{u_9(u_{10}+1)}{u_8} - \mu \leq 0\\
&-\frac{u_4(u_5+1)}{u_3} - \mu \leq 0 && -\frac{u_9(u_{10}+1)}{u_8} - \mu \leq 0\\
&\frac{u_4(u_5-1)}{u_3} - \mu \leq 0 && \frac{u_9(u_{10}-1)}{u_8} - \mu \leq 0\\
&-\frac{u_4(u_5-1)}{u_3} - \mu \leq 0 && -\frac{u_9(u_{10}-1)}{u_8} - \mu \leq 0.
\end{aligned} \tag{3.19}$$

Bounds on State at Next Apex

Let $y_{f,lb}$, $y_{f,ub}$, $\theta_{f,lb}$, $\theta_{f,ub}$, $\dot{\theta}_{f,lb}$, and $\dot{\theta}_{f,ub}$ be lower and upper bounds on y_f, θ_f, and $\dot{\theta}_f$ respectively. These constraints can be written as

$$\begin{aligned}
&y_{f,lb} - y_f(\mathbf{u}) \leq 0 && \theta_{f,lb} - \theta_f(\mathbf{u}) \leq 0 && \dot{\theta}_{f,lb} - \dot{\theta}_f(\mathbf{u}) \leq 0\\
&y_f(\mathbf{u}) - y_{f,ub} \leq 0 && \theta_f(\mathbf{u}) - \theta_{f,ub} \leq 0 && \dot{\theta}_f(\mathbf{u}) - \dot{\theta}_{f,ub} \leq 0.
\end{aligned} \tag{3.20}$$

Bounds on State at Mid-Stride

Analogously, the constraints bounding y_{mid} and θ_{mid} are

$$\begin{aligned}
&y_{mid,lb} - y_{mid}(\mathbf{u}) \leq 0 && \theta_{mid,lb} - \theta_{mid}(\mathbf{u}) \leq 0\\
&y_{mid}(\mathbf{u}) - y_{mid,ub} \leq 0 && \theta_{mid}(\mathbf{u}) - \theta_{mid,ub} \leq 0.
\end{aligned} \tag{3.21}$$

Bounds on Approximate CG-Print Length

This constraint is a heuristic measure to keep the planner from requesting stance phases so long that the leg would have to exceed its maximum length. The CG-print is the set of points over which the runner's center of mass passes during stance. If we approximate the height and horizontal velocity as being constant during stance, with

$$\dot{x}(t) = \dot{x}_f \tag{3.22a}$$

and

$$y(t) \approx \frac{y_{mid,lb} + y_{mid,ub}}{2}, \tag{3.22b}$$

the length of the CG-print can be approximated as

$$l_{CG} = \frac{\dot{x}_f}{u_2}, \tag{3.23}$$

since u_2^{-1} is the duration of the stance phase. To constrain the leg length at the start and end of stance to be less than one is therefore to require that

$$\left(\frac{\dot{x}_f}{2u_2}\right)^2 + \left(\frac{y_{mid,lb} + y_{mid,ub}}{2}\right)^2 - 1 \leq 0 \tag{3.24a}$$

and

$$\left(\frac{\dot{x}_f}{2u_7}\right)^2 + \left(\frac{y_{mid,lb} + y_{mid,ub}}{2}\right)^2 - 1 \leq 0. \tag{3.24b}$$

3.3.3 Objective Function

To produce bounding and pronking in the two-legged quadruped model, we have adopted an objective function that is the weighted sum of five terms. The first seeks to bring the runner to a desired forward apex-velocity and the second seeks to bring the runner to a desired apex-height, while the third and fourth attempt to match a desired pitch and pitch rate at apex respectively. The fifth seeks to minimize the hip torque that the legs must exert in order to produce the requested horizontal and vertical forces. The complete objective function is

$$J(\mathbf{u}) = \sum_{i=1}^{5} w_i J_i(\mathbf{u}),$$

where the scalars $w_1, \ldots, w_5$ are the weights on each term.

Minimize Errors in Apex State

Given a desired horizontal apex-velocity, $\dot{x}_{f,d}$, a desired apex-pitch, $\theta_{f,d}$, and a desired apex-pitch-rate, $\dot{\theta}_{f,d}$, along with the apex states defined by (3.12), the first three (unweighted) terms of the objective function are

$$J_1(\mathbf{u}) = \left(\dot{x}_f(\mathbf{u}) - \dot{x}_{f,d}\right)^2, \tag{3.25a}$$

$$J_2(\mathbf{u}) = \left(y_f(\mathbf{u}) - y_{f,d}\right)^2, \tag{3.25b}$$

$$J_3(\mathbf{u}) = \left(\theta_f(\mathbf{u}) - \theta_{f,d}\right)^2, \tag{3.25c}$$

and

$$J_4(\mathbf{u}) = \left(\dot{\theta}_f(\mathbf{u}) - \dot{\theta}_{f,d}\right)^2 \tag{3.25d}$$

Minimize Approximate Hip-Torques

Here we adopt the approximations of the height and horizontal velocities given by (3.22). Furthermore, we assume the pitch and pitch rate to be identically zero during stance. In that case the height and velocity of the hips is the same as that of the center of mass. Since the legs are massless, the hip torque of each leg must counteract any moment produced by the vertical and horizontal forces on that leg. The moment arm of the horizontal force on the leg is simply $y(t)$. The moment arm of the vertical force is the relative horizontal distance from the hip to the foot. Given the assumption of constant horizontal velocity, (3.22a), and the approximation of the CG-print, (3.23), the approximate relative horizontal distance is

$$x_{1,rel}(\mathbf{u}, t) = -\frac{\dot{x}_0 + \dot{x}_f(\mathbf{u})}{2}\left(t - u_1 - \frac{1}{2u_2}\right).$$

The approximate torques exerted by the two legs during their respective stance phases are therefore

$$\tilde{\tau}_1(\mathbf{u}, t) = p_{1,x}(\mathbf{u}, t)\frac{y_{mid,lb} + y_{mid,ub}}{2} - p_{1,y}(\mathbf{u}, t)\left[\frac{\dot{x}_0 + \dot{x}_f(\mathbf{u})}{2}\left(t - u_1 - \frac{1}{2u_2}\right)\right]$$

and

$$\tilde{\tau}_2(\mathbf{u}, t) = p_{2,x}(\mathbf{u}, t)\frac{y_{mid,lb} + y_{mid,ub}}{2} - p_{2,y}(\mathbf{u}, t)\left[\frac{\dot{x}_0 + \dot{x}_f(\mathbf{u})}{2}\left(t - u_6 - \frac{1}{2u_7}\right)\right].$$

The fourth objective function term is the sum of the integral of the torque squared for each leg

$$J_5(\mathbf{u}) = \int_{u_1}^{u_1+u_2^{-1}} \tau_1^2(\mathbf{u}, t)\, dt + \int_{u_6}^{u_6+u_7^{-1}} \tau_2^2(\mathbf{u}, t)\, dt\,.$$

Weights of the Objective Function Terms

The weights, $w_1, \ldots, w_5$, were tuned by hand to yield satisfactory performance. The values used are

$$w_1 = 1 \qquad w_3 = 2 \qquad w_5 = 3 \times 10^{-2}$$
$$w_2 = 10 \qquad w_4 = 1$$

3.3.4 Integral Control of the Stride-to-Stride Dynamics

The constraints and objective function given in the previous sections provide a non-linear program whose solution is the optimal vector of parameters, $\mathbf{u}$, for a system with the dynamics described by (3.11). However, as (3.11) contains at least two non-trivial assumptions (very small pitch-angles, and no hip-torques), it is unlikely that an actual system using the solution of (3.2) as input to leg controllers would reach the target $y_{f,d}$, $\theta_{f,d}$, and $\dot{\theta}_{f,d}$. To compensate for the steady-state error caused by these assumptions, a simple integral control is applied to the target state $\left(y, \theta, \dot{\theta}\right)_{f,d}$. To do this, $y_{f,d}$, $\theta_{f,d}$, and $\dot{\theta}_{f,d}$ in (3.25) are replaced by

$$\begin{aligned}
y'_{f,d}[k] &= y_{f,d} - k_I \sum_{i=1}^{k-1} \left(y_f[i] - y_{f,d}\right) \\
\theta'_{f,d}[k] &= \theta_{f,d} - k_I \sum_{i=1}^{k-1} \left(\theta_f[i] - \theta_{f,d}\right) \\
\dot{\theta}'_{f,d}[k] &= \dot{\theta}_{f,d} - k_I \sum_{i=1}^{k-1} \left(\dot{\theta}_f[i] - \dot{\theta}_{f,d}\right)
\end{aligned} \tag{3.26}$$

where k is the current stride number, and k_I is an integral control gain.

Because (3.26) takes the difference between the actual state and the desired state, the integral controller should not be used until the planner believes it is attaining

the desired state. During acceleration, this is not necessarily the case. Therefore, the integral controller was not activated until the magnitude of the differences between the values of y_f, θ_f, and $\dot{\theta}_f$ on consecutive strides fell below 5×10^{-2}.

3.4 Implementation

The model described in Chapter 2 and the OSHP controller described above are implemented in MATLAB® [1]. The model is simulated on a hybrid system simulator written by the author, with the MATLAB function `ode45` used for numerical integration of the continuous segments. The optimization described by equation (3.2) and §§ 3.3.2 and 3.3.3 is solved with the MATLAB function `fmincon`, which implements an interior-point method with a logarithmic barrier function. A full listing of the author's code can be found in the appendices.

Variable Bounds and Optimization Tolerance

The MATLAB function `fmincon` uses two different tolerances to determine feasibility and optimality. The OSHP controller implemented here uses the default feasibility tolerance of 1×10^{-6} for every stride. The default optimality tolerance is also 1×10^{-6}. However, when the bounds on y_f, θ_f, and $\dot{\theta}_f$ are loose, as they must be during acceleration, `fmincon` does not converge on a solution in a reasonable number of iterations (≤ 500). Since the goal during acceleration is simply to move towards the desired apex state, we loosen the optimality tolerance to 1×10^{-2} during acceleration. When the integral controller is turned on, the optimality tolerance is decreased in proportion to the maximum error in state until it reaches 2.5×10^{-5}. Simultaneously, the bounds on θ and $\dot{\theta}$ are tightened in proportion to the error in each of those states until they reach 7×10^{-4} above and below $\theta_{f,d}$ and $\dot{\theta}_{f,d}$ respectively. This ensures that `fmincon` can meet the tighter optimality tolerance.

Chapter 4

Evaluation of an Optimally Scaled Hip-Force Planning Controller in Simulation

This chapter describes evaluation of the OSHP controller described in Chapter 3 on a simulation of the two-legged quadruped model.

4.1 Leg Models Used in Simulation

Since the ability to work with disparate leg designs is one of the primary design goals for the development of OSHP, this section evaluates the performance of the same OSHP controller for runners with two different leg models. One is the leg model described in § 2.1: a massless leg with a revolute hip joint and a prismatic knee joint both of which are actuated and are subject to actuator limits. The other is an idealized leg that exerts no torque at the hip and can supply arbitrary vertical and horizontal forces, f_x and f_y. These leg models are denoted by *revolute-prismatic (R-P) leg* and *idealized leg* respectively.

4.1.1 Revolute-Prismatic Leg

Because the R-P Leg has two degrees of freedom, it can, within its actuator limits, supply any requested combination of f_x and f_y. § 2.1 gave the definition of the torque that the leg must exert on the body to meet these conditions. Together with the corresponding definition of the required axial force, this gives the following definition for the actuator force and torque exerted by the linear and prismatic actuators if those actuators subject to the limits $\tau_{min} \leq \tau_{act} \leq \tau_{max}$ and $f_{min} \leq f \leq f_{max}$:

$$\tau_{act}(t) = \max\{\tau_{min}, \min\{\tau_{max}, -l(t)[f_x(t)\cos\gamma(t) + f_y(t)\sin\gamma(t)]\}\},$$

$$f(t) = \max\{f_{min}, \min\{f_{max}, -(f_x(t)\sin\gamma(t) - f_y(t)\cos\gamma(t))\}\},$$

where l and *gamma* are the leg length and leg angle respectively (See Figure 2-1. The actual torque exerted on the body during simulation is

$$\mathbf{f}(t) = \begin{pmatrix} -\frac{\cos\gamma(t)}{l(t)}\tau_{act}(t) - f(t)\sin\gamma(t) \\ -\frac{\sin\gamma(t)}{l(t)}\tau_{act}(t) + f(t)\cos\gamma(t) \\ l(t)[f_x(t)\cos\gamma(t) + f_y(t)\sin\gamma(t)] \end{pmatrix}, \tag{4.1}$$

the first two elements of which will evaluate to $f_x(t)$ and $f_y(t)$ respectively as long as the actuator limits are not reached. The actuator limits used in simulation are approximately twice the limits specified in the design requirements for a single leg of the MIT Cheetah. Note that the third element of $\mathbf{f}$, the hip-torque, will generally be non-zero.

4.1.2 Idealized Leg

The idealized leg provides exactly the forces and torque requested by the OSHP:

$$\mathbf{f}(t) = \begin{pmatrix} f_x(t) \\ f_y(t) \\ 0 \end{pmatrix}. \tag{4.2}$$

It is not subject to force limits.

Table 4.1: Parameter values used in evaluation of an OSHP controller for the two-legged quadruped model

Parameter	Value
I^*	0.3
d_1^*	0.6
d_2^*	0.9
(τ_{min}, τ_{max})	$(-1, 1)$
(f_{min}, f_{max})	$(0, 6)$
μ	0.8
$(y_{f,lb}, y_{f,ub})$	$(0.75, 0.81)$
$(y_{mid,lb}, y_{mid,ub})$	$(0.75, 0.81)$
$(\theta_{f,lb}, \theta_{f,ub})$	$(\theta_0 - 3.5°, \theta_0 + 3.5°)$
$\left(\dot{\theta}_{f,lb}, \dot{\theta}_{f,ub}\right)$	$\left(\dot{\theta}_0 - 3.5°, \dot{\theta}_0 + 3.5°\right)$
t_{return}	1.4
$\mathbf{u}_0$	$(0, 1, 1, 1, 0, 0, 1, 1, 1, 0)^{\mathrm{T}}$

4.2 Results

The parameters used for the simulations in this section are given in Table 4.1.

4.2.1 Bounding

This subsection presents the simulated performance of the controller described in § 3.3 on two runners accelerating from rest to a bound with $\dot{x} = 2$. For a runner the size of the MIT Cheetah ($l_0 \approx 0.5\,\mathrm{m}$), this corresponds to a velocity of approximately 4.5 m/s or 10 mph. The first runner has idealized legs and the second has R-P legs.

Both runners begin with the initial state

$$\mathbf{x}_0 = \begin{pmatrix} x_0 \\ y_0 \\ \theta_0 \\ \dot{x}_0 \\ \dot{y}_0 \\ \dot{\theta}_0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0.8 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \tag{4.3}$$

and the desired apex state for both is defined by

$$\begin{pmatrix} y_{f,d} \\ \theta_{f,d} \\ \dot{x}_{f,d} \\ \dot{\theta}_{f,c} \end{pmatrix} = \begin{pmatrix} 0.8 \\ -2^\circ \\ 2 \\ -8^\circ \end{pmatrix}. \tag{4.4}$$

Figure 4-1 shows the height, pitch, forward velocity and pitch rate of both runners over the course of their acceleration. Figure 4-2 shows the state trajectories for a typical stride at steady state for both runners. Figure 4-3 shows the hip-forces and torques exerted by each leg of both runners during a typical stride at steady state.

4.2.2 Pronking

This subsection presents the simulated performance of the controller described in § 3.3 on two runners accelerating from rest to a pronk with $\dot{x} = 2$. As before, the first runner has idealized legs and the second has R-P legs. Both runners begin with the initial state given by (4.3), and the desired apex state defined by

$$\begin{pmatrix} y_{f,d} \\ \theta_{f,d} \\ \dot{x}_{f,d} \\ \dot{\theta}_{f,c} \end{pmatrix} = \begin{pmatrix} 0.8 \\ 0^\circ \\ 2 \\ 0^\circ \end{pmatrix}. \tag{4.5}$$

Figure 4-4 shows the height, pitch, forward velocity and pitch rate of both runners over the course of their acceleration. Figure 4-5 shows the state trajectories for a typical stride at steady state for both runners. Figure 4-6 shows the hip-forces and torques exerted by each leg of both runners during a typical stride at steady state.

4.3 Discussion

Figures 4-1 and 4-4 show that the OSHP controller described in Chapter 3 can coordinate the acceleration of a bounding or pronking quadruped over level ground. It does this without prescribing the footfall order of the legs or their touch-down angles. The two runners simulated have substantially different types of legs, but the same OSHP drives both of them to periodic gaits. Moreover, the simple integral-controller applied to the apex height, pitch, and pitch rate is sufficient to elminate the steady state error in those states.

Figure 4-4 clearly shows the effect of the hip-torques that the R-P legs exert on the body. Here the two runners are attempting to accelerate from rest, while keeping all other states at their initial values. Note that the forward velocity and height of both runners are the same after the first stride. This is because both leg-types impose exactly the vertical and horizontal forces requested by the OSHP controller. However, as Figures 4-4(b) and (d) show, the pitch dynamics of the two runners diverge as early as the first stride. The R-P legs must exert hip-torque in order to supply the requested combination of vertical and horizontal force. The controller tries to minimize an approximation of this torque, as described in § 3.3.3, but that term is weakly weighted in the objective function. As a result the torque is small enough to remain within the actuator limits, but large enough to impose a substantial disturbance on the body's pitch dynamics. After this pitch disturbance in the first stride, the apex state of the runner with R-P legs is different from that of the runner with idealized legs until both of them reach steady-state. This is to be expected as the two runners are faced with different initial conditions at the start of each stride.

Figures 4-4(a), (b), and(d) show that the y, θ, and $\dot{\theta}$ fluctuate during acceleration for the runner with idealized legs. In principle, this runner ought to be able to accelerate while maintaining zero pitch at all times, thereby eliminating the disturbance called by the small angle approximation on θ. There are three reasons that it does not. Firstly, the OSHP seeks to set the pitch to zero only at the next apex — at mid-stride, it merely imposes bounds on the pitch. If the controller chooses force-

profiles that result in a non-zero pitch at *any* point during the stride, the small-angle disturbance comes into play. Secondly, the error in apex-pitch, (3.25d), is only one of four terms in the objective function, (3.3.3). The force-profiles that minimize (3.3.3) may result in a non-zero intended apex-pitch. Thirdly, in the first few strides, the optimization tolerance is very high, as described in § 3.3. As a result the chosen force-profiles may only approximate the optimal profiles.

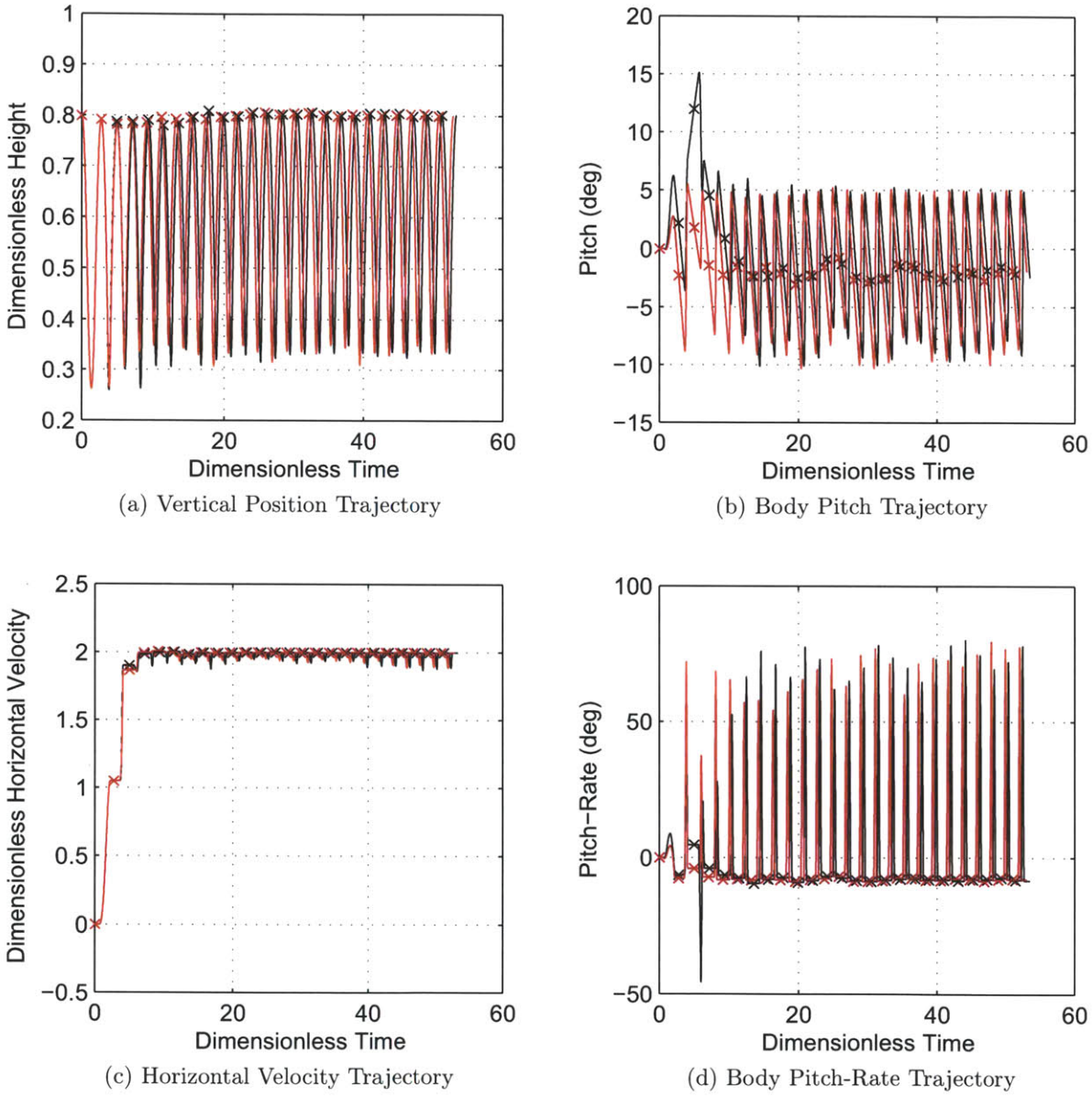
Despite their differing behavior during acceleration, the two runners reach steady-state gaits that are remarkably similar. On the one hand, some similarity is to be expected, as the two runners have the same apex-state in their final gaits. On the other hand, however, the torque exerted on the body by the P-R legs continues to be large ($> 0.3mgl_0$), while the idealized legs exert no torque.

4.4 Next Steps

The tests described in the previous section demonstrate the effectiveness of an OSHP controller for the two-legged quadruped. The next steps in this research will expand on this work in three ways: extension to a true (four-legged) quadruped model, investigation of robustness, and improvement of inter-stride control. The first involves formulating the constraints described in § 3.3.2 for a four-legged runner and investigating the effects of speed, pitch, and pitch rate on the gait chosen by the planner. The freedom of OSHP controllers to determine the touchdown order of the legs will play a much larger role for these more complex runners. It is here that OSHP can provide a significant advantage over control approaches that require the control designer to design the runner's gait as well. The second expansion of this work, investigating the robustness of OSHP controllers for both the two-legged and four-legged quadrupedal models, will entail simulated tests of the controllers performance when subjected to a variety of disturbances, including inaccurate measurements and changes in ground level.

Improvement of the inter-stride control, which is currently a simple integral controller on the desired apex state, could take many forms. More complex controllers

could be applied to the desired apex state in order to allow convergence to a wide range of apex states. A better approach, however, may be to eliminate the controller on the desired apex state and instead adjust the planner's approximate model of the body dynamics after every stride. In this way the planner would learn a model of the body dynamics that is accurate about the desired operating point.



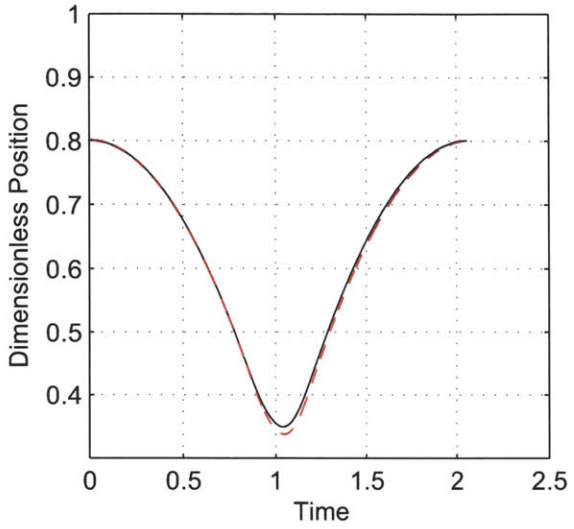
(a) Vertical Position Trajectory

(b) Body Pitch Trajectory

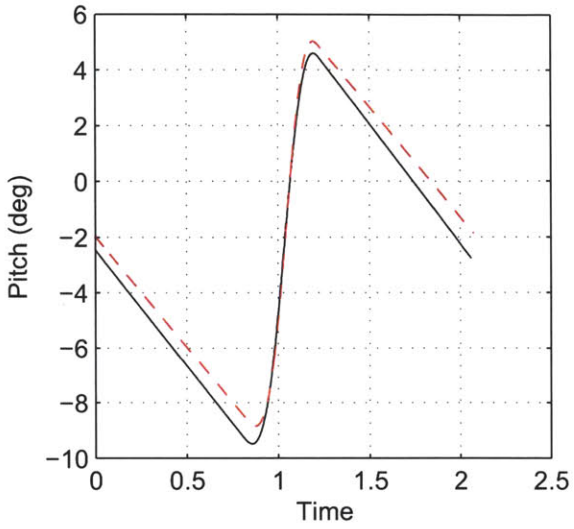
(c) Horizontal Velocity Trajectory

(d) Body Pitch-Rate Trajectory

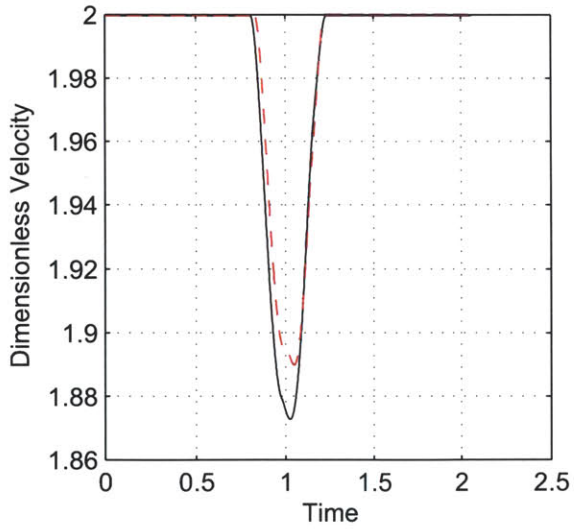
Figure 4-1: State-trajectories for runners with R-P legs (—) and idealized legs (—) during acceleration to a bound. Apex states are marked with (×) and (×) respectively.



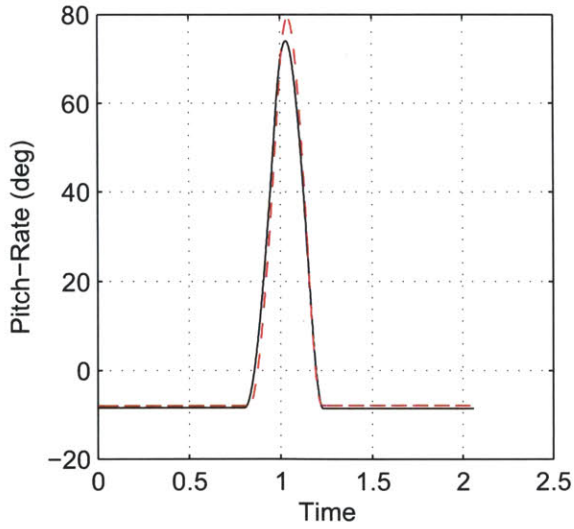
(a) Vertical Position Trajectory



(b) Body Pitch Trajectory

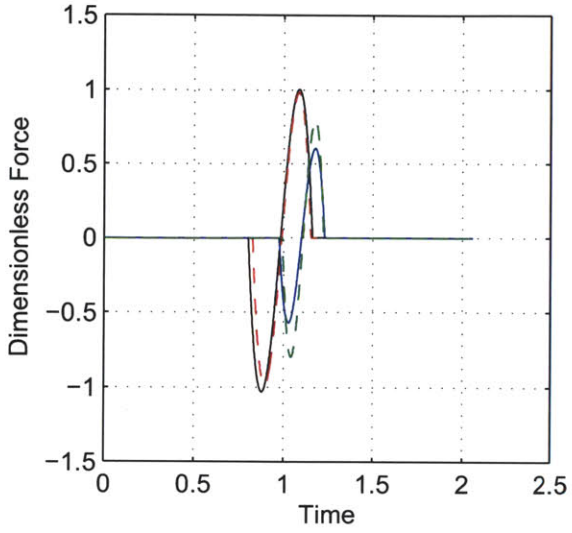


(c) Horizontal Velocity Trajectory

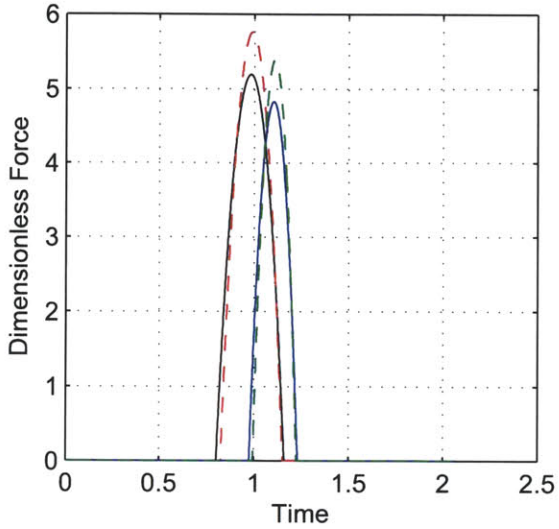


(d) Body Pitch-Rate Trajectory

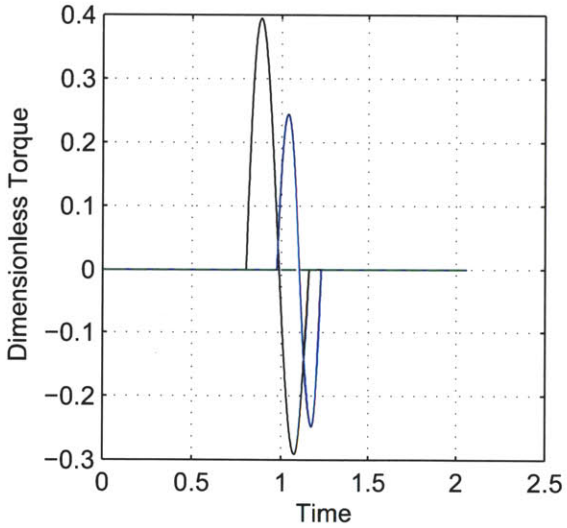
Figure 4-2: State-trajectories for runners with R-P legs (—) and idealized legs (– –) during one stride of a bound



(a) Horizontal Hip-Forces



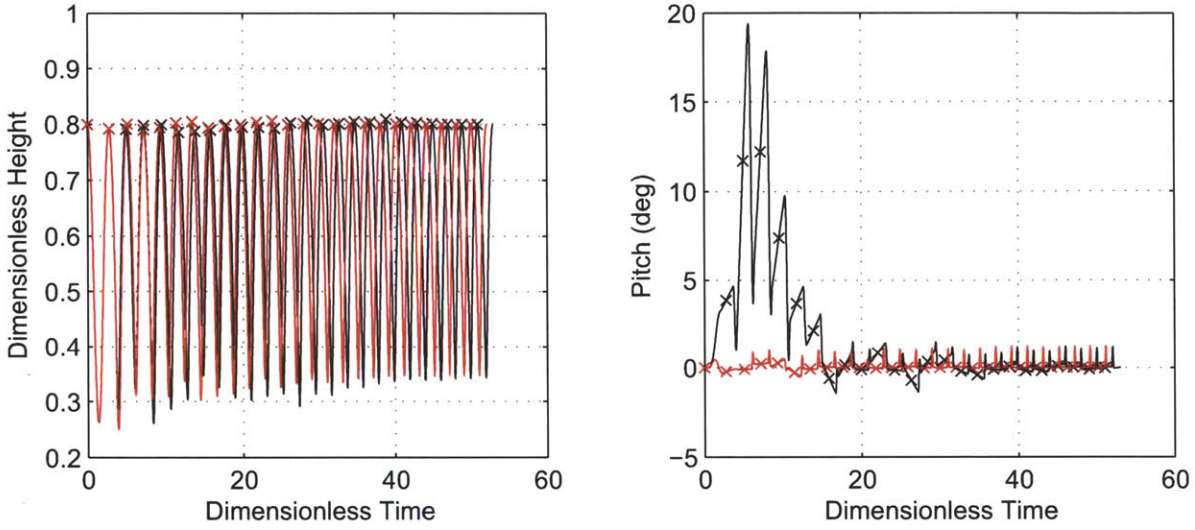
(b) Vertical Hip-Forces



(c) Hip-Torques

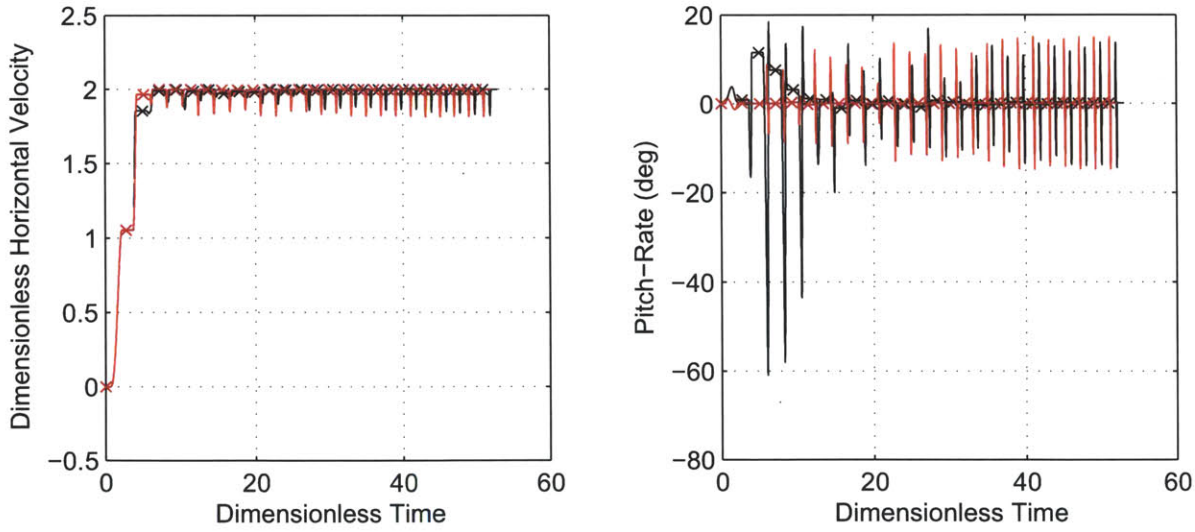
Figure 4-3: Hip-force/torque profiles during one stride of a bound for runners with

- R-P legs [Front leg (—); Hind leg (—)]
- Idealized legs [Font leg (– –); Hind leg (– –)]



(a) Vertical Position Trajectory

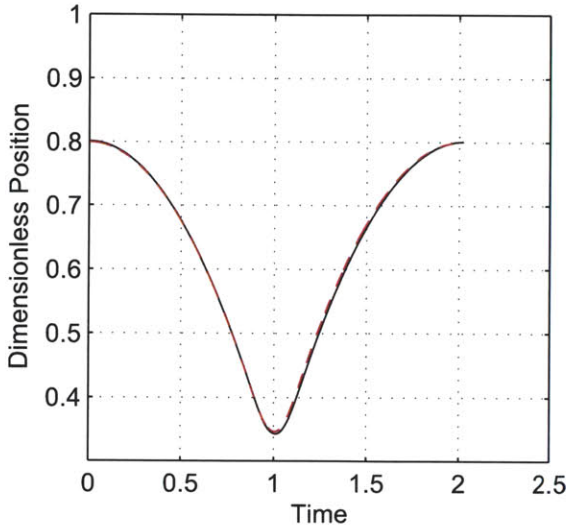
(b) Body Pitch Trajectory



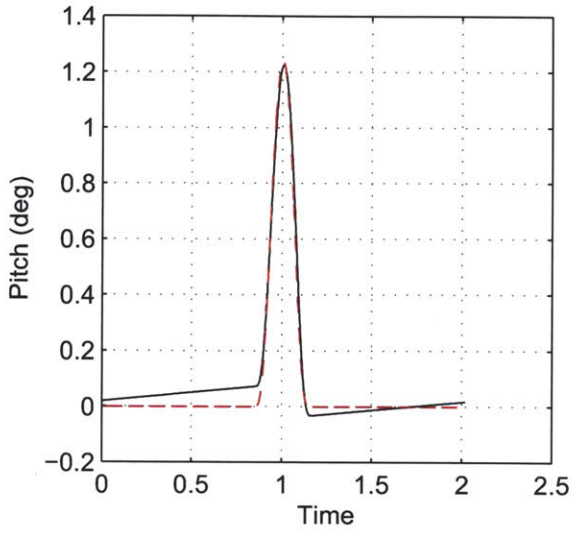
(c) Horizontal Velocity Trajectory

(d) Body Pitch-Rate Trajectory

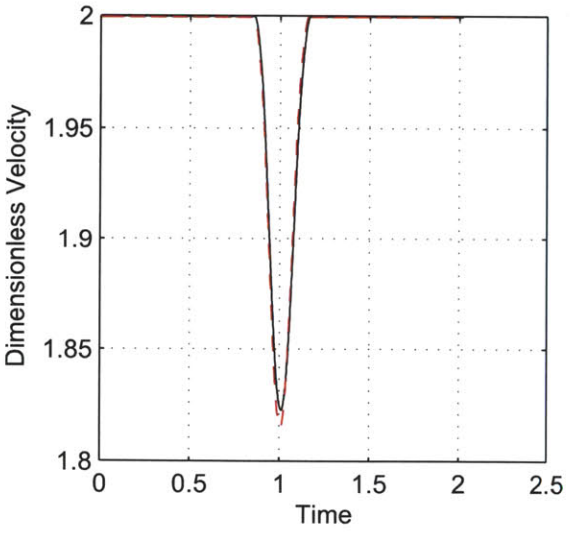
Figure 4-4: State-trajectories for runners with R-P legs (—) and idealized legs (—) during acceleration to a pronk. Apex states are marked with (×) and (×) respectively.



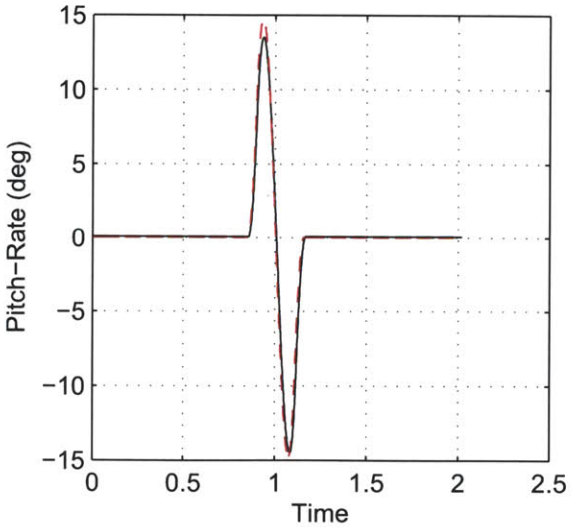
(a) Vertical Position Trajectory



(b) Body Pitch Trajectory

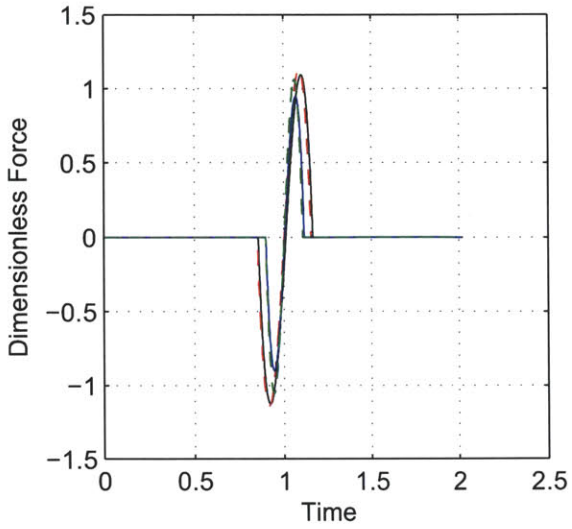


(c) Horizontal Velocity Trajectory

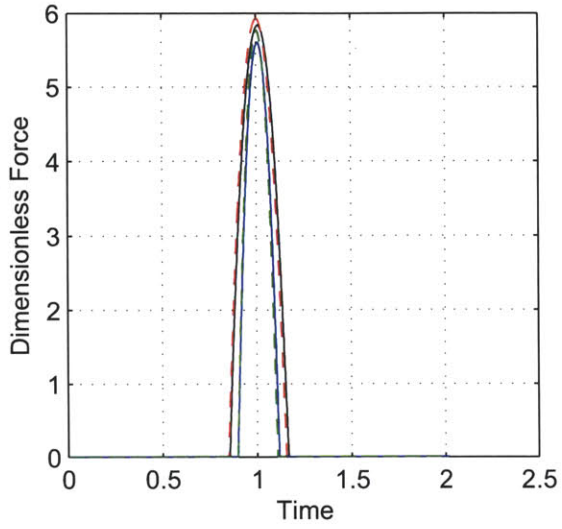


(d) Body Pitch-Rate Trajectory

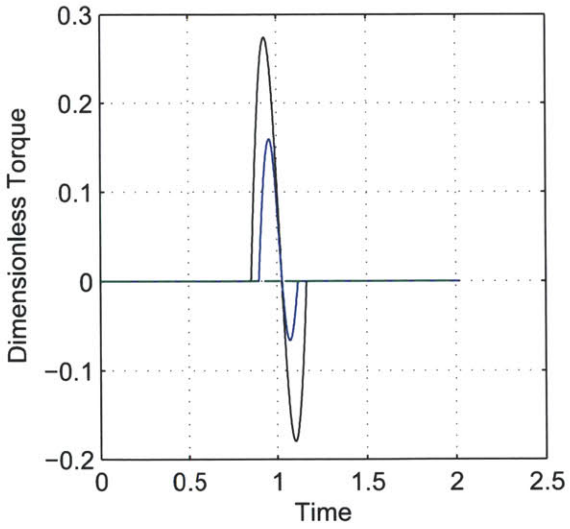
Figure 4-5: State-trajectories for runners with R-P legs (—) and idealized legs (– –) during one stride of a pronk



(a) Horizontal Hip-Forces



(b) Vertical Hip-Forces



(c) Hip-Torques

Figure 4-6: Hip-force/torque profiles during one stride of a pronk for runners with

- R-P legs [Front leg (—); Hind leg (—)]
- Idealized legs [Font leg (– –); Hind leg (– –)]

Chapter 5

Conclusion

The previous chapters have introduced and defined Optimally Scaled Hip-Force Planning as a stride-level approach to control of a running robot's body dynamics. They have also presented an implementation of OSHP for a two-legged model of a quadruped and demonstrated its effectiveness in simulation. The OSHP controller for the two-legged quadruped model is shown to be capable of producing running gaits with specified apex-states on two systems with substantially different leg models. Moreover, it does so without a pre-defined touchdown order for the legs; rather, pronking and bounding emerge as the controller seeks to set the pitch and pitch rate at the apex of the stride to zero (pronking) and non-zero (bounding) values. This research forms a first step towards the creation of a quadrupedal running controller for the MIT Cheetah project, and provides the foundation for a control framework that can be applied to diverse robotic runners.

Appendix A

Derivation of Selected States at End of Stride and Mid-Stride

This appendix presents a derivation of the expressions for the two-legged quadruped model's state at the end of the stride and at mid-stride given in equations (3.12), (3.14), and (3.15).

A.1 States at the End of Stride

As stated in § 3.3.1, the state trajectories during a stride are the sum of four or five terms:

- Constant term due to the initial state
- Linear term due to initial velocity
- Quadratic gravitational term (y only)
- One term for each applied force

Of these terms, the first three depend only on the elapsed time, t, since the start of the stride. This makes those terms simple to calculate for any t. When $t = t_f$, both legs have completed their stance phases and the remaining terms, those due to the applied forces become simple to calculate as well.

A.1.1 Calculation of y_f

The terms of y_f due to the applied forces are

$$y_{f,f_1} = \mathbb{P}_{1,y}\left(u_1 + u_2^{-1}\right) + P_{1,y}\left(u_1 + u_2^{-1}\right)\left(t_f - u_1 - u_2^{-1}\right) \tag{A.1}$$

$$y_{f,f_2} = \mathbb{P}_{2,y}\left(u_6 + u_7^{-1}\right) + P_{2,y}\left(u_6 + u_7^{-1}\right)\left(t_f - u_6 - u_7^{-1}\right), \tag{A.2}$$

where $P_{i,y}$ and $\mathbb{P}_{i,y}$ are defined by (3.13d) and (3.13f). Evaluating $P_{i,y}$ and $\mathbb{P}_{i,y}$ yields

$$P_{1,y}\left(u_1 + u_2^{-1}\right) = \frac{u_3}{u_2} \tag{A.3}$$

and

$$\mathbb{P}_{1,y}\left(u_1 + u_2^{-1}\right) = \frac{u_3}{2u_2^2}. \tag{A.4}$$

Recall that $t_f = u_3 u_2^{-1} + u_8 u_7^{-1}$. We can therefore express (A.1) and (A.2) entirely in terms of the elements of $\mathbf{u}$. This gives

$$\begin{aligned} y_{f,f_1} &= \frac{u_3}{2u_2^2} + \frac{u_3}{u_2}\left(\frac{u_3 - 1}{u_2} + \frac{u_8}{u_7} - u_1\right) \\ &= \frac{u_3}{u_2}\left(\frac{u_8}{u_7} - \frac{2u_1u_2 - 2u_3 + 1}{2u_2}\right) \end{aligned} \tag{A.5}$$

$$\tag{A.6}$$

and

$$\begin{aligned} y_{f,f_2} &= \frac{u_8}{2u_7^2} + \frac{u_8}{u_7}\left(\frac{u_8 - 1}{u_7} + \frac{u_3}{u_2} - u_6\right) \\ &= \frac{u_8}{u_7}\left(\frac{u_3}{u_2} - \frac{2u_6u_7 - 2u_8 + 1}{2u_7}\right). \end{aligned} \tag{A.7}$$

There is no linear term in the y-trajectory because $\dot{y}_0 \equiv 0$. The final expression for y_f is therefore the sum of (A.6) and (A.7) along with the constant term, y_0, and

the quadratic term, $\frac{1}{2}t_f^2$:

$$y_f = y_0 - \frac{1}{2}\left(\frac{u_3}{u_2} + \frac{u_8}{u_7}\right)^2 + \frac{u_3}{u_2}\left(\frac{u_8}{u_7} - \frac{2u_1u_2 - 2u_3 + 1}{2u_2}\right) + \frac{u_8}{u_7}\left(\frac{u_3}{u_2} - \frac{2u_6u_7 - 2u_8 + 1}{2u_7}\right)$$

A.1.2 Calculation of θ_f

Because of the small-angle assumption on θ, only the vertical forces enter into the calculation of θ_f. The small-angle assumption also yields constant moment arms for those forces. As a result, the terms of θ_f due to the applied forces are

$$\theta_{f,f_1} = \frac{d_1}{I} y_{f,f_1} \tag{A.8}$$

and

$$\theta_{f,f_2} = -\frac{d_1}{I} y_{f,f_2}. \tag{A.9}$$

The conditions at the start of the stride define the constant and linear terms of θ_f. The final expression for θ_f is therefore

$$\begin{aligned}\theta_f = \theta_0 + \dot{\theta}_0\left(\frac{u_3}{u_2} + \frac{u_8}{u_7}\right) + \frac{d_1}{I}\left(\frac{u_3}{u_2}\right)\left(\frac{u_8}{u_7} - \frac{2u_1u_2 - 2u_3 + 1}{2u_2}\right) \\ - \frac{d_2}{I}\left(\frac{u_8}{u_7}\right)\left(\frac{u_3}{u_2} - \frac{2u_6u_7 - 2u_8 + 1}{2u_7}\right)\end{aligned}$$

A.1.3 Calculation of $\dot{x}_f$

The terms of $\dot{x}_f$ due to the applied forces are

$$\dot{x}_{f,f_1} = P_{1,x}\left(u_1 + u_2^{-1}\right) \tag{A.10}$$

and

$$\dot{x}_{f,f_2} = P_{2,x}\left(u_6 + u_7^{-1}\right) \tag{A.11}$$

where $P_{i,x}$ is defined by (3.13e). Evaluating $P_{i,x}$ and adding in the constant term due to the initial conditions yields the following final expression for $\dot{x}_f$.

$$\dot{x}_f = \dot{x}_0 - \frac{u_4 u_5}{u_2} - \frac{u_9 u_{10}}{u_7}$$

A.1.4 Calculation of $\dot{\theta}_f$

The terms of $\dot{\theta}_f$ due to the applied forces are

$$\dot{\theta}_{f,f_1} = \frac{d_1}{I} P_{1,y} \left(u_1 + u_2^{-1} \right) \tag{A.12}$$

and

$$\dot{\theta}_{f,f_2} = -\frac{d_2}{I} P_{2,y} \left(u_6 + u_7^{-1} \right) \tag{A.13}$$

Evaluating $P_{i,y}$ and adding in the constant term due to the initial conditions yields the following final expression for $\dot{\theta}_f$.

$$\dot{\theta}_f = \dot{\theta}_0 - \frac{d_1}{I} \left(\frac{u_3}{u_2} \right) - \frac{d_2}{I} \left(\frac{u_8}{u_7} \right)$$

A.2 States at Mid-Stride

Because t_{mid} can fall during the stance phase of either leg, expressions for the states at t_{mid} must either be piecewise functions or functions that include Heaviside steps. Since these expressions will later be used in a second-order, gradient-based optimization, they must be continuous and twice-differentiable.

A.2.1 Calculation of y_{mid}

We use the following version of the Heaviside step function:

$$H(t) = \begin{cases} 0 & \text{if } t \leq 0, \\ 1 & \text{if } t > 0. \end{cases}$$

That function allows us to write an expression for the height of the runner's center of mass at any time, t:

$$\begin{aligned} y(t) = y_0 - \frac{t^2}{2} &+ \mathbb{P}_{1,y}(t)\, H(t-u_1) \\ &+ \left[-\mathbb{P}_{1,y}(t) + \mathbb{P}_{1,y}\left(u_1+u_2^{-1}\right) + P_{1,y}\left(u_1+u_2^{-1}\right)\left(t-u_1-u_2^{-1}\right)\right] H\left(t-u_1-u_2^{-1}\right) \\ &+ \mathbb{P}_{2,y}(t)\, H(t-u_6) \\ &+ \left[-\mathbb{P}_{2,y}(t) + \mathbb{P}_{2,y}\left(u_6+u_7^{-1}\right) + P_{2,y}\left(u_6+u_7^{-1}\right)\left(t-u_6-u_7^{-1}\right)\right] H\left(t-u_6-u_7^{-1}\right). \end{aligned} \tag{A.14}$$

The step functions and their coefficients define the height of the runner's center of mass during the different portions of the stride (before, during, and after each leg's stance phase). At t_{mid}, therefore, we have

$$\begin{aligned} y_{mid}(\mathbf{u}) = y_0 - \frac{1}{2}\left(\frac{u_3}{2u_2}+\frac{u_8}{2u_7}\right)^2 &+ \mathbb{P}_{1,y}\left(\frac{u_3}{2u_2}+\frac{u_8}{2u_7}\right) H\left(\frac{u_3}{2u_2}+\frac{u_8}{2u_7}-u_1\right) \\ &+ \left[-\mathbb{P}_{1,y}\left(\frac{u_3}{2u_2}+\frac{u_8}{2u_7}\right) + \mathbb{P}_{1,y}\left(u_1+u_2^{-1}\right)\right. \\ &\left. + P_{1,y}\left(u_1+u_2^{-1}\right)\left(\frac{u_3}{2u_2}+\frac{u_8}{2u_7}-u_1-u_2^{-1}\right)\right] H\left(\frac{u_3}{2u_2}+\frac{u_8}{2u_7}-u_1-u_2^{-1}\right) \\ &+ \mathbb{P}_{2,y}\left(\frac{u_3}{2u_2}+\frac{u_8}{2u_7}\right) H\left(\frac{u_3}{2u_2}+\frac{u_8}{2u_7}-u_6\right) \\ &+ \left[-\mathbb{P}_{2,y}\left(\frac{u_3}{2u_2}+\frac{u_8}{2u_7}\right) + \mathbb{P}_{2,y}\left(u_6+u_7^{-1}\right)\right. \\ &\left. + P_{2,y}\left(u_6+u_7^{-1}\right)\left(\frac{u_3}{2u_2}+\frac{u_8}{2u_7}-u_6-u_7^{-1}\right)\right] H\left(\frac{u_3}{2u_2}+\frac{u_8}{2u_7}-u_6-u_7^{-1}\right). \end{aligned} \tag{A.15}$$

It can be shown that the following equalities hold

$$y_{mid}(\mathbf{u})\Big|_{u_1=\left(\frac{u_3}{2u_2}+\frac{u_8}{2u_7}\right)^-} = y_{mid}(\mathbf{u})\Big|_{u_1=\left(\frac{u_3}{2u_2}+\frac{u_8}{2u_7}\right)^+} \tag{A.16a}$$

$$y_{mid}(\mathbf{u})\Big|_{u_2^{-1}=\left(\frac{u_1}{u_3}-\frac{u_8}{2u_3u_7}\right)^-} = y_{mid}(\mathbf{u})\Big|_{u_2^{-1}=\left(\frac{u_1}{u_3}-\frac{u_8}{2u_3u_7}\right)^+} \tag{A.16b}$$

$$y_{mid}(\mathbf{u})\Big|_{u_3=\left(2u_1u_2-\frac{u_2u_8}{2u_7}\right)^-} = y_{mid}(\mathbf{u})\Big|_{u_3=\left(2u_1u_2-\frac{u_2u_8}{2u_7}\right)^+} \tag{A.16c}$$

$$y_{mid}(\mathbf{u})\Big|_{u_7^{-1}=\left(\frac{u_1}{u_8}-\frac{u_3}{2u_8u_2}\right)^-} = y_{mid}(\mathbf{u})\Big|_{u_7^{-1}=\left(\frac{u_1}{u_8}-\frac{u_3}{2u_8u_2}\right)^+} \tag{A.16d}$$

$$y_{mid}(\mathbf{u})\Big|_{u_8=\left(2u_1u_7-\frac{u_7u_3}{2u_2}\right)^-} = y_{mid}(\mathbf{u})\Big|_{u_8=\left(2u_1u_7-\frac{u_7u_3}{2u_2}\right)^+} \tag{A.16e}$$

$$y_{mid}(\mathbf{u})\Big|_{u_1=\left(\frac{u_3-2}{2u_2}+\frac{u_8}{2u_7}\right)^-} = y_{mid}(\mathbf{u})\Big|_{u_1=\left(\frac{u_3-2}{2u_2}+\frac{u_8}{2u_7}\right)^+} \tag{A.16f}$$

$$y_{mid}(\mathbf{u})\Big|_{u_2^{-1}=\left(\frac{u_1}{u_3-2}-\frac{u_8}{2u_3u_7-4}\right)^-} = y_{mid}(\mathbf{u})\Big|_{u_2^{-1}=\left(\frac{u_1}{u_3-2}-\frac{u_8}{2u_3u_7-4}\right)^+} \tag{A.16g}$$

$$y_{mid}(\mathbf{u})\Big|_{u_3=\left(2u_1u_2-\frac{u_2u_8}{2u_7}+2\right)^-} = y_{mid}(\mathbf{u})\Big|_{u_3=\left(2u_1u_2-\frac{u_2u_8}{2u_7}+2\right)^+} \tag{A.16h}$$

$$y_{mid}(\mathbf{u})\Big|_{u_7^{-1}=\left(\frac{u_1}{u_8}-\frac{u_3-2}{2u_8u_2}\right)^-} = y_{mid}(\mathbf{u})\Big|_{u_7^{-1}=\left(\frac{u_1}{u_8}-\frac{u_3-2}{2u_8u_2}\right)^+} \tag{A.16i}$$

$$y_{mid}(\mathbf{u})\Big|_{u_8=\left(2u_1u_7-\frac{u_7(u_3-2)}{2u_2}\right)^-} = y_{mid}(\mathbf{u})\Big|_{u_8=\left(2u_1u_7-\frac{u_7(u_3-2)}{2u_2}\right)^+} \tag{A.16j}$$

$$y_{mid}(\mathbf{u})\Big|_{u_6=\left(\frac{u_3}{2u_2}+\frac{u_8}{2u_7}\right)^-} = y_{mid}(\mathbf{u})\Big|_{u_6=\left(\frac{u_3}{2u_2}+\frac{u_8}{2u_7}\right)^+} \tag{A.16k}$$

$$y_{mid}(\mathbf{u})\Big|_{u_2^{-1}=\left(\frac{u_6}{u_3}-\frac{u_8}{2u_3u_7}\right)^-} = y_{mid}(\mathbf{u})\Big|_{u_2^{-1}=\left(\frac{u_6}{u_3}-\frac{u_8}{2u_3u_7}\right)^+} \tag{A.16l}$$

$$y_{mid}(\mathbf{u})\Big|_{u_3=\left(2u_6u_2-\frac{u_2u_8}{2u_7}\right)^-} = y_{mid}(\mathbf{u})\Big|_{u_3=\left(2u_6u_2-\frac{u_2u_8}{2u_7}\right)^+} \tag{A.16m}$$

$$y_{mid}(\mathbf{u})\Big|_{u_7^{-1}=\left(\frac{u_6}{u_8}-\frac{u_3}{2u_8u_2}\right)^-} = y_{mid}(\mathbf{u})\Big|_{u_7^{-1}=\left(\frac{u_6}{u_8}-\frac{u_3}{2u_8u_2}\right)^+} \tag{A.16n}$$

$$y_{mid}(\mathbf{u})\Big|_{u_8=\left(2u_6u_7-\frac{u_7u_3}{2u_2}\right)^-} = y_{mid}(\mathbf{u})\Big|_{u_8=\left(2u_6u_7-\frac{u_7u_3}{2u_2}\right)^+} \tag{A.16o}$$

$$y_{mid}(\mathbf{u})\Big|_{u_1=\left(\frac{u_8-2}{2u_7}+\frac{u_3}{2u_2}\right)^-} = y_{mid}(\mathbf{u})\Big|_{u_1=\left(\frac{u_8-2}{2u_7}+\frac{u_3}{2u_2}\right)^+} \tag{A.16p}$$

$$y_{mid}(\mathbf{u})\Big|_{u_7^{-1}=\left(\frac{u_6}{u_8-2}-\frac{u_3}{2u_8u_2-4}\right)^-} = y_{mid}(\mathbf{u})\Big|_{u_7^{-1}=\left(\frac{u_6}{u_8-2}-\frac{u_3}{2u_8u_2-4}\right)^+} \tag{A.16q}$$

$$y_{mid}(\mathbf{u})\Big|_{u_8=\left(2u_6u_7-\frac{u_7u_3}{2u_2}+2\right)^-} = y_{mid}(\mathbf{u})\Big|_{u_8=\left(2u_6u_7-\frac{u_7u_3}{2u_2}+2\right)^+} \tag{A.16r}$$

$$y_{mid}(\mathbf{u})\Big|_{u_2^{-1}=\left(\frac{u_6}{u_3}-\frac{u_8-2}{2u_3u_7}\right)^-} = y_{mid}(\mathbf{u})\Big|_{u_2^{-1}=\left(\frac{u_6}{u_3}-\frac{u_8-2}{2u_3u_7}\right)^+} \tag{A.16s}$$

$$y_{mid}(\mathbf{u})\Big|_{u_3=\left(2u_6u_2-\frac{u_2(u_8-2)}{2u_7}\right)^-} = y_{mid}(\mathbf{u})\Big|_{u_3=\left(2u_6u_2-\frac{u_2(u_8-2)}{2u_7}\right)^+} \tag{A.16t}$$

The corresponding equalities for the first and second partial derivatives of y with respect to the elements of $\mathbf{u}$ can also be shown to hold. Therefore, the expression for y_{mid} given in (3.14) is continuous and twice-differentiable.

A.2.2 Calculation of θ_{mid}

We can use Heaviside steps in the same manner to define the runner's pitch at t_{mid}:

$$\begin{aligned}
\theta_{mid}(\mathbf{u}) = {} & \theta_0 + \dot{\theta}_0\left(\frac{u_3}{2u_2}+\frac{u_8}{2u_7}\right) + \frac{d_1}{I}\mathbb{P}_{1,y}\left(\frac{u_3}{2u_2}+\frac{u_8}{2u_7}\right)H\left(\frac{u_3}{2u_2}+\frac{u_8}{2u_7}-u_1\right) \\
& + \frac{d_1}{I}\left[-\mathbb{P}_{1,y}\left(\frac{u_3}{2u_2}+\frac{u_8}{2u_7}\right)+\mathbb{P}_{1,y}\left(u_1+u_2^{-1}\right)\right. \\
& \left. + P_{1,y}\left(u_1+u_2^{-1}\right)\left(\frac{u_3}{2u_2}+\frac{u_8}{2u_7}-u_1-u_2^{-1}\right)\right]H\left(\frac{u_3}{2u_2}+\frac{u_8}{2u_7}-u_1-u_2^{-1}\right) \\
& + \frac{d_2}{I}\mathbb{P}_{2,y}\left(\frac{u_3}{2u_2}+\frac{u_8}{2u_7}\right)H\left(\frac{u_3}{2u_2}+\frac{u_8}{2u_7}-u_6\right) \\
& + \frac{d_2}{I}\left[-\mathbb{P}_{2,y}\left(\frac{u_3}{2u_2}+\frac{u_8}{2u_7}\right)+\mathbb{P}_{2,y}\left(u_6+u_7^{-1}\right)\right. \\
& \left. + P_{2,y}\left(u_6+u_7^{-1}\right)\left(\frac{u_3}{2u_2}+\frac{u_8}{2u_7}-u_6-u_7^{-1}\right)\right]H\left(\frac{u_3}{2u_2}+\frac{u_8}{2u_7}-u_6-u_7^{-1}\right).
\end{aligned} \tag{A.17}$$

Comparing (3.14) and (3.15) we find that θ_{mid} is a linear combination of the terms of y_{mid} plus an additional linear term. Because of this, the continuity and differentiability of $y_{mid}(\mathbf{u})$ imply the continutity and differentiability of $\theta_{mid}(\mathbf{u})$.

Appendix B

Code Listing for MATLAB Implementation

B.1 Model Classes

B.1.1 State-Space Model

```
classdef ssModel < handle
    properties (Abstract)
        x
        u
        params
    end
    properties
        t = 0;
    end
    methods
        function newObj = clone(this)
            mcThis = metaclass(this);
            mpCell = mcThis.Properties;
            constructor = str2func(mcThis.Name);
            newObj = constructor();
            for i = 1:length(mpCell)
                newObj.(mpCell{i}.Name) = this.(mpCell{i}.Name);
            end
        end
    end
end
```

B.1.2 Hybrid System Model

```
classdef hybridModel < ssModel
    properties (Abstract)
        phaseList
        stateLabels
        stateScales
    end
    properties
        controller = [];
    end
    methods (Abstract)
        % Needs to configure model for next phase and accumulate history
        updateModel(obj,Tout,Xout,TE,XE,IE)
    end
    methods
        function applyControl(obj)
            obj.u = obj.controller.controlLaw(obj);
        end
        function simulatePhase(obj,historyObj)
            if nargin == 2 && historyObj.checkModel(obj)
                haveHistoryObj = true;
            else
                haveHistoryObj = false;
            end

            phaseClass = class(obj.params.phase);
            % Check that the phase is valid and initialize fcn handles
            switch phaseClass
                case 'char'
                    if ~any(cellfun(@(y) strcmp(y,obj.params.phase),...
                            obj.phaseList))
                        disp(['"' obj.params.phase ...
                              '" is not a recognized phase for this model']);
                        return
                    end
                    dynamicsFun = str2func([obj.params.phase 'Dynamics']);
                    eventsFun = str2func([obj.params.phase 'Events']);
                case 'logical'
                    dynamicsFun = str2func([class(obj) '.dynamics']);
                    eventsFun = str2func([class(obj) '.eventFun']);
                otherwise
                    disp('Unsupported phase class');
            end
            % Set control inputs
```

```
            if ~isempty(obj.controller)
                if strcmp(obj.controller.flavor,'phase')
                    obj.applycontrol;
                end
            end
            % Initialize variables
            refine = 10;

            %Configure solver
            options = odeset('RelTol',1e10,'AbsTol',1e9, ...
                             'OutputFcn',@odeTimeout,'Events',eventsFun, ...
                             'Refine',refine);

            % Integrate until terminal event
            [Tout,Xout,TE,XE,IE] = ode45(dynamicsFun,[obj.t obj.t+1e1], ...
                                         obj.x,options,obj.u,obj.params);

            if numel(TE)==0
                disp('ERROR: No event found');
                disp(obj.x)
                TE = Tout(end);
                XE = Xout(end,:);
                IE = 7;
            end
            %Configure model for next phase
            if haveHistoryObj
                obj.updateModel(TE,XE,IE,historyObj);
            else
                obj.updateModel(TE,XE,IE)
            end
        end
    end
end
```

B.1.3 Running System Model

```
classdef runningModel < hybridModel
    properties
        n_animation_nodes = 1;
        status = 'normal';
    end
    methods
        function reset(obj,x0,u0,t0)
            obj.status = 'normal';
            obj.x = x0;
```

```
    obj.u = u0;
    obj.t = t0;
end
function simulateStride(obj,historyObj)
    % Check status
    if strcmp(obj.status,'failure')
        disp('Runner has fallen or flipped over!');
        disp('It must be reset before taking a stride.');
        return
    else
        obj.status = 'normal';
    end
    % Set control inputs if necessary
    if ~isempty(obj.controller)
        if strcmp(obj.controller.flavor,'a2a')
            obj.applyControl;
        end
    end
    if nargin == 2
        % Simulate until end of stride
        while strcmp(obj.status,'normal')
            obj.simulatePhase(historyObj);
        end
    else
        % Simulate until end of stride
        while strcmp(obj.status,'normal')
            obj.simulatePhase;
        end
    end
    % Display result message
    if strcmp(obj.status,'finishedStride')
        obj.resetLegs;
        obj.status = 'normal';
        dispDebug('stride','Done with one stride.')
    elseif strcmp(obj.status,'failure')
        dispDebug('stride','Stride ended in failure!')
    elseif strcmp(obj.status,'error')
        dispDebug('stride','Error during stride!')
    else
        disp(['"' obj.status '" is not a valid model status!'])
    end
end
function X = getX(obj,xInd,uInd)
    % GETX Extracts a combination of states and actions
    %   X = GETX(OBJ,XIND,UIND) returns a column vector consisting
```

```
            %   of the elements of x specified by XIND followed by the
            %   elements of u specified by UIND.
            %
            %   6/10/10
            %   MIT Biomimetic Robotics Lab
            %   A. K. Valenzuela
            X = [obj.x(xInd);obj.u(uInd)];
        end
        function setX(obj,X,xInd,uInd)
            % SETX Updates a combination of states and actions
            %   SETX(OBJ,X,XIND,UIND) assigns the elements in X to the
            %   elements of x specified by xInd and then to the values of u
            %   specified by uInd.
            %
            %   6/10/10
            %   MIT Biomimetic Robotics Lab
            %   A. K. Valenzuela
            obj.x(xInd) = X(1:length(xInd));
            obj.u(uInd) = X(length(xInd)+1:end);
        end
    end
end
```

B.1.4 Planar Rigid Body (Spine) Runner

```
classdef planarRigidBodyRunner < runningModel
    properties
        x = [0; 0; 0; 0; 0; 0];
        u = [];
        params = [];
        phaseList = [];
        stateLabels = {'x', 'y', 'theta', 'xdot', 'ydot', 'thetadot'};
        stateScales = [1, 1, 180/pi, 1, 1, 180/pi];
        connectionMatrix = [];
    end
    methods (Static)
        xDot = dynamics(t,x,u,params);
        prms = defaultParameters;
        [value, isterminal, direction] = eventFun(t, x, u, params);
    end
    methods
        function this = planarRigidBodyRunner(x_in, u_in, params_in, ...
                                               controller_in, varargin)
            optargin = size(varargin,2);
            stdargin = nargin - optargin;
```

```
if stdargin < 4
    error('planarRigidBodyRunner:ArgsIn', ...
          ['The constructor for the planarRigidBody class ' ...
           'requires at least four arguments']);
else
    this.x = x_in;
    this.u = u_in;
    this.params = planarRigidBodyRunner.defaultParameters;
    if isstruct(params_in)
        params_in_names = fieldnames(params_in);
        params_in_ind = isfield(this.params, params_in_names);
        for ii = 1:length(params_in_names)
            if params_in_ind(ii)
                this.params.(params_in_names{ii}) = ...
                    params_in.(params_in_names{ii});
            end
        end
    end
    this.controller = controller_in;
end
if optargin > 0 % All inputs after first three should be legs
    for ii = 1:optargin
        if isa(varargin{ii}, 'leg')
            this.params.legs = [this.params.legs; varargin{ii}];
            this.params.angle_leg_attach_pts = ...
                [this.params.angle_leg_attach_pts; ...
                 this.params.legs(end).attach_pt.angle];
            this.params.dist_leg_attach_pts = ...
                [this.params.dist_leg_attach_pts; ...
                 this.params.legs(end).attach_pt.distance];
            this.params.leg_inputs_partition = ...
                [this.params.leg_inputs_partition; ...
                 this.params.leg_inputs_partition(end) + ...
                    this.params.legs(end).n_inputs];
            if any(strcmp('foot_position', ...
                          properties(varargin{ii})))
                this.connectionMatrix = ...
                    [this.connectionMatrix; ...
                        [1, this.n_animation_nodes + 1; ...
                         this.n_animation_nodes + 1, ...
                            this.n_animation_nodes + 2;
                         this.n_animation_nodes + 2, ...
                             this.n_animation_nodes + 1;
                         this.n_animation_nodes + 1, 1]];
                this.n_animation_nodes = this.n_animation_nodes+2;
```

```
                else
                    this.connectionMatrix = ...
                        [this.connectionMatrix; ...
                            [1, this.n_animation_nodes + 1;
                             this.n_animation_nodes + 1, 1]];
                    this.n_animation_nodes = this.n_animation_nodes+1;
                end
            else
                error('planarRigidBodyRunner:OptArgsClass', ...
                      ['All optional arguments to the ' ...
                       'planarRigidBody constructor must ' ...
                       'be leg objects');
            end
        end
    end
    this.params.phase = false(1, length(this.params.legs));
end

function addLegs(this,varargin)
    for ii = 1:length(varargin)
        if isa(varargin{ii}, 'leg')
            this.params.legs = [this.params.legs; varargin{ii}];
            this.params.angle_leg_attach_pts = ...
                [this.params.angle_leg_attach_pts; ...
                 this.params.legs(end).attach_pt.angle];
            this.params.dist_leg_attach_pts = ...
                [this.params.dist_leg_attach_pts; ...
                 this.params.legs(end).attach_pt.distance];
            if any(strcmp('foot_position', properties(varargin{ii})))
                this.connectionMatrix = ...
                    [this.connectionMatrix; ...
                        [1, this.n_animation_nodes + 1; ...
                         this.n_animation_nodes + 1, ...
                             this.n_animation_nodes + 2; ...
                         this.n_animation_nodes + 2, ...
                             this.n_animation_nodes + 1; ...
                         this.n_animation_nodes + 1, 1]];
                this.n_animation_nodes  = this.n_animation_nodes + 2;
            else
                this.connectionMatrix = ...
                    [this.connectionMatrix; ...
                        [1, this.n_animation_nodes + 1; ...
                         this.n_animation_nodes + 1, 1]];
                this.n_animation_nodes  = this.n_animation_nodes + 1;
            end
```

```
        params.phase = [params.phase, false];
    else
        error('planarRigidBodyRunner.addLegs:ArgsClass', ...
            'All arguments to the addLegs method must be leg objects');
    end
    end
end

function rmLegs(this,indices)
    this.params.legs(indices) = [];
    this.params.angle_legs_attach_pts(indices) = [];
    this.params.dist_legs_attach_pts(indices) = [];
    this.params.phase(indices) = [];
    cur_ind = 2;
    connectionMatrix_ind = [];
    for ii = 1:length(this.params.legs)
        if any(strcmp('foot_position', ...
                    properties(this.params.legs(ii))))
            if any(ii == indices)
                connectionMatrix_ind = [connectionMatrix_ind, ...
                                        cur_ind:cur_ind+3];
            end
            cur_ind = cur_ind + 4;
        else
            if any(ii == indices)
                connectionMatrix_ind = [connectionMatrix_ind, ...
                                        cur_ind:cur_ind+1];
            end
            cur_ind = cur_ind + 2;
        end
    end
    this.connectionMatrix(connectionMatrix_ind, :) = [];
end

        function updateModel(this,TE,XE,IE,historyObj)
                [TE, ITE] = sort(TE);
                if nargin == 5
        for ii = 1:length(this.params.legs)
            if any(strcmp('foot_position', ...
                        properties(this.params.legs(ii))))
                if isempty(this.params.xyT)
                    this.params.xyT = NaN(2, length(this.params.legs));
                end
                if this.params.phase(ii)
                    this.params.xyT(:,ii) = ...
```

```
                    this.params.legs(ii).foot_position;
                else
                    this.params.xyT(:,ii) = NaN;
                end
            end
        end
                    historyObj.addPhaseData(this.x, this.u, [this.t, TE(1)], ...
                            this.params);
                end
                this.t = TE(1);
                this.x = XE(ITE(1),:)';
                ie = IE(~(TE TE(1)));

                %Determine next phase
    n_legs = length(this.params.legs);
    if all(ie <= n_legs) % Not a fall or end of stride
        event = false(1, n_legs);
        event(ie) = true;
        % Update leg data
        this.updateLegs(event);
        % Update phase
        this.params.phase = xor(this.params.phase,event);
        this.status = 'normal';
    elseif ie == n_legs+1   % End of stride
        this.status = 'finishedStride';
        dispDebug('planarRigidBodyRunner','Done with one stride')
    else                    % Fall
        this.status = 'failure';
        dispDebug('planarRigidBodyRunner', ...
                  'Runner hit the ground during aerial phase!')
    end
        end

function updateLegs(this, event)
    indices = 1:length(this.params.legs);
    for ii = indices(event)
        this.params.legs(ii).update(this.t, this.x, ...
            this.u(this.params.leg_inputs_partition(ii):...
                   this.params.leg_inputs_partition(ii+1)-1), ...
            this.params.phase(ii));
    end
end

function resetLegs(this)
    for ii = 1:length(this.params.legs)
```

```
            this.params.legs(ii).has_hit = false;
            this.params.legs(ii).update(this.t);
        end
    end

        function setParam(this, name, value)
            if isfield(this.params, name)
                this.params.(name) = value;
            else
                disp(['''' name ''' is not a valid parameter name'])
            end
        end
        function value = getParam(this, name)
            if isfield(this.params, name)
                value = this.params.(name);
            else
                disp(['''' name ''' is not a valid parameter name'])
            end
    end

        function nodes = nodeLocations(this, t, X, u, xyT)
    % Node 1: CG
    % Node 2   end: Leg attachement points

    % Put input arguments into conventional form
    t = t'; X = X'; u = u';
    % Constants
    n_t = size(X, 2);
    n_legs = length(this.params.legs);

    % CG position
    cg_node = X(1:2,:);

    % Calculate offsets of leg attachment points from CG
    abs_leg_angles = repmat(X(3,:), n_legs, 1) + ...
                     repmat(this.params.angle_leg_attach_pts, 1, n_t);
    legs_x_offset = ...
                    repmat(this.params.dist_leg_attach_pts,1,n_t).* ...
                    cos(abs_leg_angles);
    legs_y_offset = ...
                    repmat(this.params.dist_leg_attach_pts,1,n_t).* ...
                    sin(abs_leg_angles);

    % Interleave offsets and add to CG position
    leg_nodes = repmat(cg_node, n_legs, 1) + ...
```

```
                reshape([legs_x_offset'; legs_y_offset'], n_t, 2*n_legs)';

    if ~isempty(xyT)
        % Calculate positions of feet
        swing_ind = isnan(xyT);
        foot_nodes = NaN(2*length(this.params.legs), length(X));

        % Position of stance feet given by xyT
        foot_nodes(~swing_ind) = xyT(~swing_ind);

        % Position of swing feet given by legs
        for ii = length(this.params.legs): 1:1
            if any(strcmp('footPosition', ...
                        methods(this.params.legs(ii))))
                cur_leg_swing_ind = squeeze(swing_ind(1, ii, :));
                cur_foot_node = foot_nodes(2*ii-1:2*ii, :);
                if any(cur_leg_swing_ind)
                    cur_foot_node(:, cur_leg_swing_ind) = ...
                      this.params.legs(ii).footPosition( ...
                        t(cur_leg_swing_ind), ...
                        X(:, cur_leg_swing_ind), ...
                        u(this.params.leg_inputs_partition(ii):...
                          this.params.leg_inputs_partition(ii+1)-1, ...
                          cur_leg_swing_ind));
                end
                if ii == length(this.params.legs)
                    leg_nodes = [leg_nodes(1:2*ii, :); cur_foot_node];
                else
                    leg_nodes = [leg_nodes(1:2*ii, :); ...
                        cur_foot_node; leg_nodes(2*(ii+1)-1:end, :)];
                end
            end
        end
    end
    % Concatenate node data for output
    nodes = [cg_node; leg_nodes];
end

        function NGRF = groundRxns(this, t, X, u, params_in)
    % Constants
    n_t = size(X, 1);
    n_legs = length(this.params.legs);

    leg_forces = zeros(3*n_legs, n_t);
    for ii = 1:n_legs
```

```
            leg_forces(3*ii-2:3*ii, :) = params_in.legs(ii).forces(t, X,...
                u(this.params.leg_inputs_partition(ii):...
                    this.params.leg_inputs_partition(ii+1)-1), ...
                params_in.phase(ii));
        end
        NGRF.abs = sqrt(leg_forces(1:3:3*n_legs,:).^2 + ...
                        leg_forces(2:3:3*n_legs,:).^2);
        NGRF.xComp = leg_forces(1:3:3*n_legs,:);
        NGRF.yComp = leg_forces(2:3:3*n_legs,:);
    end

    function hip_torques = hipTorques( this, t, X, u, params_in)
        % Constants
        n_t = length(t);
        n_legs = length(this.params.legs);

        leg_forces = zeros(3*n_legs, n_t);
        hip_torques = zeros(n_legs, n_t);
        for ii = 1:n_legs
            leg_forces(3*ii-2:3*ii, :) = ...
                params_in.legs(ii).forces(t, X,...
                    u(:,this.params.leg_inputs_partition(ii):...
                        this.params.leg_inputs_partition(ii+1)-1), ...
                    params_in.phase(ii));
            hip_torques(ii, :) = leg_forces(3*ii, :);
        end
    end
end
end

function prms = defaultParameters
    prms.lstar = 1;
    prms.mu_s = 1;
    prms.phase = [];
    prms.legs = [];
    prms.angle_leg_attach_pts = [];
    prms.dist_leg_attach_pts = [];
    prms.leg_inputs_partition = 1;
    prms.xyT = [];
end

function xDot = dynamics(t,x,u,params)
q = x(1:3);
qDot = x(4:6);
a_legs = params.angle_leg_attach_pts;
d_legs = params.dist_leg_attach_pts;
```

```
f_legs = zeros(3,length(params.legs));
for ii = 1:length(params.legs)
    f_legs(:,ii) = ...
        params.legs(ii).forces(t, x,...
                               u(params.leg_inputs_partition(ii):...
                                 params.leg_inputs_partition(ii+1)-1), ...
                                 params.phase(ii));
end
% disp(f_legs(2,:));
xDot = [qDot;
        sum(f_legs(1:2, :),2)    [0; 1];
        1/params.lstar *...
            sum(d_legs.*...
                sum([ sin(q(3)+a_legs), cos(q(3)+a_legs)].* ...
                    f_legs(1:2, :)',2)) + ...
                sum(f_legs(3, :),2)];
end


function [values, isterminal, direction] = eventFun(t, x, u, params)
    n_legs = length(params.legs);
    values = zeros(2*n_legs + 2, 1);
    isterminal = ones(2*n_legs + 2, 1);
    direction =  ones(2*n_legs + 2, 1);

    % Leg events
    for ii = 1:n_legs
        [values(ii), isterminal(ii), direction(ii)] = ...
            params.legs(ii).legEvent(t, x,...
                                     u(params.leg_inputs_partition(ii):...
                                       params.leg_inputs_partition(ii+1)-1),...
                                     params.phase(ii));
    end

    % End of stride event
    if all( cat(1,params.legs.has_hit) ) && all( ~params.phase )
        values(n_legs+1) = x(5);
    else
        values(n_legs+1) = 1;
    end

    % Fall event (CG hits ground)
    values(n_legs+2) = x(2);
    values(n_legs+3:end) = x(2) + params.dist_leg_attach_pts .* sin(x(3) + ...
                           params.angle_leg_attach_pts);
end
```

B.1.5 Leg Models

```
classdef leg < handle_with_clone
    properties
        attach_pt = struct('distance', 0, 'angle', 0);
        t0 = Inf;
        has_hit = false;
        ground = [];
    end
    methods (Abstract)
        f = forces(this, t, x_body);
        [values, isterminal, direction] = legEvent(this, t, x, u, stance);
    end
    methods
        function this = leg(attach_dist, attach_angle, ground_in, t0_in)
            if nargin > 1
                this.attach_pt.distance = attach_dist;
                this.attach_pt.angle = attach_angle;
            end
            if nargin > 2
                this.ground = ground_in;
            else
                this.ground = ground_level();
            end
            if nargin > 3
                this.t0 = t0_in;
            end
        end

        function t0_out = getT0(this)
            t0_out = this.t0;
        end

        function setT0(this, t0_in)
            this.t0 = t0_in;
        end

        function err = update(this, t, x, u, stance)
            err = 0;
            if nargin > 2
                if ~stance
                    this.has_hit = true;
                    this.t0 = t;
                end
            else
```

```
                this.t0 = Inf;
            end
        end
    end
end

classdef leg_rAct_pAct < leg
    properties
        l_leg_min = 0.5;
        l_leg_max = 1;
        gamma_max = 60*pi/180;
        gamma_min =  60*pi/180;
        f_max = 3;
        tau_max = 0.5;
        foot_position = [];
        ctrl_stance = [];
        ctrl_swing = [];
        n_inputs = [];
    end

    methods
        function this = leg_rAct_pAct(attach_dist, attach_angle, ...
                                ctrl_stance_in, ctrl_swing_in, ...
                                ground_in, t0_in)
            if nargin > 5
                super_args = {attach_dist, attach_angle, ground_in, t0_in};
            elseif nargin > 4
                super_args = {attach_dist, attach_angle, ground_in};
            elseif nargin > 2
                super_args = {attach_dist, attach_angle};
            else
                super_args = {};
            end

            this = this@leg(super_args{:});

            if nargin >= 4
                this.ctrl_stance = ctrl_stance_in;
                this.n_inputs = this.ctrl_stance.n_inputs;
                this.ctrl_swing = ctrl_swing_in;
            end
        end

        function f_hip = forces(this, t, x_body, u, stance, xy_foot)
            % Change input to match my array conventions.
            if size(t,1) ~= 1 || size(x_body,1) ~= 6
```

```
        t = t';
        u = u';
        x_body = x_body';
    end

    f_hip = zeros(3, length(t));
    if stance
        gamma = this.stanceLegAngle(x_body);
        if nargin < 6
            l_leg = this.stanceLegLength(x_body);
        else
            l_leg = this.stanceLegLength(x_body, xy_foot);
        end
        [f, tau] = this.ctrl_stance.controlLaw(this, t, u, l_leg, gamma);
        f = min(f, this.f_max);
        tau = min(max(tau, -this.tau_max), this.tau_max);
        f_hip(1,:) = -tau.*cos(gamma)./l_leg - f.*sin(gamma);
        f_hip(2,:) = -tau.*sin(gamma)./l_leg + f.*cos(gamma);
        f_hip(3,:) = -tau;
    end
end

function [value, isterminal, direction] = legEvent(this, t, x, u, stance)
    if stance           % Event value is applied force or leg length
        force = this.forces(t, x, u, stance);
        value = min(force(2), 1-this.stanceLegLength(x));
    else                % Event value is foot clearance
        %fprintf('Time: %f\t', t)
        value = this.ground.clearance(this.footPosition(t, x, u));
        %fprintf('Clearance: %f\n', value)
    end
    isterminal = 1;
    direction = -1;
end

function err = update(this, t, x_body, u, stance)
    if nargin > 2
        if stance
            this.ctrl_swing.setLiftOffState( ...
                this.stanceLegLength(x_body), ...
                this.stanceLegAngle(x_body), ...
                x_body(5));
            this.foot_position = [];
        else
            this.foot_position = this.footPosition(t, x_body, u);
```

```
            end
            err = update@leg(this, t, x_body, u, stance);
        else
            err = update@leg(this, t);
            this.ctrl_swing.setApexState(t);
        end
    end

    function xy_hip = hipPosition(this, x_body)
        alpha_plus_theta = x_body(3,:)+this.attach_pt.angle;
        xy_hip = x_body(1:2, :) + this.attach_pt.distance * ...
                    [cos(alpha_plus_theta);
                     sin(alpha_plus_theta)];
    end

    function gamma = stanceLegAngle(this, x_body, xy_foot)
        xy_hip = this.hipPosition(x_body);
        if nargin < 3
            xy_foot = this.foot_position;
        end
        gamma = atan2(xy_foot(1)    xy_hip(1,:), ...
                      xy_hip(2,:)   xy_foot(2));
    end

    function l_leg = stanceLegLength(this, x_body, xy_foot)
        if nargin < 3
            xy_foot = repmat(this.foot_position, 1, size(x_body, 2));
        end
        xy_diff = this.hipPosition(x_body)   xy_foot;
        l_leg = sqrt( xy_diff(1,:).^2 + xy_diff(2,:).^2 );
    end

    function xy_foot = footPosition(this, t, x_body, u)
        [l_leg, gamma] = this.ctrl_swing.controlLaw(this, t, x_body, u);
        l_leg = min(max(l_leg, this.l_leg_min), this.l_leg_max);
        gamma = min(max(gamma, this.gamma_min+x_body(3)), this.gamma_max+x_body(3));
        xy_hip = this.hipPosition(x_body);
        xy_foot = xy_hip + ...
                  [l_leg.*sin(gamma);  l_leg.*cos(gamma)];
    end
  end
end

classdef leg_xy_force_poly < leg
    properties
        coeff_x = [];
```

```
        coeff_asym_x = [];
        coeff_y = [];
        n_inputs = 5;
    end
    methods
        function this = leg_xy_force_poly(attach_dist, attach_angle, ...
                                          coeff_x_in, coeff_asym_x_in, coeff_y_in, ...
                                          ground_in, t0_in)
            if nargin > 6
                super_args = {attach_dist, attach_angle, ground_in, t0_in};
            elseif nargin > 5
                super_args = {attach_dist, attach_angle, ground_in};
            elseif nargin > 2
                super_args = {attach_dist, attach_angle};
            else
                super_args = {};
                coeff_x_in = [];
                coeff_asym_x_in = [];
                coeff_y_in = [];
            end
            this = this@leg(super_args{:});
            this.coeff_x = coeff_x_in;
            this.coeff_asym_x = padarray(coeff_asym_x_in, ...
                                          size(coeff_x_in) - size(coeff_asym_x_in), ...
                                          0, 'pre');
            this.coeff_y = coeff_y_in;
        end

        function f = forces(this, t, x_body, u, stance)
            if stance
                f = [this.xForce(t, u)'; this.yForce(t, u)'; zeros(1, length(t))];
            else
                f = zeros(3, length(t));
            end
        end

        function fy = yForce(this, t, u)
            fy = u(3)*polyval(this.coeff_y, u(2)*(t - this.t0));
        end

        function fx = xForce(this, t, u)
            fx = u(4)*polyval(this.coeff_x + u(5)*this.coeff_asym_x, u(2)*(t - this.t0));
        end

        function [value, isterminal, direction] = legEvent(this, t, x, u, stance)
```

```
            if stance           % Event value is y force
                value = this.yForce(t, u);
            else                % Event value is time until contact
                value = u(1)    t;
            end
            isterminal = 1;
            direction =  1;
        end
    end
end
```

B.2 Controller Classes

B.2.1 Abstract Controller Class

```
classdef ctrl_abstract < handle
    methods (Abstract)
        u = controlLaw(sys)
    end
end
```

B.2.2 OSHP Controller

```
classdef ctrl_y2poly_x3poly < ctrl_abstract
    properties
        obj_list = {};
        c_list = {};
        ceq_list = {};
        obj_handles = {};
        c_handles = {};
        ceq_handles = {};
        hess_obj_handles = {};
        hess_c_handles = {};
        hess_ceq_handles = {};
        obj_weights = [];
        prms_obj_nlcon_hess = struct('t_bounds', [0; 0], ...
                                     'y_bounds', [0; 0], ...
                                     'th_bounds', [0; 0], ...
                                     'thDot_bounds', [0; 0], ...
                                     'y_ant_bounds', [0; 0], ...
                                     'y_pos_bounds', [0; 0], ...
                                     'y_mid_bounds', [0; 0], ...
                                     'th_ant_bounds', [0; 0], ...
                                     'th_pos_bounds', [0; 0], ...
```

```
                                'th_mid_bounds', [0; 0], ...
                                'th_mid_target', 0);
    prms_lb_ub = struct('lb',[0, 0, 0.5, 0.2, 1, 0, 0, 0.5, 0.2, 1],...
                        'ub',[2, 2e1, 2, 2, 1, 2, 2e1, 2, 2, 1], ...
                        't_return', [0; 0], ...
                        't_swing_prev', []);
    prms_goal_state = struct('goal_state', [0; 0; 0; 0], ...
                             'max_delta_xDot', Inf);
    u_prev = [];
    u0 = [0, 1, 1, 1, 0, 0, 1, 1, 1, 0];
    max_delta_xDot = 0.25;
    %ub = Inf(1, 10);
    constant_args = [];
    filename_root = '';
    opts_fmincon = optimset('Algorithm', 'interior-point', ...
                                    'AlwaysHonorConstraints', 'none', ...
                                    'GradConstr', 'on', ...
                                    'GradObj', 'on', ...
                                    'Hessian', 'user-supplied', ...
                                    'TolFun', 1e 5 , ...
                                    'Display', 'off');
    obj_handle = [];
    nlcon_handle = [];
    hess_handle = []
    flavor = 'a2a';
end
methods (Static)
    generateMFiles(directory , list , dictionary);
    dictionary = buildDictionary();
end

methods
    function this = ctrl_y2poly_x3poly(params_in , obj_list_in , ...
                                    c_list_in , ceq_list_in , ...
                                    obj_weights_in)
        if nargin < 1
            return
        else
            if nargin < 2 || isempty(obj_list_in)
                error('ctrl_2leg_y2poly_x3poly:obj_list', ...
                      ['You must specify at least one ' ...
                       'objective function term']);
            end
            tmp = fieldnames(params_in);
            for ii = 1:length(tmp)
```

```
        if isfield(this.prms_obj_nlcon_hess, tmp{ii})
            this.prms_obj_nlcon_hess.(tmp{ii}) = ...
                params_in.(tmp{ii});
        end
        if isfield(this.prms_lb_ub, tmp{ii})
            this.prms_lb_ub.(tmp{ii}) = params_in.(tmp{ii});
        end
        if isfield(this.prms_goal_state, tmp{ii})
            this.prms_goal_state.(tmp{ii}) = params_in.(tmp{ii});
        end
    end
    this.obj_list = obj_list_in;
    if nargin > 2
        this.c_list = c_list_in;
        if nargin > 3
            this.ceq_list = ceq_list_in;
        end
    end
    if nargin > 4
        if size(obj_weights_in) == size(this.obj_list)
            this.obj_weights = obj_weights_in;
        else
            error('ctrl_y2poly_x3poly:obj_weights', ...
                sprintf(['The number of obj. function ' ...
                'terms (%d) does not match the number ' ...
                'of weights (%d).'], ...
                length(this.obj_list), ...
                length(obj_weights_in)) )
        end
    else
        this.obj_weights = ones(size(this.obj_list));
    end
end

% Get path to containing folder
container_path = regexprep(which('ctrl_y2poly_x3poly'), ...
    '@ctrl_y2poly_x3poly.ctrl_y2poly_x3poly\.m$', '');

% Check for objective and constraint functions that need to be
% generated.
dictionary = this.buildDictionary;
to_generate_list = {};

n_obj_list = length(this.obj_list);
for ii = 1:length(this.obj_list)
```

```
    if ~exist([container_path dictionary(this.obj_list{ii})], ...
            'file')
        to_generate_list = [to_generate_list, this.obj_list(ii)];
    end
end

if ~isempty(this.c_list)
    n_c_list = length(this.c_list);
    for ii = 1:length(this.c_list)
        if ~exist([container_path dictionary(this.c_list{ii})], ...
                'file')
            to_generate_list = [to_generate_list, this.c_list(ii)];
        end
    end
end

if ~isempty(this.ceq_list)
    n_ceq_list = length(this.ceq_list);
    for ii = 1:length(this.ceq_list)
        if ~exist([container_path dictionary(this.ceq_list{ii})],...
                'file')
            to_generate_list = ...
                [to_generate_list, this.ceq_list(ii)];
        end
    end
end

% Generate functions if necessary
if ~isempty(to_generate_list)
    this.generateMFiles(container_path,to_generate_list,dictionary)
end

% Populate handle arrays
this.obj_handles = cell(n_obj_list, 1);
for ii = 1:n_obj_list
    this.obj_handles{ii} = str2func(dictionary(this.obj_list{ii}));
    this.hess_obj_handles{ii} = ...
        str2func(['hess_' dictionary(this.obj_list{ii})]);
end

if ~isempty(this.c_list)
    this.c_handles = cell(n_c_list, 1);
    for ii = 1:length(this.c_list)
        this.c_handles{ii} = str2func(dictionary(this.c_list{ii}));
        this.hess_c_handles{ii} = ...
```

```
                str2func(['hess_' dictionary(this.c_list{ii})]);
        end
    end

    if ~isempty(this.ceq_list)
        this.ceq_handles = cell(n_ceq_list, 1);
        for ii = 1:length(this.ceq_list)
            this.ceq_handles{ii} = ...
                str2func(dictionary(this.ceq_list{ii}));
            this.hess_ceq_handles{ii} = ...
                str2func(['hess_' dictionary(this.ceq_list{ii})]);
        end
    end
end

function u = controlLaw(this, sys)
    % Set boundary conditions
    if isempty(this.prms_goal_state.goal_state)
        error('ctrl_2leg_y2poly:controlLaw',...
            'You must set the goal state before running the controller');
    end
    if isempty(this.u_prev)
        this.u_prev = sys.u;
    end
    if isempty(this.constant_args)
        switch length(sys.params.legs)
            case 2
                angles = sys.params.angle_leg_attach_pts;
                distances = sys.params.dist_leg_attach_pts;
            case 4
                angles = sys.params.angle_leg_attach_pts([1, 3]);
                distances = sys.params.dist_leg_attach_pts([1, 3]);
            otherwise
                error('ctrl_y2poly_x3poly:n_legs', ...
                        [num2str(length(sys.params.legs)) ...
                         ' is not a valid number of legs for this '...
                         'controller']);
        end
        this.constant_args = [sys.params.lstar; ...
                              angles; distances; sys.params.mu_s ];
    end

    % Set Hessian function for optimizaton
    if isempty(this.opts_fmincon.HessFcn)
        this.opts_fmincon.HessFcn = @hess_fun;
```

```
end

% Set arguments for optimization functions
bc = [sys.x([2,3,4,6]), this.strideGoalState(sys);];
args = [reshape(bc, 8, 1); this.strideArgs(); this.constant_args];

% Set bounds for optimization variables
[lb, ub] = this.strideLBUB(sys);

% Run optimization
[u, fval, exitflag] = fmincon(@obj_fun, this.u_prev,...
        [], [], [], [], lb, ub, ...
        @nlcon_fun, this.opts_fmincon);

% Handle optimization failure
if exitflag <= 0 && exitflag ~= 3
    error('ctrl_2leg_y2poly:controlLaw', ...
        ['Optimization failed with exitflag ' num2str(exitflag) ...
         '. Terminating simulation.']);
end
this.u_prev = u;
u([1, 6]) = u([1, 6]) + sys.t;

function [obj, grad_obj] = obj_fun(u)
    obj = 0;
    grad_obj = zeros(size(u));
    for ii = 1:length(this.obj_handles)
        [obj_ii, grad_obj_ii] = this.obj_handles{ii}(u, args);
        obj = obj + this.obj_weights(ii)*obj_ii;
        grad_obj = grad_obj + this.obj_weights(ii)*grad_obj_ii;
    end
end

function [c, ceq, grad_c, grad_ceq] = nlcon_fun(u)
    if isempty(this.c_handles)
        c = [];
        grad_c = [];
    else
        c_cell = cell(size(this.c_handles));
        grad_c_cell = c_cell';
        for ii = 1:length(this.c_handles)
            [c_cell{ii}, grad_c_cell{ii}] = ...
                this.c_handles{ii}(u, args);
        end
        c = cell2mat(c_cell);
```

```
        grad_c = cell2mat(grad_c_cell);
    end
    if isempty(this.ceq_handles)
        ceq = [];
        grad_ceq = [];
    else
        ceq_cell = cell(size(this.ceq_handles));
        grad_ceq_cell = ceq_cell';
        for ii = 1:length(this.ceq_handles)
            [ceq_cell{ii}, grad_ceq_cell{ii}] = ...
                this.ceq_handles{ii}(u, args);
        end
        ceq = cell2mat(ceq_cell);
        grad_ceq = cell2mat(grad_ceq_cell);
    end
end

function hessian = hess_fun(u, lambda)
    hessian = zeros(length(u));
    for ii = 1:length(this.hess_obj_handles)
        hessian = hessian + ...
                  this.obj_weights(ii)* ...
                    this.hess_obj_handles{ii}(u, args);
    end
    if ~isempty(this.hess_c_handles)
        kk = 1;
        for ii = 1:length(this.hess_c_handles)
            hess_ii = this.hess_c_handles{ii}(u,args);
            for jj = 1:size(hess_ii, 3)
                hessian = hessian + ...
                          lambda.ineqnonlin(kk)*hess_ii(:,:,jj);
                kk = kk + 1;
            end
        end
    end
    if ~isempty(this.hess_ceq_handles)
        kk = 1;
        for ii = 1:length(this.hess_ceq_handles)
            hess_ii = this.hess_ceq_handles{ii}(u,args);
            for jj = 1:size(hess_ii, 3)
                hessian = hessian + ...
                          lambda.eqnonlin(kk)*hess_ii(:,:,jj);
                kk = kk + 1;
            end
        end
```

```
            end
        end
    end

    function goal_state_out = strideGoalState(this, sys)
        goal_state_out = this.prms_goal_state.goal_state;
        goal_state_out(3) = ...
            min(sys.x(4)+this.prms_goal_state.max_delta_xDot, ...
                goal_state_out(3));
        %fprintf('Target velocity: %f\n', goal_state_out(3));
    end

    function [lb_out, ub_out] = strideLBUB(this, sys)
        if isempty(this.prms_lb_ub.t_swing_prev)
            this.prms_lb_ub.t_swing_prev = [0; 0];
        elseif sys.t ~= 0
            ind1 = 1:5:length(sys.u);
            ind2 = ind1 + 1;
            this.prms_lb_ub.t_swing_prev = sys.t    ...
                                           (sys.u(ind1) + sys.u(ind2).^ 1);
        end
        lb_out = this.prms_lb_ub.lb;
        ub_out = this.prms_lb_ub.ub;

        lb_out([1,6]) = max(lb_out([1, 6]), ...
                            (this.prms_lb_ub.t_return    ...
                             this.prms_lb_ub.t_swing_prev)');
    end

    function args_out = strideArgs(this)
        % Note: this depends on all of the fields in prms_obj_nlcon_hess
        % being column vectors
        args_out = cell2mat(struct2cell(this.prms_obj_nlcon_hess));
    end

    function setGoalState(this, goal_state_in)
        this.goal_state = goal_state_in;
    end
  end
end

function dictionary = buildDictionary()
    dictionary = containers.Map(...
        {   'fric_cones_2_legs', 'fric_cones_4_legs', ...
            'time_dir_2_legs', 'time_dir_4_legs', ...
            'bounds_tf_2_legs', 'bounds_tf_4_legs', ...
```

```
    'bounds_yf_2_legs', 'bounds_yf_4_legs', ...
    'bounds_thf_2_legs', 'bounds_thf_4_legs', ...
    'bounds_thf__torque_approx__2_legs', ...
        'bounds_thf__torque_approx__4_legs', ...
    'bounds_thDotf_2_legs', 'bounds_thDotf_4_legs', ...
    'bounds_thDotf__torque_approx__2_legs', ...
        'bounds_thDotf__torque_approx__4_legs', ...
    'bounds_y_td_2_legs', 'bounds_y_td_4_legs', ...
    'bounds_y_lo_2_legs', 'bounds_y_lo_4_legs', ...
    'bounds_y_mid_2_legs', 'bounds_y_mid_4_legs', ...
    'bounds_th_td_2_legs', 'bounds_th_td_4_legs', ...
    'bounds_th_lo_2_legs', 'bounds_th_lo_4_legs', ...
    'bounds_th_mid_2_legs', 'bounds_th_mid_4_legs', ...
    'positive_y_mid_2_legs', 'positive_y_mid_2_legs', ...
    'positive_xDot_mid_2_legs', 'positive_xDot_mid_2_legs', ...
    'x_stance_dist_2_legs', 'x_stance_dist_4_legs', ...
    'zero_th_mid_2_legs', 'zero_th_mid_2_legs', ...
    'target_th_mid_2_legs', 'target_th_mid_2_legs', ...
    'min_tf_2_legs', 'min_tf_4_legs', ...
    'min_y_error_2_legs', 'min_y_error_4_legs', ...
    'min_th_error_2_legs', 'min_th_error_4_legs', ...
    'min_th_error__torque_approx__2_legs', ...
        'min_th_error__torque_approx_4_legs', ...
    'min_thDot_error_2_legs', 'min_thDot_error_4_legs', ...
    'min_thDot_error__torque_approx__2_legs', ...
        'min_thDot_error__torque_approx_4_legs', ...
    'max_xDot_2_legs', 'max_xDot_4_legs', ...
    'min_xDot_error_2_legs', 'min_xDot_error_4_legs', ...
    'min_stance_time_2_legs', 'min_stance_time_4_legs', ...
    'min_approx_torque_2_legs', 'min_approx_torque_4_legs', ...
    'min_impulse_x_2_legs', 'min_impulse_x_4_legs', ...
    'min_impulse_y_2_legs', 'min_impulse_y_4_legs'}, ...
{   'ineqFricCones_2Legs', 'ineqFricCones_4Legs', ...
    'ineqTimeDir_2Legs', 'ineqTimeDir_4Legs', ...
    'ineqBoundsTf_2Legs', 'ineqBoundsTf_4Legs', ...
    'ineqBoundsYf_2Legs', 'ineqBoundsYf_4Legs', ...
    'ineqBoundsThf_2Legs', 'ineqBoundsThf_4Legs', ...
    'ineqBoundsThf__torqueApprox__2Legs', ...
        'ineqBoundsThf__torqueApprox__4Legs', ...
    'ineqBoundsThDotf_2Legs', 'ineqBoundsThDotf_4Legs', ...
    'ineqBoundsThDotf__torqueApprox__2Legs', ...
        'ineqBoundsThDotf__torqueApprox__4Legs', ...
    'ineqBoundsYTD_2Legs', 'ineqBoundsYTD_4Legs', ...
    'ineqBoundsYLO_2Legs', 'ineqBoundsYLO_4Legs', ...
    'ineqBoundsYMid_2Legs', 'ineqBoundsYLO_4Legs', ...
```

```
            'ineqBoundsThTD_2Legs', 'ineqBoundsThTD_4Legs', ...
            'ineqBoundsThLO_2Legs', 'ineqBoundsThLO_4Legs', ...
            'ineqBoundsThMid_2Legs', 'ineqBoundsThMid_4Legs', ...
            'ineqPositiveYMid_2Legs', 'ineqPositiveYMid_2Legs', ...
            'ineqPositiveXDotMid_2Legs', 'ineqPositiveXDotMid_2Legs', ...
            'ineqXStanceDist_2Legs', 'ineqXStanceDist_4Legs', ...
            'eqZeroThMid_2Legs', 'eqZeroThMid_2Legs', ...
            'eqTargetThMid_2Legs', 'eqTargetThMid_2Legs', ...
            'objMinTf_2legs', 'objMinTf_4legs', ...
            'objMinYError_2legs', 'objMinYError_4legs', ...
            'objMinThError_2legs', 'objMinThError_4legs', ...
            'objMinThError__torqueApprox__2legs', ...
                'objMinThError__torqueApprox__4legs', ...
            'objMinThDotError_2legs', 'objMinThDotError_4legs', ...
            'objMinThDotError__torqueApprox__2legs', ...
                'objMinThDotError__torqueApprox__4legs', ...
            'objMaxXDot_2legs', 'objMaxXDot_4legs', ...
            'objMinXDotError_2legs', 'objMinXDotError_4legs', ...
            'objMinStanceTime_2Legs', 'objMinStanceTime_4Legs', ...
            'objMinApproxTorque_2Legs', 'objMinApproxTorque_4Legs', ...
            'objMinImpulseX_2Legs', 'objMinImpulseX_4Legs', ...
            'objMinImpulseY_2Legs', 'objMinImpulseY_4Legs'});
end

function [obj_handle, nlcon_handle, hess_handle] = generateMFiles(directory,...
                                                                  list, ...
                                                                  dictionary)
    % Check for saved basic symbolic variables
    if exist([directory 'sym_base.mat'], 'file')
        sym_base = load([directory 'sym_base.mat']);
    else
        % Create basic symbolic variables
        syms t lstar mu_s a_ant a_pos d_ant d_pos
        syms u1 u2 u3 u4 u5 u6 u7 u8 u9 u10 ...
             u11 u12 u13 u14 u15 u16 u17 u18 u19 u20 real;
        syms y0 th0 xDot0 yDot0 thDot0
        c = zeros(28,1);
        c([1,4,7,10]) =  6;
        c([2,5,8,11]) = 6;
        c([3,6,9,12]) = 0;
        c([13,17,21,25]) =  12;
        c([14,18,22,26]) = 18;
        c([15,19,23,27]) =  6;
        c([16,20,24,28]) = 0;
        p1y_nom = poly2sym(c(1:3),t);
        p2y_nom = poly2sym(c(4:6),t);
```

```
p4y_nom = poly2sym(c(7:9),t);
p8y_nom = poly2sym(c(10:12),t);

p1x_nom = poly2sym(c(13:16),t);
p1x_asym = p1x_nom + 6*u5*t^2    6*u5*t;

p2x_nom = poly2sym(c(17:20),t);
p2x_asym = p2x_nom + 6*u10*t^2    6*u10*t;

p4x_nom = poly2sym(c(21:24),t);
p4x_asym = p4x_nom + 6*u15*t^2    6*u15*t;

p8x_nom = poly2sym(c(25:28),t);
p8x_asym = p8x_nom + 6*u20*t^2    6*u20*t;

p1y = u3*compose(p1y_nom, u2*(t u1), t, t);
p1x = u4*compose(p1x_asym, u2*(t u1), t, t);

p2y = u8*compose(p2y_nom, u7*(t u6), t, t);
p2x = u9*compose(p2x_asym, u7*(t u6), t, t);

p4y = u13*compose(p4y_nom, u12*(t u11), t, t);
p4x = u14*compose(p4x_asym, u12*(t u11), t, t);

p8y = u18*compose(p8y_nom, u17*(t u16), t, t);
p8x = u19*compose(p8x_asym, u17*(t u16), t, t);

f1x = p1x*(heaviside(t u1)    heaviside(t (u1 + 1/u2)));
f1y = p1y*(heaviside(t u1)    heaviside(t (u1 + 1/u2)));

f2x = p2x*(heaviside(t u6)    heaviside(t (u6 + 1/u7)));
f2y = p2y*(heaviside(t u6)    heaviside(t (u6 + 1/u7)));

f4x = p4x*(heaviside(t u11)    heaviside(t (u11 + 1/u12)));
f4y = p4y*(heaviside(t u11)    heaviside(t (u11 + 1/u12)));

f8x = p8x*(heaviside(t u16)    heaviside(t (u16 + 1/u17)));
f8y = p8y*(heaviside(t u16)    heaviside(t (u16 + 1/u17)));

py = u3*compose(p1y_nom, u2*(t u1), t, t);
px = u4*compose(p1x_asym, u2*(t u1), t, t);


% Calculate components of yDotf
yDot0 = 0;
```

```
    yDot_f1y = int(p1y, t, u1, t);
    yDotf_f1y = int(p1y, t, u1, u1 + 1/u2);
    yDot_f2y = int(p2y, t, u6, t);
    yDotf_f2y = int(p2y, t, u6, u6 + 1/u7);
    yDot_f4y = int(p4y, t, u11, t);
    yDotf_f4y = int(p4y, t, u11, u11 + 1/u12);
    yDot_f8y = int(p8y, t, u16, t);
    yDotf_f8y = int(p8y, t, u16, u16 + 1/u17);
    yDot_g =  t;

    % Calculate xDotf
    xDotf_f1x = int(p1x, t, u1, u1 + 1/u2);
    xDotf_f2x = int(p2x, t, u6, u6 + 1/u7);
    xDotf_f4x = int(p4x, t, u11, u11 + 1/u12);
    xDotf_f8x = int(p8x, t, u16, u16 + 1/u17);
    xDotf = xDot0 + xDotf_f1x + xDotf_f2x + xDotf_f4x + xDotf_f8x;

    % Find tf
    tf = solve(yDot_g + yDotf_f1y + yDotf_f2y + yDotf_f4y + yDotf_f8y, t);

    % Calculate yf
    yf_f1y = int(yDot_f1y, t, u1, u1 + 1/u2) + yDotf_f1y*(tf   (u1 + 1/u2));
    yf_f2y = int(yDot_f2y, t, u6, u6 + 1/u7) + yDotf_f2y*(tf   (u6 + 1/u7));
    yf_f4y = int(yDot_f4y, t, u11, u11 + 1/u12) + yDotf_f4y*(tf   (u11 + 1/u12));
    yf_f8y = int(yDot_f8y, t, u16, u16 + 1/u17) + yDotf_f8y*(tf   (u16 + 1/u17));
    yf_yDot0 = yDot0*tf;
    yf_g =  tf^2/2;
    yf = y0 + yf_f1y + yf_f2y + yf_f4y + yf_f8y + yf_g + yf_yDot0;

    sym_base = struct('p1x', p1x, 'p2x', p2x, 'p4x', p4x, 'p8x', p8x, 'px', px, ...
        'p1y', p1y, 'p2y', p2y, 'p4y', p4y, 'p8y', p8y, 'py', py, ...
        'f1x', f1x, 'f2x', f2x, 'f4x', f4x, 'f8x', f8x, ...
        'f1y', f1y, 'f2y', f2y, 'f4y', f4y, 'f8y', f8y, ...
        'tf', tf, 'yf', yf, 'xDotf', xDotf);
    save([directory 'sym_base.mat'], 'p1x', 'p2x', 'p4x', 'p8x', 'px', ...
        'p1y', 'p2y', 'p4y', 'p8y', 'py', ...
        'f1x', 'f2x', 'f4x', 'f8x', ...
        'f1y', 'f2y', 'f4y', 'f8y', ...
        'tf', 'yf', 'xDotf');
end

% Create symbolic variables for the functions specified by 'list'
n_list = length(list);
% Create args variables
args_list = {'y0', 'th0', 'xDot0', 'thDot0', 'yf_d', 'thf_d', ...
```

```
            'xDotf_d','thDotf_d', 't_lb', 't_ub', 'y_lb', 'y_ub', ...
            'th_lb', 'th_ub', 'thDot_lb', 'thDot_ub', ...
            'y_ant_lb', 'y_ant_ub', 'y_pos_lb', 'y_pos_ub', ...
            'y_mid_lb', 'y_mid_ub', 'th_ant_lb', 'th_ant_ub', ...
            'th_pos_lb', 'th_pos_ub', 'th_mid_lb', 'th_mid_ub', ...
            'th_mid_target', 'Istar', 'a_ant', 'a_pos', 'd_ant', ...
            'd_pos', 'mu_s'};
args = sym('args', [length(args_list), 1]);
for ii = 1:length(args_list)
    eval([ 'syms args' num2str(ii) ';' ]);
    eval([ args_list{ii} ' = args' num2str(ii) ';']);
end

syms t u1 u2 u3 u4 u5 u6 u7 u8 u9 u10 ...
       u11 u12 u13 u14 u15 u16 u17 u18 u19 u20 real;
for ii = 1:n_list
    tmp = regexp(list{ii}, ...
                '(.*?)(?:__|_(?=[24]))(.*?)?(?:__|(?=[24]))(\d)_?(?:legs|$)', ...
                'tokens', 'once');
    if isempty(tmp)
        error('generateMFiles:n_legs', ...
              [list{ii} ' does not specify a valid number of legs.']);
    end
    name = tmp{1};
    desc = tmp{2};
    n_legs = str2num(tmp{3});
    switch name
        case 'fric_cones'
            if n_legs == 2
                n_fun = 8;
            else
                n_fun = 16;
            end
            fun = sym('fun', [n_fun, 1]);

            fun(1) = u4*(1+u5)/u3   mu_s;
            fun(2) = u4*(1 u5)/u3   mu_s;
            fun(3) = u9*(1+u10)/u8   mu_s;
            fun(4) = u9*(1 u10)/u8   mu_s;
            fun(5) =  u4*(1+u5)/u3   mu_s;
            fun(6) =  u4*(1 u5)/u3   mu_s;
            fun(7) =  u9*(1+u10)/u8   mu_s;
            fun(8) =  u9*(1 u10)/u8   mu_s;
            if n_legs == 4
                fun(9) = u14*(1+u15)/u13   mu_s;
```

```
        fun(10) = u14*(1 u15)/u13    mu_s;
        fun(11) = u19*(1+u20)/u18    mu_s;
        fun(12) = u19*(1 u20)/u18    mu_s;
        fun(13) =  u14*(1+u15)/u13    mu_s;
        fun(14) =  u14*(1 u15)/u13    mu_s;
        fun(15) =  u19*(1+u20)/u18    mu_s;
        fun(16) =  u19*(1 u20)/u18    mu_s;
    end

case 'time_dir'
    if n_legs == 2
        n_fun = 2;
    else
        n_fun = 4;
    end

    tf = adjustForLegNumber(sym_base.tf, n_legs);
    fun = sym('fun', [n_fun, 1]);

    fun(1) = u1 + 1/u2    tf;
    fun(2) = u6 + 1/u7    tf;
    if n_legs == 4
        fun(3) = u11 + 1/u12    tf;
        fun(4) = u16 + 1/u17    tf;
    end

case 'bounds_yf'
    n_fun = 2;

    yf = adjustForLegNumber(sym_base.yf, n_legs);

    fun = sym('fun', [n_fun, 1]);
    fun(1) = yf    y_ub;
    fun(2) = y_lb    yf;

case {'bounds_thf', 'bounds_thDotf'}

    n_fun = 2;

    yDot0 = 0;
    tf = adjustForLegNumber(sym_base.tf, n_legs);
    yf = adjustForLegNumber(sym_base.yf, n_legs);
    xDotf = adjustForLegNumber(sym_base.xDotf, n_legs);
    p1x = sym_base.p1x;
    p1y = sym_base.p1y;
```

```
p2x = sym_base.p2x;
p2y = sym_base.p2y;
if n_legs == 4
    p4x = sym_base.p4x;
    p4y = sym_base.p4y;
    p8x = sym_base.p8x;
    p8y = sym_base.p8y;
end

% Here we assume a constant moment arm for each leg
% Note: cosine takes care of the sign of the moment
r_ant = d_ant*cos(a_ant+th0);
r_pos = d_pos*cos(a_pos+th0);

% Calculate thDotf
if n_legs == 2
    thDot_f1 = int(r_ant*p1y/Istar, t, u1, t);
    thDotf_f1 = int(r_ant*p1y/Istar, t, u1, u1 + 1/u2);
    thDot_f2 = int(r_pos*p2y/Istar, t, u6, t);
    thDotf_f2 = int(r_pos*p2y/Istar, t, u6, u6 + 1/u7);

    thDotf = thDot0 + thDotf_f1 + thDotf_f2;

elseif n_legs == 4
    thDot_f1 = int(r_ant*p1y/Istar, t, u1, t);
    thDotf_f1 = int(r_ant*p1y/Istar, t, u1, u1 + 1/u2);
    thDot_f2 = int(r_ant*p2y/Istar, t, u6, t);
    thDotf_f2 = int(r_ant*p2y/Istar, t, u6, u6 + 1/u7);
    thDot_f4 = int(r_pos*p4y/Istar, t, u11, t);
    thDotf_f4 = int(r_pos*p4y/Istar, t, u11, u11 + 1/u12);
    thDot_f8 = int(r_pos*p8y/Istar, t, u16, t);
    thDotf_f8 = int(r_pos*p8y/Istar, t, u16, u16 + 1/u17);

    thDotf = thDot0 + thDotf_f1 + thDotf_f2 + ...
        thDotf_f4 + thDotf_f8;
end

if strcmp(name, 'bounds_thf')
    % Calculate thf
    thf_thDot0 = thDot0*tf;

    thf_f1 = int(thDot_f1, t, u1, u1 + 1/u2) + ...
                thDotf_f1*(tf   (u1 + 1/u2));
    thf_f2 = int(thDot_f2, t, u6, u6 + 1/u7) + ...
                thDotf_f2*(tf   (u6 + 1/u7));
```

```
        if n_legs == 4
            thf_f4 = int(thDot_f4, t, u11, u11 + 1/u12) + ...
                     thDotf_f4*(tf   (u11 + 1/u12));
            thf_f8 = int(thDot_f8, t, u16, u16 + 1/u17) + ...
                     thDotf_f8*(tf   (u16 + 1/u17));

            thf = th0 + thf_thDot0 + thf_f1 + thf_f2 + ...
                    thf_f4 + thf_f8;
        else
            thf = th0 + thf_thDot0 + thf_f1 + thf_f2;
        end

        fun = sym('fun', [n_fun, 1]);
        fun(1) = thf   (thf_d + th_ub);
        fun(2) = (thf_d + th_lb)   thf;
    else    % Bounds on thDotf
        fun = sym('fun', [n_fun, 1]);
        fun(1) = thDotf   (thDotf_d + thDot_ub);
        fun(2) = (thDotf_d + thDot_lb)   thDotf;
    end

case 'x_stance_dist'
    if n_legs == 2
        n_fun = 2;
    else
        error('generateMFiles:x_stance_dist', 'Not implemented');
    end

    xDotf = adjustForLegNumber(sym_base.xDotf, n_legs);
    yf = adjustForLegNumber(sym_base.yf, n_legs);

    fun = sym('fun', [n_fun, 1]);
    y_nom = ((y_mid_lb+y_mid_ub)/2);
    fun(1) = (xDotf/(2*u2))^2 + y_nom^2   1;
    fun(2) = (xDotf/(2*u7))^2 + y_nom^2   1;

case 'bounds_y_mid'
    if n_legs == 2
        n_fun = 2;
    else
        error('generateMFiles:bounds_y_mid', 'Not implemented');
        n_fun = 8;
    end
```

```
    Py = int(sym_base.py, t);
    PPy = int(Py, t);
    t_mid = adjustForLegNumber(sym_base.tf, n_legs)/2;

    y_mid = heaviside_sum({y0   t_mid^2/2,...
                           subs(PPy, {t}, {t_mid}), ...
                           subs(PPy, {u1, u2, u3, t}, ...
                                {u6, u7, u8, t_mid}), ...
                           ( subs(PPy, {t}, {t_mid}) ...
                            + subs(PPy, {u1, t}, {0, 1/u2}) ...
                            + subs(Py, {u1, t}, {0, 1/u2}) ...
                                 *(t_mid u1 1/u2)), ...
                           ( subs(PPy, {u1, u2, u3, t}, ...
                                  {u6, u7, u8, t_mid}) ...
                            + subs(PPy, {u1, u2, u3, t},
                                   {0, u7, u8, 1/u7}) ...
                            + subs(Py, {u1, u2, u3, t}, ...
                                   {0, u7, u8, 1/u7}) ...
                                 *(t_mid u6 1/u7))}, ...
                           {1,t_mid u1, t_mid u6, ...
                            t_mid u1 1/u2, t_mid u6 1/u7});

    fun = heaviside_sum.empty(0, 1);
    fun(1,:) =  y_mid + y_mid_lb;
    fun(2,:) = y_mid   y_mid_ub;

case 'bounds_th_mid'
    if n_legs == 2
        n_fun = 2;
    else
        error('generateMFiles:zero_th_mid', 'Not implemented');
        n_fun = 8;
    end

    % Here we assume a constant moment arm for each leg
    % Note: cosine takes care of the sign of the moment
    r_ant = d_ant*cos(a_ant);
    r_pos = d_pos*cos(a_pos);

    Py = int(sym_base.py, t);
    PPy = int(Py, t);
    t_mid = adjustForLegNumber(sym_base.tf, n_legs)/2;

    th_mid = heaviside_sum({th0 + thDot0*t_mid,...
                           (r_ant/lstar)*subs(PPy, {t}, {t_mid}), ...
```

```
                    (r_pos/lstar)*subs(PPy, {u1, u2, u3, t}, ...
                                      {u6, u7, u8, t_mid}), ...
                    (r_ant/lstar)*( subs(PPy, {t}, {t_mid}) ...
                     + subs(PPy, {u1, t}, {0, 1/u2}) ...
                     + subs(Py, {u1, t}, {0, 1/u2}) ...
                          *(t_mid u1 1/u2)), ...
                    (r_pos/lstar)*( subs(PPy, {u1, u2, u3, t}, ...
                                       {u6, u7, u8, t_mid}) ...
                     + subs(PPy, {u1, u2, u3, t}, {0, u7, u8, 1/u7}) ...
                     + subs(Py, {u1, u2, u3, t}, {0, u7, u8, 1/u7}) ...
                          *(t_mid u6 1/u7))}, ...
                    {1,t_mid u1, t_mid u6, t_mid u1 1/u2, t_mid u6 1/u7});

    fun = heaviside_sum.empty(0, 1);
    fun(1,:) =  th_mid + th_mid_lb;
    fun(2,:) = th_mid    th_mid_ub;


case 'min_xDot_error'
    n_fun = 1;
    xDotf = adjustForLegNumber(sym_base.xDotf, n_legs);
    fun = (xDotf_d    xDotf )^2;

 case 'min_y_error'
     n_fun = 1;

     yf = adjustForLegNumber(sym_base.yf, n_legs);

     fun = (yf    yf_d)^2;

 case {'min_th_error', 'min_thDot_error'}

     n_fun = 1;

     yDot0 = 0;
     tf = adjustForLegNumber(sym_base.tf, n_legs);
     yf = adjustForLegNumber(sym_base.yf, n_legs);
     xDotf = adjustForLegNumber(sym_base.xDotf, n_legs);
     p1x = sym_base.p1x;
     p1y = sym_base.p1y;
     p2x = sym_base.p2x;
     p2y = sym_base.p2y;
     if n_legs == 4
         p4x = sym_base.p4x;
         p4y = sym_base.p4y;
```

```
    p8x = sym_base.p8x;
    p8y = sym_base.p8y;
end

% Here we assume a constant moment arm for each leg
% Note: cosine takes care of the sign of the moment
r_ant = d_ant*cos(a_ant);
r_pos = d_pos*cos(a_pos);


% Calculate thDotf
if n_legs == 2
    thDot_f1 = int(r_ant*p1y/lstar, t, u1, t);
    thDotf_f1 = int(r_ant*p1y/lstar, t, u1, u1 + 1/u2);
    thDot_f2 = int(r_pos*p2y/lstar, t, u6, t);
    thDotf_f2 = int(r_pos*p2y/lstar, t, u6, u6 + 1/u7);

    thDotf = thDot0 + thDotf_f1 + thDotf_f2;

elseif n_legs == 4
    thDot_f1 = int(r_ant*p1y/lstar, t, u1, t);
    thDotf_f1 = int(r_ant*p1y/lstar, t, u1, u1 + 1/u2);
    thDot_f2 = int(r_ant*p2y/lstar, t, u6, t);
    thDotf_f2 = int(r_ant*p2y/lstar, t, u6, u6 + 1/u7);
    thDot_f4 = int(r_pos*p4y/lstar, t, u11, t);
    thDotf_f4 = int(r_pos*p4y/lstar, t, u11, u11 + 1/u12);
    thDot_f8 = int(r_pos*p8y/lstar, t, u16, t);
    thDotf_f8 = int(r_pos*p8y/lstar, t, u16, u16 + 1/u17);

    thDotf = thDot0 + thDotf_f1 + thDotf_f2 + ...
        thDotf_f4 + thDotf_f8;
end

if strcmp(name, 'min_th_error')
    % Calculate thf
    thf_thDot0 = thDot0*tf;

    thf_f1 = int(thDot_f1, t, u1, u1 + 1/u2) + ...
                thDotf_f1*(tf   (u1 + 1/u2));
    thf_f2 = int(thDot_f2, t, u6, u6 + 1/u7) + ...
                thDotf_f2*(tf   (u6 + 1/u7));

    if n_legs == 4
        thf_f4 = int(thDot_f4, t, u11, u11 + 1/u12) + ...
                    thDotf_f4*(tf   (u11 + 1/u12));
```

```
            thf_f8 = int(thDot_f8, t, u16, u16 + 1/u17) + ...
                        thDotf_f8*(tf   (u16 + 1/u17));

            thf = th0 + thf_thDot0 + thf_f1 + thf_f2 + ...
                    thf_f4 + thf_f8;
        else
            thf = th0 + thf_thDot0 + thf_f1 + thf_f2;
        end

        fun(1) = (thf thf_d)^2;
    else    % thDotf
        fun(1) = (thDotf thDotf_d)^2;
    end

case 'min_approx_torque'

    n_fun = 1;

    xDotf = adjustForLegNumber(sym_base.xDotf, n_legs);
    p1x = sym_base.p1x;
    p1y = sym_base.p1y;
    p2x = sym_base.p2x;
    p2y = sym_base.p2y;
    if n_legs == 4
        p4x = sym_base.p4x;
        p4y = sym_base.p4y;
        p8x = sym_base.p8x;
        p8y = sym_base.p8y;
    end

    % Here we assume a constant moment arm for each leg
    % Note: cosine takes care of the sign of the moment
    r_ant = d_ant*cos(a_ant);
    r_pos = d_pos*cos(a_pos);

    % Calculate approximate shoulder torques
    x1_rel =  (xDot0 + xDotf)/2*(t   u1   1/(2*u2));
    x2_rel =  (xDot0 + xDotf)/2*(t   u6   1/(2*u7));
    y_rel =  (y_mid_lb + y_mid_ub)/2;
    tau1 =  p1x*y_rel + p1y*x1_rel;
    tau2 =  p2x*y_rel + p2y*x2_rel;

    fun = int(tau1^2, t, u1, u1 + 1/u2) + ...
          int(tau2^2,t, u6, u6 + 1/u7);
```

```
            otherwise
                error('generateMFiles:badTag', ...
                    [name ' is not a valid tag']);
        end

        n_u = 5*n_legs;
        u = sym('u',[n_u, 1]);
        if isa(fun, 'sym')
            grad = sym('grad', [n_u, n_fun]);
            hessian = reshape(sym('hess', [n_u, n_fun*n_u]), [n_u, n_u, n_fun]);
        else
            clear grad hessian;
        end

        % Calculate gradients and Hessians
        for jj = 1:n_fun
            fun(jj) = eval(fun(jj));
            grad(:, jj) = simplify(jacobian(fun(jj), reshape(u, n_u, 1)).');
            hessian(:, :, jj) = simplify(jacobian(grad(:,jj), ...
                                                  reshape(u, n_u, 1)).');
        end

        matlabFunction(fun, grad, 'file', [directory dictionary(list{ii})], ...
                       'vars', {u, args});
        matlabFunction(hessian, 'file', ...
                       [directory 'hess_' dictionary(list{ii})], 'vars', {u, args});
    end
end

function sym_out = adjustForLegNumber(sym_in, n_legs)
    switch n_legs
        case 2
            syms t lstar mu_s a_ant a_pos d_ant d_pos
            syms y0 th0 xDot0 yDot0 thDot0
            syms u1 u2 u3 u4 u5;
            u6 = 0; u7 = 1; u8 = 0; u9 = 0; u10 = 0;
            u16 = 0; u17 = 1; u18 = 0; u19 = 0; u20 = 0;
            u11 = sym('u6'); u12 = sym('u7'); u13 = sym('u8'); u14 = sym('u9'); u15 = sym('u10');
            sym_out = eval(sym_in);
        case 4
            sym_out = sym_in;
        otherwise
            error('adjustForLegNumber', ...
                [num2str(n_legs) ' is not a valid number of legs']);
    end
end
```

B.2.3 Leg Controllers

```
classdef ctrl_leg_swing < ctrl_abstract
    properties
        t0_prev = Inf;
        t_apex = 0;
        t_apex_prev = 0;
        l_apex = 0.3;
        gamma_apex = 70*pi/180;
        l_lo = 0;
        gamma_lo = 0;
        yDot_lo = 0;
        l_max = 1;
        gamma_max = 70*pi/180;
    end
    methods
        function [l_leg, gamma_leg] = controlLaw(this, sys, t, x_body, u)
            if all(t > this.t_apex)
                this.t0_prev = sys.t0;
                this.t_apex_prev = this.t_apex;
            end
            pre_ind = (t < this.t0_prev);
            post_ind = (t > this.t0_prev);

            % Assign values for pre stance points
            if any(pre_ind)
                xDot = x_body(4,pre_ind);
                del_t = 1./u(2,pre_ind);
                xy_hip = sys.hipPosition(x_body(:,pre_ind));
                y_foot_rel = ( xy_hip(2,:).* ...
                            (1   (u(1,pre_ind)   t(pre_ind))./ ...
                               (u(1,pre_ind) this.t_apex_prev)).^2);
                x_foot_rel =  xDot.*del_t./2;
                l_leg(pre_ind) = ...
                    this.l_apex*(u(1,pre_ind)   t(pre_ind))./ ...
                        (u(1,pre_ind) this.t_apex_prev) + ...
                    sqrt(x_foot_rel.^2 + y_foot_rel.^2).* ...
                        (1   (u(1,pre_ind)   t(pre_ind))./ ...
                            (u(1,pre_ind) this.t_apex_prev)).^2;
                gamma_leg(pre_ind) = atan2(x_foot_rel,  y_foot_rel);
            end

            % Assign values for post stance points
            if any(post_ind)
                yDot = x_body(5,post_ind);
```

```
            l_leg(post_ind) = 0;
            gamma_leg(post_ind) = 0;
        end
    end

    function setApexState(this, t_apex_in)
        this.t_apex = t_apex_in;
    end

    function setLiftOffState(this, l_lo_in, gamma_lo_in, yDot_lo_in)
        this.l_lo = l_lo_in;
        this.gamma_lo = gamma_lo_in;
        this.yDot_lo = yDot_lo_in;
    end
  end
end
```

```
classdef ctrl_leg_fx_fy_to_f_tau < ctrl_abstract
    properties
        coeff_x = [];
        coeff_asym_x = [];
        coeff_y = [];
        n_inputs = 5;
    end
    methods
        function this = ctrl_leg_fx_fy_to_f_tau(coeff_x_in, coeff_asym_x_in,...
                                                coeff_y_in)
            this.coeff_x = coeff_x_in;
            this.coeff_asym_x = padarray(coeff_asym_x_in, ...
                                         size(coeff_x_in)   size(coeff_asym_x_in), ...
                                         0, 'pre');
            this.coeff_y = coeff_y_in;
        end
        function [f, tau] = controlLaw(this, sys, t, u_in, l_leg, gamma)
            fy = u_in(3)*polyval(this.coeff_y, u_in(2)*(t   sys.t0));
            fx = u_in(4)*polyval(this.coeff_x + ...
                                 u_in(5)*this.coeff_asym_x, u_in(2)*(t   sys.t0));
            f =     (fx.*sin(gamma)   fy.*cos(gamma));
            tau =  l_leg.*(fx.*cos(gamma) + fy.*sin(gamma));
        end
    end
end
```

B.3 Model History Classes

B.3.1 Generic Model History

```
classdef modelHistory < handle
    properties
        model = [];
        deltaT = 0.001;
        phaseData = struct('x0',{},'u',{},'tspan',{},'params',{});
        last_phase_animated = 0;
        last_time_index = 0;
        tData = [];
        xData = [];
        uData = [];
        xyTData = [];
        animationData = struct('R',{},'C',{},...
                               'MenuOptions',{},'MenuData',{});
    end
    methods
        function TF = checkModel(this,modelObj)
            if isempty(this.model)
                this.model = modelObj;
                TF = true;
            elseif this.model == modelObj
                TF = true;
            else
                TF = false;
                disp(['This modelHistory is already associated with a ' ...
                      'different model (modelHistory.m)']);
            end
        end
        function addPhaseData(this, x0, u, tspan, params, varargin)
            if mod(length(varargin),2)
                error('modelHistory:addPhaseData', ...
                      'Additional phase data must be given as name, value pairs');
            end
            currentPhaseData = struct('x0', x0, 'u', u,'tspan', tspan, 'params', params);
            for ii = 1:length(varargin)/2
                currentPhaseData.(varargin{2*ii-1}) = varargin{2*ii};
            end
            if length(fieldnames(this.phaseData))~=length(fieldnames(currentPhaseData))
                if isempty(this.phaseData)
                    this.phaseData = currentPhaseData;
                else
                    error('modelHistory:addPhaseData', ...
```

```
                'Cannot change number of fields in phaseData struct!');
        end
    else
        this.phaseData = [this.phaseData; currentPhaseData];
    end
end
function animate(this, draw)
    if nargin < 2
        draw = true;
    end

    if isempty(this.animationData)
        this.generateAnimationData;
    elseif abs(this.animationData.MenuData(1,end) ...
                this.phaseData(end).tspan(2)) > this.deltaT
        this.generateAnimationData([this.animationData.MenuData(1,end) ...
                                    this.tData(end)])
    end

    if draw
        animationGUI(this. animationData.R,...
            this.animationData.C, zeros(1),...
            this.animationData.MenuOptions,...
            this.animationData.MenuData);
    end
end
function generateAnimationData(this,tspan)
    if isempty(this.tData)
        this.generateTimeHistories
    elseif abs(this.tData(end) this.phaseData(end).tspan(2)) > this.deltaT
        this.generateTimeHistories([this.tData(end) ...
                                    this.phaseData(end).tspan(2)]);
    end
    startInd = this.last_time_index + 1;
    if isempty(this.animationData)
        this.animationData(1).R = ...
            this.model.nodeLocations(this.tData(startInd:end),...
                                     this.xData(startInd:end,:),...
                                     this.uData(startInd:end,:),...
                                     this.xyTData(:,:,startInd:end));
    else
        this.animationData(1).R = [this.animationData.R(:,1:end), ...
            this.model.nodeLocations(this.tData(startInd:end), ...
                                     this.xData(startInd:end,:),...
                                     this.uData(startInd:end,:),...
```

```
                                        this.xyTData(:,:,startInd:end))];
    end
    this.animationData.R = this.animationData.R(:,1:length(this.tData));
    this.animationData(1).C = this.model.connectionMatrix;
    this.animationData.MenuOptions = [{'t'},this.model.stateLabels];
    this.animationData.MenuData = [this.tData';this.xData'];
    this.last_time_index = length(this.tData);
end
function generateTimeHistories(this,tspan)
    this.deltaT = 0.001;
    if isempty(this.phaseData)
        error('modelHistory:noPhaseData','Cannot animate without data');
    else
        params = this.model.params;
        if nargin == 2
            this.tData(end) = [];
            this.xData(end,:) = [];
            this.uData(end,:) = [];
            this.NGRFData.abs(:,end) = [];
            this.NGRFData.xComp(:,end) = [];
            this.NGRFData.yComp(:,end) = [];
            if isfield(params, 'xyT')
                this.xyTData(:,:,end) = [];
            end
            startInd = this.last_phase_animated + 1;
        else
            startInd = 1;
        end
        for i = startInd:length(this.phaseData)
            time = this.phaseData(i).tspan(1):...
                   this.deltaT:...
                   this.phaseData(i).tspan(2);
            if numel(time) > 2
                params = this.phaseData(i).params;
                dynamicsFun = str2func([class(this.model) '.dynamics']);
                options = odeset('RelTol',1e 6,'AbsTol',1e 5, ...
                                 'OutputFcn',@odeTimeout);
                [Tout,Xout] = ode45(dynamicsFun,time,...
                    this.phaseData(i).x0,...
                    options,this.phaseData(i).u,params);
                n = length(Tout)   1;
                this.tData = [this.tData;Tout(1:n)];
                this.xData = [this.xData;Xout(1:n,:)];
                this.uData = [this.uData;repmat(this.phaseData(i).u',n,1)];
                this.NGRFData = ...
```

```
                    [this.NGRFData, ...
                     this.model.groundRxns(Tout(1:n), Xout(1:n,:), ...
                                           this.phaseData(i).u, params)];
                if isfield(params, 'xyT')
                    this.xyTData = ...
                        cat(3,this.xyTData,repmat(params.xyT,[1,1,n]));
                end
            end
        end
        this.last_phase_animated = length(this.phaseData);
        if exist('time','var') && numel(time) > 2
            this.tData = [this.tData;Tout(end)];
            this.xData = [this.xData;Xout(end,:)];
            this.uData = [this.uData;this.phaseData(end).u'];
            this.NGRFData = ...
                [this.NGRFData, ...
                 this.model.groundRxns(Tout(end), Xout(end,:), ...
                 this.phaseData(i).u, params)];
            if isfield(params, 'xyT')
                this.xyTData = cat(3,this.xyTData,params.xyT);
            end
        end
        this.NGRFData = struct('abs', cat(2, this.NGRFData.abs),...
                               'xComp', cat(2, this.NGRFData.xComp),...
                               'yComp', cat(2, this.NGRFData.yComp));
    end
  end
end
```

B.3.2 Model History for Planar Rigid Body Runner Models

```
classdef pRBR_modelHistory < modelHistory
    properties
        NGRFData = struct('abs',[],'xComp',[],'yComp',[]);
        torqueData = [];
    end
    methods
        function addPhaseData(this, x0, u, tspan, params, varargin)
            for ii = 1:length(params.legs)
                params.legs(ii) = params.legs(ii).clone;
            end
            addPhaseData@modelHistory(this, x0, u, tspan, params, varargin{:})
        end
        function generateAnimationData(this,tspan)
```

```
            if nargin == 2
                generateAnimationData@modelHistory(this,tspan);
            else
                generateAnimationData@modelHistory(this);
            end
            this.animationData.MenuOptions(1) = {'tau'};
            this.animationData.MenuOptions = ...
                [this.animationData.MenuOptions,...
                {'NGRF1',...
                 'NGRF2',...
                 'NGRF1_x',...
                 'NGRF2_x',...
                 'NGRF1_y',...
                 'NGRF2_y', ...
                 'Torque1', ...
                 'Torque2'}];
            this.animationData.MenuData = ...
                [this.animationData.MenuData;...
                 this.NGRFData.abs; this.NGRFData.xComp; this.NGRFData.yComp;...
                 this.torqueData];
        end
        function generateTimeHistories(this,tspan)
            if nargin == 2
                generateTimeHistories@modelHistory(this,tspan);
            else
                generateTimeHistories@modelHistory(this);
            end
            n_t = length(this.tData);
            this.torqueData = zeros(2, n_t);
            for ii = 1:length(this.phaseData) % Split this up to avoid running out of memory
                [~, ind_start] = min(abs(this.tData    this.phaseData(ii).tspan(1)));
                [~, ind_end] = min(abs(this.tData    this.phaseData(ii).tspan(2)));
                this.torqueData(:,ind_start:ind_end) = ...
                    this.model.hipTorques(this.tData(ind_start:ind_end), ...
                                          this.xData(ind_start:ind_end,:), ...
                                          this.uData(ind_start:ind_end,:), ...
                                          this.phaseData(ii).params );
            end
        end
    end
end
```

B.4 Testing Script

```
function out = acceleration_test(n_strides, leg_type, record)
```

```
    prms.lstar = 0.3;
    prms.mu_s = 0.8;
    prms.a_ant = 0;
    prms.a_pos = pi;
    prms.d_ant = 0.6;
    prms.d_pos = 0.9;
    prms.t_bounds = [0; 2];
    prms.y_bounds = [0.75; 0.81];
    prms.th_bounds = 3.5*pi/180*([1; 1]);
    prms.thDot_bounds = 6*pi/180*[1; 1];
    prms.y_mid_bounds = [0.25; 0.35];
    prms.th_mid_bounds = 3*pi/180*[1; 1];
    prms.max_delta_xDot = Inf;
    prms.t_return = 1.4*[1; 1];
    prms.t_swing_prev = prms.t_return;
    obj_terms = {...
        'min_xDot_error_2_legs' ...
        'min_y_error_2_legs' ...
        'min_th_error_2_legs' ...
        'min_thDot_error_2_legs' ...
        'min_approx_torque_2_legs' ...
    };
obj_weights = [1e0, 1e0, 2e0, 1e1, 3e 2];
    ineq_terms = {...
        'time_dir_2_legs', ...
        'fric_cones_2_legs' ...
        'bounds_yf_2_legs', ...
        'bounds_y_mid_2_legs', ...
        'bounds_thf_2_legs', ...
        'bounds_thDotf_2_legs', ...
        'bounds_th_mid_2_legs', ...
        'x_stance_dist_2_legs', ...
    };
    eq_terms = {...
        %'zero_th_mid_2_legs',...
    };

    u0 = [0, 1, 1, 1, 0, 0, 1, 1, 1, 0]';

    bc = [0.8,      0.8; ...
        0*pi/180,  0*pi/180; ...
        0,         2; ...
        0*pi/180,  0*pi/180];
    %bc = [0.8,      0.8; ...
        %0*pi/180,   2*pi/180; ...
```

```
    %0,         2; ...
    %0*pi/180,   8*pi/180];

switch leg_type
    case 'leg_rAct_pAct'
        grnd = ground_level(0);
        leg1 = leg_rAct_pAct(prms.d_ant, prms.a_ant, ...
                             ctrl_leg_fx_fy_to_f_tau([ 12  18   6  0], ...
                                                     [6   6  0], [ 6  6  0]), ...
                             ctrl_leg_swing(), grnd);
        leg2 = leg_rAct_pAct(prms.d_pos, prms.a_pos, ...
                             ctrl_leg_fx_fy_to_f_tau([ 12  18   6  0], ...
                                                     [6   6  0], [ 6  6  0]), ...
                             ctrl_leg_swing(), grnd);
        leg1.l_leg_min = 0.25;
        leg1.l_leg_max = 1;
        leg1.gamma_min =  70*pi/180;
        leg1.gamma_max = 70*pi/180;
        leg2.l_leg_min = 0.25;
        leg2.l_leg_max = 1;
        leg2.gamma_min =  70*pi/180;
        leg2.gamma_max = 70*pi/180;
    case 'leg_xy_force_poly'
        leg1 = leg_xy_force_poly(prms.d_ant, prms.a_ant, [ 12  18   6  0], ...
                                 [6   6  0], [ 6  6  0]);
        leg2 = leg_xy_force_poly(prms.d_pos, prms.a_pos, [ 12  18   6  0], ...
                                 [6   6  0], [ 6  6  0]);
    otherwise
        error('steady_state_test:leg_type', ...
              sprintf('%s is not a valid leg type', leg_type));
end
prms.goal_state = bc(:,2);
ctrl_2011_07_15 = ctrl_y2poly_x3poly(prms, obj_terms, ineq_terms, ...
                                     eq_terms, obj_weights);
ctrl_2011_07_15.opts_fmincon.TolFun = 1e 2;
ctrl_2011_07_15.opts_fmincon.MaxIter = 2e3;
pRBR_2011_07_15 = ...
    planarRigidBodyRunner([0; bc(1,1); bc(2,1); bc(3,1); 0; bc(4,1)], u0, ...
    prms, ctrl_2011_07_15, leg1, leg2);

t_array = zeros(size(pRBR_2011_07_15.t,1), n_strides + 1);
x_array = zeros(size(pRBR_2011_07_15.x,1), n_strides + 1);
u_array = zeros(size(pRBR_2011_07_15.u,1), n_strides + 1);
t_array(:,1) = pRBR_2011_07_15.t;
x_array(:,1) = pRBR_2011_07_15.x;
```

```
u_array(:,1) = pRBR_2011_07_15.u;
if record
    phist = pRBR_modelHistory;
end
k1 = 1;
k2 = 1e 2;
k3 = 5e0;
k4 = 5e 2;
apply_controller = false;
for ii = 2:n_strides+1;
    if apply_controller
        ctrl_2011_07_15.prms_obj_nlcon_hess.th_bounds = ...
            max(1e 3,min(ctrl_2011_07_15.prms_obj_nlcon_hess.th_bounds(2),...
                        k3*abs(pRBR_2011_07_15.x(3)    bc(2,2))*[1; 1]));
        ctrl_2011_07_15.prms_obj_nlcon_hess.thDot_bounds = ...
            max(1e 3,min(ctrl_2011_07_15.prms_obj_nlcon_hess.thDot_bounds,...
                        k3*abs(pRBR_2011_07_15.x(6)    bc(4,2))*[1; 1]));
        ctrl_2011_07_15.prms_goal_state.goal_state([1,2,4]) = ...
            ctrl_2011_07_15.prms_goal_state.goal_state([1,2,4])    ...
            k1*(pRBR_2011_07_15.x([2,3,6])    bc([1,2,4],2));
        ctrl_2011_07_15.opts_fmincon.TolFun = ...
            max(1e 6,min(ctrl_2011_07_15.opts_fmincon.TolFun, ...
                        k2*norm(pRBR_2011_07_15.x([2,3,6])    bc([1,2,4],2),Inf)));
    end
    fprintf(['Stride %d\tTolFun = %e\tth = %f\tthDot = %f\t' ...
             'th_bounds = %e\t thDot_bounds = %e\n'], ...
            ii 1,ctrl_2011_07_15.opts_fmincon.TolFun, ...
            x_array(3,ii 1)*180/pi, x_array(6,ii 1)*180/pi, ...
            max(ctrl_2011_07_15.prms_obj_nlcon_hess.th_bounds), ...
            max(ctrl_2011_07_15.prms_obj_nlcon_hess.thDot_bounds));
    %fprintf('Stride %d\n',ii);
    if record
        pRBR_2011_07_15.simulateStride(phist);
        phist.animate(false);
    else
        pRBR_2011_07_15.simulateStride;
    end
    t_array(:,ii) = pRBR_2011_07_15.t;
    x_array(:,ii) = pRBR_2011_07_15.x;
    u_array(:,ii) = pRBR_2011_07_15.u;
    if ~apply_controller && norm(x_array([2,3,6],ii)    ...
                                 x_array([2,3,6],ii 1),Inf) < k4
        apply_controller = true;
        disp('Controller start...');
    end
```

```
    end
    if record
        out = struct('n_strides', n_strides, 'leg_type', leg_type, 'bc', bc, ...
                     'prms', prms, 't_array', t_array, 'x_array', x_array, ...
                     'u_array', u_array, 'phist', phist);
    else
        out = struct('n_strides', n_strides, 'leg_type', leg_type, 'bc', bc, ...
                     'prms', prms, 't_array', t_array, 'x_array', x_array, ...
                     'u_array', u_array);
    end
end
```

B.5 Helper Functions

```
function stop = odeTimeout(t,X,flag,u,params)
persistent tStart timeout
stop = 0;
if isempty(flag)
    tElapsed = toc(tStart);
    if tElapsed > timeout
        stop = 1;
        disp('Solver timed out!')
    end
elseif strcmp(flag,'init')
    tStart = tic;
    timeout = 10;
end
end

function Y = heaviside(X)
%HEAVISIDE    Step function.
%    HEAVISIDE(X) is 0 for X <= 0, 1 for X > 0.
%    HEAVISIDE(X) is not a function in the strict sense.
%    See also DIRAC.

%   Copyright 1993 2008 The MathWorks, Inc.
%   $Revision: 1.1.6.1 $  $Date: 2009/03/09 20:41:30 $

Y = zeros(size(X));
Y(X > 0) = 1;

classdef heaviside_sum < handle_with_clone
    properties
        coeffs = {};
        args = {};
```

```
    n_terms = 0;
end % properties
methods
    function this = heaviside_sum(coeffs_in, args_in)
        if nargin == 0
            return;
        elseif nargin == 2
            if numel(coeffs_in) ~= numel(args_in)
                error('heaviside_sum:numel', sprintf(['The number of ' ...
                    'Heaviside-step coefficients, %d, does not match ' ...
                    'the number of Heaviside-step arguments, %d.'], ...
                    numel(coeffs_in), numel(args_in)));
            else
                this.n_terms = numel(coeffs_in);
            end

            if ~isa(coeffs_in, 'cell')
                coeffs_in = num2cell(coeffs_in);
            end
            for ii = 1:numel(coeffs_in)
                if ~isa(coeffs_in{ii}, 'sym')
                    coeffs_in{ii} = sym(coeffs_in{ii});
                end
            end
            this.coeffs = reshape(coeffs_in, [], 1);

            if ~isa(args_in, 'cell')
                args_in = num2cell(args_in);
            end
            for ii = 1:numel(args_in)
                if ~isa(args_in{ii}, 'sym')
                    args_in{ii} = sym(args_in{ii});
                end
            end
            this.args = reshape(args_in, [], 1);
        end
    end % heaviside_sum constructor

    function R = diff(this, varargin)
        % DIFF Differentiate, assuming continuity at all step locations
        R = heaviside_sum(diff(this.coeffs, varargin{:}), this.args);
    end

    function R = jacobian(this, varargin)
        % JACOBIAN Compute transpose of gradient
```

```
    R = heaviside_sum(cellfun(@jacobian, this.coeffs, ...
                              repmat(varargin, this.n_terms, 1),
                              'UniformOutput', false), this.args);
end

function out = transpose(this)
    out = clone(this);
    for ii = 1:out.n_terms
        out.coeffs{ii} = out.coeffs{ii}';
    end
end
function out = eval(this)
    out = clone(this);
    for ii = 1:out.n_terms
        out.coeffs{ii} = evalin('caller', char(out.coeffs{ii}));
        out.args{ii} = evalin('caller', char(out.args{ii}));
    end
end

function S = sym(this)
    % SYM Convert to a single symbolic expression with Heaviside steps
    S = sym(zeros(padarray(size(this(1).coeffs{1}),
                           [0 ndims(this) ndims(this(1).coeffs{1})], ...
                           1, 'post').*size(this)));
    for ii = 1:numel(this)
        for jj = 1:this(ii).n_terms
            switch ndims(S)
                case 2
                    if size(S, 2) == 1 || numel(this) == 1
                        S(ii) = S(ii) ...
                                + this(ii).coeffs{jj}*heaviside(this(ii).args{jj});
                    else
                        S(:,ii) = S(:,ii) ...
                                  + this(ii).coeffs{jj}*heaviside(this(ii).args{jj});
                    end
                case 3
                    S(:,:,ii) = S(:,:,ii) ...
                                + this(ii).coeffs{jj}*heaviside(this(ii).args{jj});
                otherwise
                    error('heaviside_sum:sym', ...
                          'Arrays with ndims > 3 not supported');
            end
        end
    end
end
```

```
function g = matlabFunction(this,varargin)
    % process inputs
    N = getSyms(varargin);
    funs = {this, varargin{1:N}};
    funs = cellfun(@(f)sym(f),funs,'UniformOutput',false);
    args = varargin(N+1:end);

    g = matlabFunction(funs{:}, args{:});

    % find the index separating the functions from the option/value pairs
    % return the last index of the functions, or 0 if none
    function N = getSyms(args)
        chars = cellfun(@ischar,args) | cellfun(@isstruct,args);
        N = find(chars,1,'first');
        if isempty(N)
            N = length(args);
        else
            N = N 1;
        end
    end
end

function y = plus(this, that)
    if isa(that, 'heaviside_sum')
        coeffs = this.coeffs;
        args = this.args;
        for ii = 1:that.n_terms
            ind = 0;
            for jj = 1:this.n_terms
                if that.args{ii} == this.args{jj}
                    ind = jj;
                    break;
                end
            end
            if ind
                coeffs{ind} = coeffs{ind} + that.coeffs{ii};
            else
                coeffs = [coeffs; that.coeffs(ii)];
                args = [args; that.args(ii)];
            end
        end
        y = heaviside_sum(coeffs, args);
    else
        tmp = heaviside_sum(sym(that), 1);
```

```
            y = this + tmp;
        end
    end

    function y = minus(this, that)
        y = this + (that);
    end

    function y = mtimes(this, a)
        y = clone(this);
        for ii = 1:y.n_terms
            y.coeffs{ii} = a*y.coeffs{ii};
        end
    end

    function y = uminus(this)
        y = this.mtimes(-1);
    end
  end % methods
end % classdef
```

Bibliography

[1] MATLAB.

[2] Dimitri P. Bertsekas. *Nonlinear Programming.* Athena Scientific, Belmont, MA, 2nd edition, 1999.

[3] R. Blickhan. The spring-mass model for running and hopping. *Journal of Biomechanics*, 22(11/12):1217–1227, 1989.

[4] Jonas Buchli and Auke Jan Ijspeert. Self-organized adaptive legged locomotion in a compliant quadruped robot. *Autonomous Robots*, 25(4):331–347, 2008.

[5] Jonas Buchli, Michael Mistry, M. Kalakrishnan, P. Pastor, and S. Schaal. Compliant Quadruped Locomotion Over Rough Terrain. *To appear in Proceedings IROS*, 100, 2009.

[6] M. Buehler, M. H. Raibert, and R. Playter. Robots step outside. *Int. Symp. Adaptive Motion of Animals and Machines (AMAM), Ilmenau, Germany*, pages 1–4, 2005.

[7] Yang Cao, Shengtai Li, and Linda Petzold. Adjoint sensitivity analysis for differential-algebraic equations: The adjoint DAE system and its numerical solution. *SIAM Journal on Scientific Computing*, 24(3):1076, 2003.

[8] Panagiotis Chatzakos and Evangelos Papadopoulos. Self-stabilising quadrupedal running by mechanical design. *Applied Bionics and Biomechanics*, 6(1):73–85, 2009.

[9] Avis H. Cohen and David L. Boothe. Sensorimotor interactions during locomotion: principles derived from biological systems. *Autonomous Robots*, 7(3):239–245, 1999.

[10] Steve Collins, Andy Ruina, Russell L. Tedrake, and Martijn Wisse. Efficient bipedal robots based on passive-dynamic walkers. *Science (New York, N.Y.)*, 307(5712):1082–5, February 2005.

[11] Steven H Collins, Martijn Wisse, and Andy Ruina. A Three-Dimensional Walking Robot with. *The International Journal of Robotics Research*, 20(7):607–615, 2001.

[12] Stelian Coros, Andrej Karpathy, Ben Jones, Lionel Reveret, and Michiel Van De Panne. Locomotion Skills for Simulated Quadrupeds. *ACM Transactions on Graphics*, 30(4):Article TBD, 2011.

[13] Yasuhiro Fukuoka, Hiroshi Kimura, and a. H. Cohen. Adaptive Dynamic Walking of a Quadruped Robot on Irregular Terrain Based on Biological Concepts. *The International Journal of Robotics Research*, 22(3-4):187–202, March 2003.

[14] R. M. Ghigliazza, R. Altendorfer, P. Holmes, and D. Koditschek. A Simply Stabilized Running Model. *SIAM Review*, 2(2):187–218, 2003.

[15] Cavagna A. Giovanni, Norman C Heglund, and C Richard Taylor. Mechanical work basic mechanisms in terrestrial locomotion : two for minimizing energy expenditure. *American Journal of Physiology - Regulatory, Integrative and Comparative Physiology*, 233(5):R243–R261, 1977.

[16] Milton Hildebrand. Further studies on locomotion of the cheetah. *Journal of Mammalogy*, 42(1):84, February 1961.

[17] Milton Hildebrand. Gaits Quadrupedal of Vertebrates. *BioScience*, 39(11):766–775, 1989.

[18] I.a. Hiskens and M.a. Pai. Trajectory sensitivity analysis of hybrid systems. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 47(2):204–220, 2000.

[19] J. Hodgins, J. Koechling, and M. H. Raibert. Running experiments with a planar biped. In G. Giralt and M. Ghallab, editors, *3rd Internatioinal Symposium on Robotics Research*, Cambridge, MA, 1985. MIT Press.

[20] Hiroshi Kimura, Seiichi Akiyama, and Kazuaki Sakurama. Dynamic walking and running of the quadruped using neural oscillator. *Autonomous Robots*, 7(3):247–258, 1999.

[21] Hiroshi Kimura, Yasuhiro Fukuoka, and A. H. Cohen. Adaptive Dynamic Walking of a Quadruped Robot on Natural Ground Based on Biological Concepts. *The International Journal of Robotics Research*, 26(5):475–490, May 2007.

[22] M. Anthony Lewis and George A. Bekey. Gait adaptation in a quadruped robot. *Autonomous Robots*, 12(3):301–312, 2002.

[23] Tad McGeer. Passive Bipedal Running. *Proc. R. Soc. B*, 240(1297):107–134, May 1990.

[24] Tad McGeer. Passive Dynamic Walking. *The International Journal of Robotics Research*, 9(2):62–82, April 1990.

[25] Ioannis Poulakakis. *Stabilizing Monopedal Robot Running: Reduction-by-Feedback and Compliant Hybrid Zero Dynamics.* Ph.d., The University of Michigan, 2009.

[26] Ioannis Poulakakis and Jessy W. Grizzle. The Spring Loaded Inverted Pendulum as the Hybrid Zero Dynamics of an Asymmetric Hopper. *IEEE Transactions on Automatic Control*, 54(8):1779–1793, August 2009.

[27] Ioannis Poulakakis, Evangelos Papadopoulos, and M. Buehler. On the Stability of the Passive Dynamics of Quadrupedal Running with a Bounding Gait. *The International Journal of Robotics Research*, 25(7):669–687, 2006.

[28] M. H. Raibert and H.B. Brown Jr. Experiments in balance with a 2D one-legged hopping machine. *Journal of Dynamic Systems, Measurement, and Control*, 106:75–81, 1984.

[29] M. H. Raibert, H.B. Brown Jr, and M. Chepponis. Experiments in Balance with a 3D One-Legged Hopping Machine. *International Journal of Robotics Research1*, 3(2):75–92, 1984.

[30] M. H. Raibert, M. Chepponis, and H. Brown Jr. Running on four legs as though they were one. *IEEE Journal of Robotics and Automation*, 2(2):70–82, 1986.

[31] C. D. Remy, K. W. Buffinton, and R. Siegwart. Stability Analysis of Passive Dynamic Walking of Quadrupeds. *The International Journal of Robotics Research*, August 2009.

[32] E. N. Rozenvasser. General sensitivity equations of discontinuous systems. *Automation and Remote Control*, 28:400–404, 1967.

[33] Andre Seyfarth, Hartmut Geyer, Michael Günther, and Reinhard Blickhan. A movement criterion for running. *Journal of biomechanics*, 35(5):649–55, May 2002.

[34] Alexander Shkolnik, Michael Levashov, Ian R. Manchester, and Russell L. Tedrake. Bounding on rough terrain with the LittleDog robot. *The International Journal of Robotics Research*, 30(2):192–215, December 2010.

[35] Bruno Siciliano and Oussama Khatib, editors. *Springer Handbook of Robotics*. Springer-Verlag, Heidelberg, 2008.

[36] Koushil Sreenath, HW Park, Ioannis Poulakakis, and Jessy W. Grizzle. A compliant hybrid zero dynamics controller for stable, efficient and fast bipedal walking on MABEL. *International Journal of Robotics Research*, Pre-print:1–24, 2010.

[37] Arjan van der Schaft and Hans Schumacher. *An introduction to hybrid dynamical systems*, volume 251 of *Lecture Notes in Control and Information Sciences*. Springer London, London, 2000.

[38] Rebecca M Walter and David R Carrier. Ground forces applied by galloping dogs. *The Journal of experimental biology*, 210(2):208–16, 2007.

[39] E. R. Westervelt, Jessy W. Grizzle, and D. E. Koditschek. Hybrid zero dynamics of planar biped walkers. *IEEE Transactions on Automatic Control*, 48(1):42–56, January 2003.

[40] Eric R. Westervelt, Christine Chevallereau, Benjamin Morris, Jessy W. Grizzle, and Jun Ho Choi. Experimental Results for Walking. In *Feedback Control of Dynamic Bipedal Robot Locomotion*, volume 1, chapter 8, pages 213–248. CRC Press, Boca Raton, FL, 2007.

[41] P G Weyand, D B Sternlight, M J Bellizzi, and S Wright. Faster top running speeds are achieved with greater ground forces not more rapid leg movements. *Journal of Applied Physiology*, 89(5):1991–1999, November 2000.

[42] John E. Wilson. Walking Toy (US Patent No. 2,140,275), 1936.