Blind Bipedal Walking on Rough Terrain

by

Karsten P. Ulland

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

Bachelor of Science in Electrical Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1995

Author .....
Department of Electrical Engineering and Computer Science
May 30, 1995

Certified by .....
Gill Pratt
Assistant Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
F. R. Morgenthaler
Chairman, Department Committee on Graduate Theses

Blind Bipedal Walking on Rough Terrain

by

Karsten P. Ulland

Submitted to the Department of Electrical Engineering and Computer Science
on May 30, 1995, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Electrical Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This project develops a rough terrain walking algorithim using a simulated robot with prismatic knees and applies it to the control of a simulated robot with revolute knees through virtual modeling. The primary source of robustness lies in the use of force control at the joint level to effect cartesian position and velocity control. No detailed knowledge or sensing of the upcomming terrain is used or required at any time. Data is presented to show the effectiveness of this approach and to demonstrate the realistic limitations to which these models adhere.

Thesis Supervisor: Gill Pratt
Title: Assistant Professor of Electrical Engineering and Computer Science

Acknowledgments

I have to express my gratitude first to Gill, Jerry, Peter, Anne, Marc, Robert, Karl, Dave, Achilles and the Leg Lab in general. They were all a tremendous help. This thesis would not have happened without them.

I also wish to thank Myk, my roommates Matt, Phil, and Brad, Phrank and the Tang and Nestlee Quick hockey team, Parag and the PUTZ volleyball team, and everyone else who kept me company out here this year.

I must thank my parents for all their support and for the freedom they gave me to follow my dreams. I also want to thank all my elementary and high school teachers, who pushed me to realize my potential, especially Mr. Danielson, Mr. Lightfoot, Mrs. Moen, and Mr. Berg. I also owe a debt to all the people back in my home town of LaCrescent, MN, and at the Prince of Peace Church, for taking an interest in my plans and development, and for always meeting me with a warm welcome. Support from so many has helped to carry me through.

I do not know if theses are usually dedicated to anyone, but I want to send this one out to Odin and Justin—my not-so-little dog and not-so-little brother, respectively. Here's to a fun-filled summer together!

Contents

List of Figures

Chapter 1

Introduction

1.1 Why Legs?

Although the wheel was invented thousands of years ago, scientists, inventors, science fiction writers, and dreamers have been contemplating legged locomotion for at least hundreds of years. Legs are attractive for numerous reasons. A vehicle or robot that walks instead of rolls has distinct advantages over uneven terrain. An active suspension is inherently a part of a legged system, allowing large scale control of body height and attitude. Legs can handle larger discontinuities than wheels or even treads. Finally, some motivation comes from a desire to make robots that more closely resemble their human inventors. Robots could then interact with people more like people interact with each other. Although legs are an arguably superior form of locomotion, wheels are found nearly everywhere today while legs are found principally on toys and temperamental robots in a few research facilities. The reason is that wheeled systems are much easier to control, power, and maintain, given today's technology. Self contained legged systems still suffer from severe limitations in speed, balance capability, sensing, and power density. Research in all of these areas promises to continue this slow walk towards reality for this dreamy mode of movement.

1.2 Why Blind?

While arbitrarily detailed information about the terrain can be provided to the robot in simulations, actual implementations limit the precision to which this information can be known. For this reason, minimizing the knowledge of upcoming terrain in the simulation eases the algorithm's implementation on a physical system. In the limit, forward sensors are eliminated altogether, and the robot walks blind. This removes a large amount of computation from the walking control. It also means that vision or ranging could be used for a higher level, low frequency path planning to guide the general direction of the walking, so as to avoid rises and drop-offs which are too steep to handle.

1.3 Why Bipedal?

Implementing such a system on a bipedal robot seemed to focus the issues most concisely. Since this model has point feet, no torque can be exerted on the ground. This means the gait is only statically stable in the double support phase. As roughly 10 percent of the time is spent in double support, dynamic stability becomes an issue. The controller has to run fast enough to stay ahead of the tipping time. The maximum joint torques and/or forces become a limiting factor, although computation time is limited at the same time. The option of stopping to think was not allowed, making for a reactive algorithm which does its best with the conditions in which it finds itself. This requires an approach which performs a reasonable number of calculations and exercises loose control over the robot's inherent behavior. It must remain sufficiently flexible to accommodate what lies ahead.

1.4 The Approach

The algorithm developed in this work uses information from the robot's current state—horizontal velocity, joint positions, and ground contact—to determine control

outputs. It was first implemented on a model with revolute hips and prismatic knees. Then calculation functions to transform the inputs and outputs of the controller for a robot with revolute knees were added. This adapted the algorithm for use on a second model, which is representative of an existing physical robot. Parameters were tuned to smooth vertical oscillation, minimize horizontal acceleration, and handle maximally steep positive and negative slopes.

Chapter 2

Background

2.1 A Brief History

Countless attempts at walking, running, hopping, or crawling machines have been made in the past few centuries—some successful, some not so successful. Concepts have been based on novel mathematical relationships, elaborate mechanical devices, and observations of people and animals. Creations have ranged from mechanical horses and bugs to computer controlled pogo-sticks and man-amplifier suits. While purely mechanical systems typically stumble because they cannot adapt to changes in the terrain, computer controlled systems still get bogged down with even moderately uneven terrain. Sensor limitations and non-ideal actuators further complicate the problems.

Recent work in legged locomotion is progressing in a handful of laboratories around the world. Walking robots range from bipeds to hexapods [1] [3] [4], while running and hopping robots have operated on one, two, and four legs [8]. The study of legged locomotion in animals, humans, and in simulations is more wide-spread [5] [6]. Applications range from medicine and optimizing human athletic performance to computer animation and actually building legged robots [5] [7] [9].

2.2 A Summary of Rough Terrain Work

Most rough terrain work has been based around four and six legged walking platforms [4] or general running systems [2] [9]. The primary issue in this work is foot placement: both choosing where to put each foot on the ground and how to load it after contact. Primary problems include terrain detection, path planning, and actuator power. Vision systems are beginning to develop the processing required to pick out relevant features in video data, and sufficient path planning algorithms have been worked out. However, these issues and the actuator limitations still limit most of this work to simulations.

In a simulated world, these problems can be solved by assumption. Arbitrarily detailed knowledge of the environment and the robot can be provided for free, power density and actuator speed can be increased as required, and arbitrary computing power can be dedicated to the simulation. The constraint of time need not even exist, as frames can be rendered slowly and played back at a faster rate. Simulation has allowed research to continue beyond the barriers still standing in the path of physical system development. Once real systems catch up with simulation assumptions, the knowledge gained from simulated systems should speed further development.

2.3 Biped Walking on Rough Terrain

While some biped walking algorithms have been implemented with varying levels of success and robustness, any rough terrain attempts on two legs have presumably involved adapting smooth terrain algorithms to deal with irregularities in the surface. Many attempts seem to be an afterthought based on an initial smooth terrain algorithm's success. The work in M.I.T.'s Leg Lab has always assumed smooth terrain in actual implementations. Knowledge of when the flight phase will end is quite important for their robots. This requires knowledge of what lies ahead, which can be provided by sensing systems or fulfilled assumptions.

Some papers from this lab deal with rough terrain but assume that the terrain is known in advance. While these approaches will be useful once sensing technology catches up, they provide little beyond interesting thought and simulations for now.

The inverted pendulum walker demonstrates the closest approach to rough terrain preparation from the onset [3]. Since the motion is using force control to keep the body at a constant height above the ground, uneven terrain becomes a perturbation to the leg length. This observation led me to develop an biped controller which tried to keep the body at a constant height. I expected rather robust results from this if two major precautions were taken to avoid detrimental interaction with the ground. The first is to lift the feet high enough to clear what terrain may be under the swing leg. The second is to return the swing leg to the uncertain ground level without slamming it down. Once these problems are solved, quite extreme terrain became easily passable.

Chapter 3

The Models

3.1 Overview

This simulation models a planar bipedal robot walking on rough terrain. Roll, yaw, and lateral movement are prevented by a planar training joint attached to the ground. This connection only allows for translation in the XZ plane and pitching about the Y axis. The robot's knowledge of the terrain is based solely on information available from foot contact. The terrain consists of an XY grid of points with specified elevations, connected by planar patches. The model parameters are set to approximate an actual biped robot being developed in this lab. Actuator force limits and robot mass properties model those of the actual robot.

3.2 The Ground

Interactions with the ground model a collision with a pre-loaded linear spring damper. To speed computation the simulation only checks for contact at the user specified points of the robot. This can lead to entertaining crashes when the robot falls through the floor and hangs from it's feet, but that non-reality is not an issue unless things go dramatically wrong.

When a specified contact point lands on the ground, four spring damper systems

act on it. Spring constants, damping and pre-loads can be set for the X, Y, Z, and θ directions. These values must be tuned for each robot until feet land without bouncing and do not excessively penetrate the ground.

A rough terrain addition to this ground model was provided by Peter Dilworth, a staff member in the M.I.T. Leg Lab. The model takes in a terrain file that allows one to specify how high the ground is at a grid of points in the XY plane. A new ground contact function linearly interpolates in two dimensions between the specified points to produce a three dimensional terrain. Although arbitrary heights can be specified at any point on the XY plane, the work discussed here kept the terrain linear. Heights only changed with movement in the X direction. The robot is limited to move in two dimensions, but it is still a three dimensional structure. Varying the terrain with Y would make the robot walk along a hillside, effectively, since the left and right legs are separated in the Y direction. I have left this as a future expansion of this work.

3.2.1 Ground Variables

While interactions with the terrain are handled internally, it is nice to see what the robot is doing, and it is necessary to know how far off the ground the robot is. Having access to the terrain variables allows the control system to easily calculate the body height, and it makes it possible to graph the terrain. The following variables are shown in Figures 3-1 and 3-2:

q.z The vertical hip position relative to original ground level. This value is provided by the simulator. However, if more realistic issues such as robustness to sensor noise are important, this data should not be used. Instead the height to the ground should be calculated through a leg which is in contact with it.

ter_z The height of the terrain above the original ground. This is updated each control cycle to give the height of the terrain directly under the body. No state directly uses this information, but it is useful for displaying the robot's

view of the terrain it walked over.

z_ter The position of the hips relative to the terrain. This is the difference between q.z and ter_z and serves as the robot's measure of body height. A PD controller modifies the desired vertical force on the body to control this height.

3.3 KWALKA

My first approach, referred to as KWALKA, employs revolute hips on the bottom of a block body with prismatic knees. This allowed speedy initial development of a walking algorithm with intent to modify it for uneven ground. Initial work was easier with the prismatic joints for a number of reasons. Problems were more easily diagnosed and offered more intuitive solutions than a system with a revolute joint at each location. The whole structure was easy to directly analyze and tune. The variables measuring joint positions were easily understood, so images of desired behavior were easily quantified and geometric constraints and actuator limits were simple to choose.

Once the high level control had reached stability on level ground, adaptation to uneven ground proved rather trivial. This is perhaps a result of the force-based control when dealing with the ground. Changes required to deal with rough terrain involved checks to guarantee ground clearance on the swing foot, and handle situations when feet hit the ground unexpectedly. Uphill grades favored shorter stride lengths, while downhill sections were best handled with longer steps. Further work improved the efficiency and reduced peak actuator forces and torques.

The peak actuator output required was of primary concern in developing a realistic simulation. A robot with an enormous power to weight ratio has little trouble dealing with hills and moving its limbs quickly enough to maintain very high velocities, but real mobile robots have real limits which force the development of more efficient approaches. The key to keeping the simulation realistic was keeping the peak actuator output below the limits on the physical robot. For the hip joints,

this is easily checked. With prismatic knees, however, a constant force limit is not sufficient when trying to emulate a robot with revolute knees. The peak longitudinal force a revolute knee can produce is a function of the knee bend. While a peak longitudinal force could be calculated based on imaginary knee bend, the most direct way to emulate the real robot is to reconfigure the simulation.

One may ask why the robot was not built with prismatic knees, if these are simpler to conceptualize. Prismatic joints are usually avoided for reasons of complexity and durability of actual implementation. Additional material is required to prevent binding, and the drive system is continually subject to any forces on the leg. Since the power to weight ratio is a primary limitation for mobile robots, designs which require additional material to perform a given task are not preferred. Reliability and mean time between failures is a second issue. Having more parts which are pushed closer to their limits has a detrimental influence on the run time to repair ratio, making a real robot very time consuming to maintain.

3.3.1 KWALKA Variables

The configuration variables of KWALKA are shown in Figure 3-1.

q.pitch Measures the body rotation from vertical, with nose towards the ground being positive. On a real planar system, this is easily sensed from the planar joint rotation. On a free robot, deviation from vertical would have to be measured with some sort of gyroscope, camera system, or vertical range-finder. Obtaining this data is not trivial, but it is not exceedingly difficult. It is very important information to have.

q.rhip Measures rotation of the hip relative to the body. Rotation under and towards the rear marks the positive direction. This data is readily available from a potentiometer or shaft encoder on a real system.

q.rknee Measures displacement along the leg axis. Upward motion is positive, and full extension is the zero point. This is effectively a measure of shank

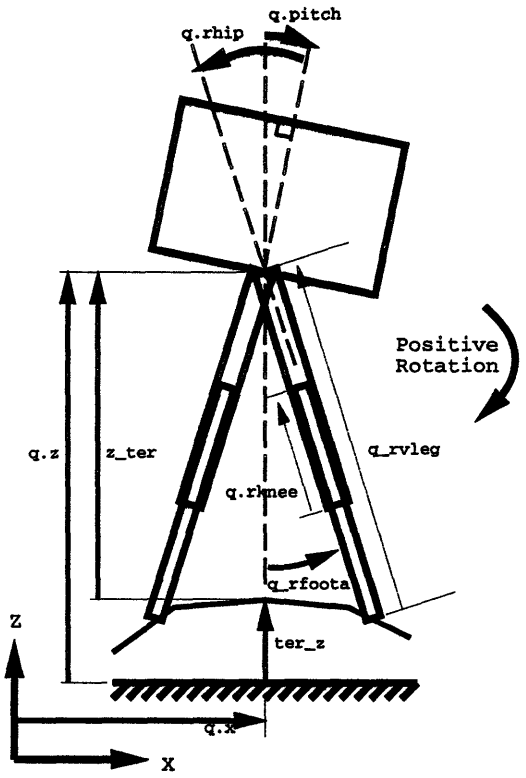

Figure 3-1: Configuration Variables of KWALKA

retraction. A variety of linear position sensors could provide this input on a real robot, such as linear pots or magnetic coils,

q.z The distance from the hip to base ground. This is easily obtained on a two dimensional robot, as the vertical motion of the support can be measured; however, one of two different methods would be required on a free robot: Either calculate the ground height through the knee and hip parameters when a foot is on the ground, or add some sort of altimeter/range-finder to monitor the ground height below the body. Precision requirements may force one to use the more complicated method.

q.x The distance from the hip projection on the X axis to the global origin. This simply keeps track of how far the robot has traveled, but it's derivative is very important for the gait control algorithm. With a boom or treadmill, this is straight forward data to obtain. However, two integrations of inertial data is one of the few ways to measure this in a free robot, and that makes noise a key issue. The algorithm has no need to accurately know the X position, so a real implementation could approximate step lengths if the data were deemed useful. The terrain needs this information, however, and it also allows the simulator view to track the robot as with walks.

qd.x The velocity along the X direction. This provides input for a first order velocity control in the double support phases. It also is important as a condition to end double support. Once the velocity is sufficient to coast over the lead foot, the trailing leg must enter the swing phase in order to be in place soon enough to prevent a fall.

theta_r The angle of the hip relative to vertical. This value increases as the hip swings down and to the rear. It is simply calculated from the global body angle and the angle between the thigh and the body. Position controlling the supporting hip to this value keeps the body near level.

3.4 KWALKB

In order to prove sufficient efficiency, the algorithm was implemented on a second model, known as kwalkb. This model has revolute hips and knees, with the approximate geometry, mass properties, and actuator limitations of the physical robot. It required transformations on both sides of the control system to express the actual joint data in terms that the controller understood and to map the controller's commands onto the joints. This is a required step for implementation on the actual robot, as well, so it was work necessary at some point. While the controller still operates on a virtual prismatic leg, the actuator limits now correspond directly to those of the actual robot. Now physically achievable performance is easy to verify.

Also, when the controller requests too much torque, motor saturation is now simulated. The maximum torque is provided is fixed, placing a constraint on the maximum virtual leg force which varies with knee bend.

3.4.1 KWALKB Variables

In addition to its configuration variables, KWALKB needs to calculate the data that the virtual leg would provide. All the variables are pictured in Figure 3-2. Only the right side variables are listed, since the same description is true for the left side as well. The physical variables are as follows:

q.pitch Measures body rotation relative to vertical, as on KWALKA.

q.rhip Measures rotation of the hip relative to the body. Rotation under and towards the rear marks the positive direction. This data is readily available from a potentiometer or shaft encoder on a real system.

theta_r The angle between the thigh and vertical. This is the same on KWALKA.

q.rknee The outside angle between an extension of the thigh and the shank. This is the angle which fixes the distance between the hip and the foot. As with q.rhip, this is easily measured with a rotary sensor.

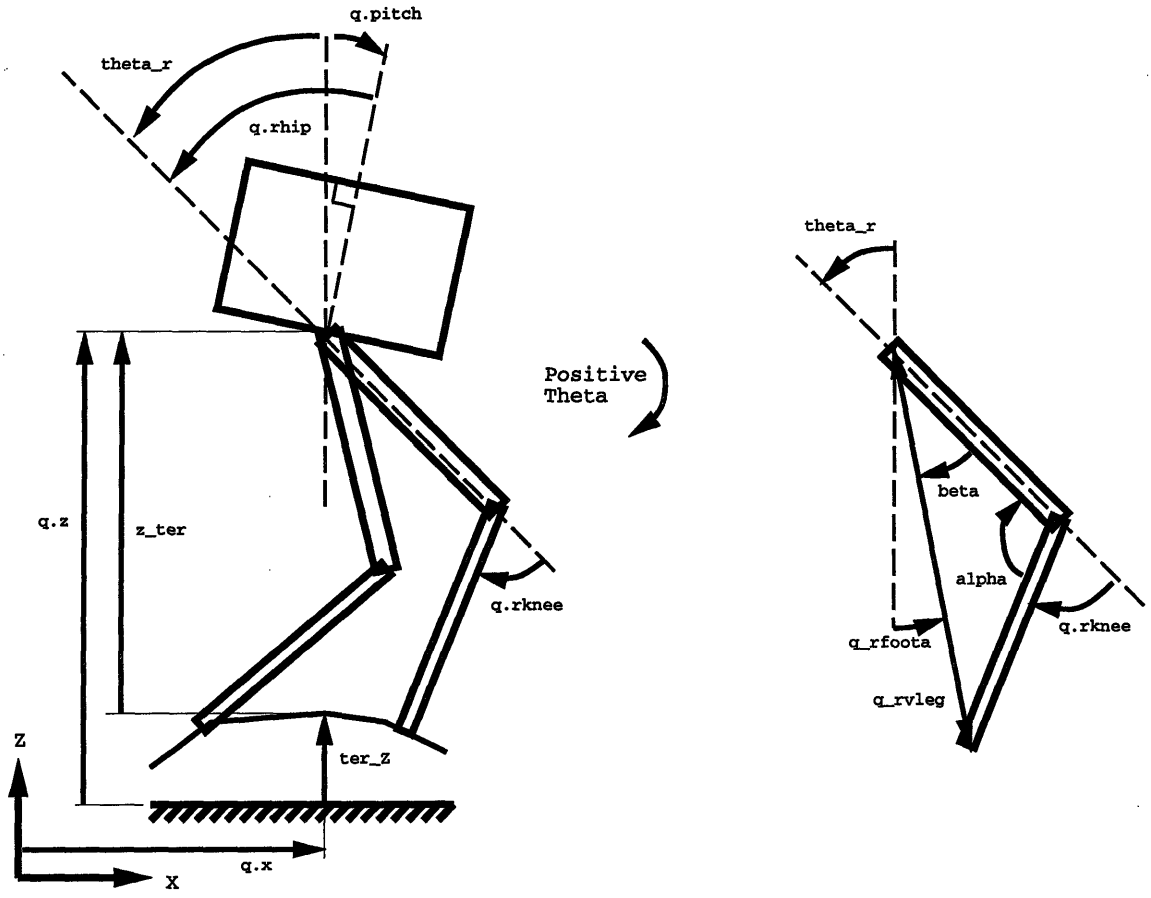

Figure 3-2: Configuration and Virtual Variables of KWALKB

q.z Hip height from ground zero. This is the same as for KWALKA.

From these physical values, the controller can compute the data that a prismatic leg would be returning if its end points coincided with those of the real leg. The virtual data is put in the following variables:

q_rfoota The angle a line connecting the hip to the foot makes from the vertical. From the physical variables, alpha and beta, the upper and obtuse angles in the real leg/virtual leg triangle can be found. These provide an intermediate step to calculate the angle of the virtual leg with respect to vertical using the law of cosines.

q_rvleg The distance between the hip and the foot. This is the length of the virtual leg. Again, the law of cosines can be used to express this length in terms of alpha and the link lengths.

3.5 Servos

In both models, a variety of servos can drive each joint. The default option is a limp servo, which exerts no torque at the joint. The planar joint employs limp servos along the X, Z, and pitch axes. In some states, other robot joint servos are set to be limp to keep foot contacts from generating contending shear forces.

A position control servo uses proportional and derivative gains to control the angle or extension of the joint. This mode is used to put the robot in the initial configuration. Once walking begins, position control is used to move the swing leg and to control the body pitch. The proportional term is closely related to the maximum forces, so the program takes this parameter from the user. In order to achieve the desired damping characteristics, the user specifies a damping ratio, from which a function calculates the required derivative gain.

A third type of servo available is PD position control with a feed forward force or torque term. This mode uses the same gains as the PD mode, but adds the ability

to directly add output torque of force to the PD control. This can shrink the steady state errors when approximate forces can be calculated. The PD control can then correct for errors between the requirements and the estimates, achieving the same total errors with lower PD gains, and therefore, better stability.

A pure feed forward servo is a fourth option. This ignores the position and derivative terms entirely and exerts the commanded force or torque, regardless of where the output is or how it responds.

A final mode is the feed forward servo with damping. The commanded force or torque is applied, but a viscous friction term generates a resistive force proportional to the velocity. This generates a constant velocity servo, which is used to lower the feet to the ground.

Servo outputs are constrained to reasonable limits on KWALKA and to the limits on the actual robot on KWALKB. For KWALKB, this ensures both that controls within the simulator will not exceed the maxima of the real robot and that the simulation's behavior will also predict performance degradation as situational demands exceed the actuators' abilities. Left unchecked, the actuator outputs could be recorded and screened to verify that maxima were not exceeded, but this would not demonstrate what may happen if the real robot is subject to situations which demand greater torques than the motors can provide.

Chapter 4

The High Level Control

4.1 The Basic Control System

The control system is a finite state machine which cycles through a series of states corresponding to each phase of the walking gait. While entering each state, the joint servos are put into a desired mode. One or more conditions specify when to transition to another state. While these conditions are not met, the joints are controlled as directed by the current state. The controller for both models operates on the distance from the hip to the foot and the angle between vertical and the hip to foot line. This in effect assumes that both models have a revolute hip with a prismatic knee.

For kwalka, this is the arrangement, but for kwalkb, this is a virtual structure. The commands for this virtual leg and the data about it must be transformed to deal with the leg that is actually there. This high level controller computes desired Cartesian forces which are then transformed to desired joint torques or forces. The desired vertical force is computed by a proportional plus derivative controller. It commands a vertical force to maintain the body height at the nominal Z parameter in all states. In single support mode, the only direction of force possible is in the line of the virtual leg. This fixes the ratio of the vertical to horizontal forces, so only one Cartesian force can be controlled in single support mode. By setting the

desired vertical force, body height is controlled, while forward velocity is left to do as it will.

The desired horizontal force is calculated only when both feet are on the ground. A simple proportional velocity controller works to maintain the horizontal velocity at the set-point. The robot's forward velocity slows significantly as it moves over the foot in single support, due to the constant height constraint. Once the body has crossed the foot, it accelerates until the swing leg lands. This controller simply applies a force proportional to the velocity error whenever possible (ie. in double support phases). The maximum horizontal force varies with configuration, so the force is limited at the joint level. When the controller commands too great a horizontal force, a net upward force results on one of the feet. The controller assumes that the ground is sticky and can therefore be pulled upon as well as pushed. This behavior is prevented by checking the net force on a supporting foot due to the joints. If this net force ever tries to pull on the ground, the joint commands are adjusted to maintain a slight downward force on the ground.

4.2 Control Variables

Several variables affect the nature of the walking behavior.

qd_d.x Desired X velocity. This parameter is the set point for a proportional velocity controller. The controller uses the difference between this set point and the actual Cartesian velocity to determine the desired horizontal force. In the double support phases of the gait, the weight is distributed between the feet to provide this force while supporting the robot as well.

q_d.z Desired hip height. This input is the set point for a PD controller that calculates a desired Cartesian force to control the robot's height above the ground.

theta_min Minimum forward angle to terminate the swing phase. This parameter specifies how far forward the swing leg must rotate before the swing phase is complete. This corresponds quite closely to stride length.

swing_lead The offset used when the swing leg mirrors the support leg. This value is added to the negated support leg angle in order to put the swing leg ahead of being symmetric with the support leg. This allows better velocity control, since the lead leg touches down farther ahead of the body. The widened stride length allows a broader range of horizontal forces to be commanded.

g_clearance A vertical offset for the swing leg. The swing foot must clear the last ground contact point by this value before the foot is considered clear to swing.

swing_hi A vertical offset that determines the desired swing leg height. The desired value of the swing foot height is set from this offset, which is greater than the g_clearance offset. This puts the leg in motion towards a higher point. The clearance point provides a trigger to begin the swing, in anticipation of where the foot will soon be.

swing_h A vertical offset for the swing leg relative to the support leg. On uneven terrain, the swing foot may leave the ground well below the support foot. If the desired position obtained from the swing_hi offset is insufficient to clear the height of the support foot, swing_h is used to find a new desired position for the swing foot.

4.3 The States

The finite state machine's states include: initialization, stabilization, and repeated states for double support, double to single support transition, single support, and single to double support transition. Figure 4-1 summarizes the sequence of states. The repeated states treat the left and right steps separately, so the state

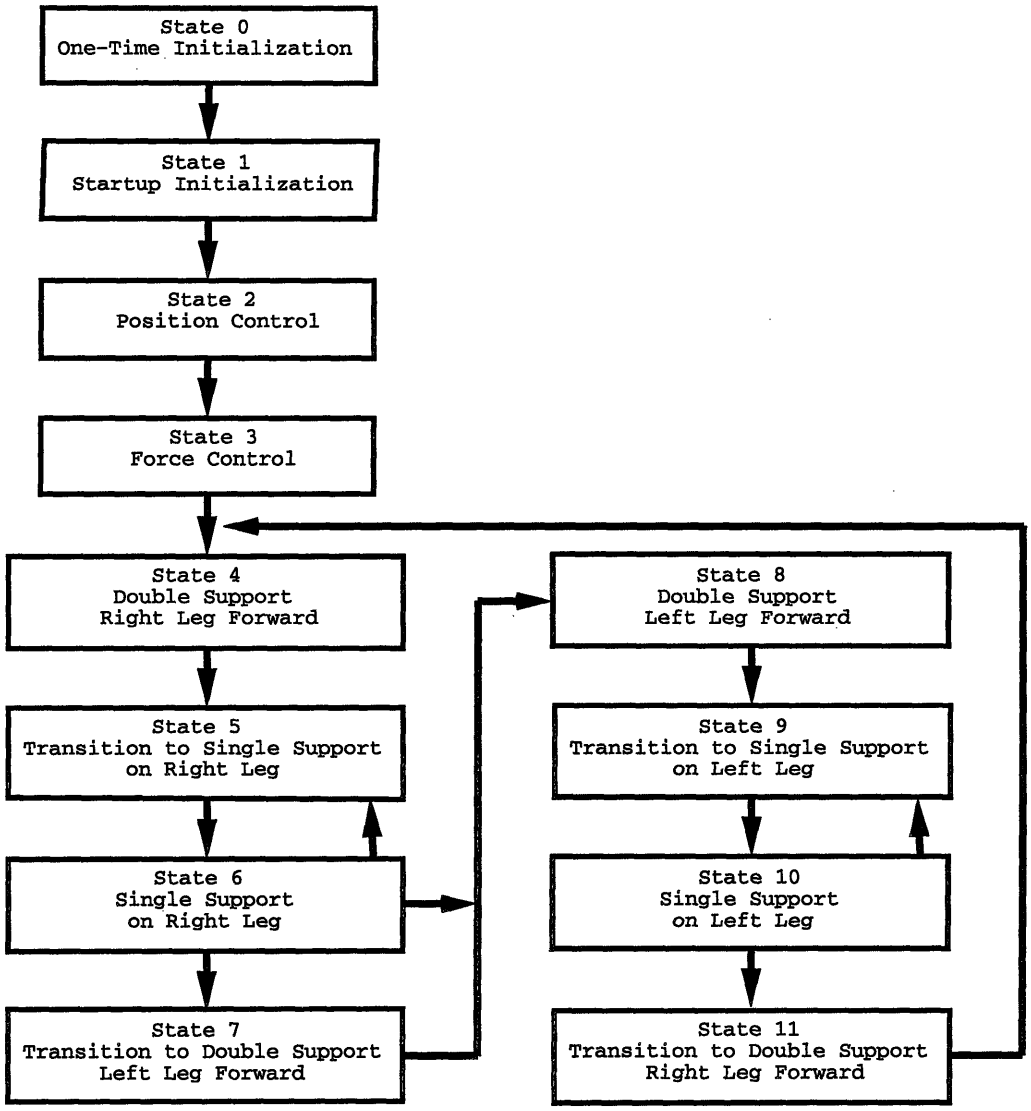

Figure 4-1: The Controller States

completely summarizes the phase and lead leg in the gait. This proved to be more straight forward, although somewhat repetitive.

4.3.1 Before Checking the State

There are a few calculations which are the same for all states. Rather than repeat these inside each state, they appear in adapt_control, which is called at the beginning of the control loop, before the state is determined. Two of these items use the rough terrain function to interpolate the current terrain height under the body, and update both the altitude of the terrain and the body height relative to it. Although the terrain altitude is not used in the control, it is useful to be able to graph the robot's perception of the ground height over time. I also used the difference between the ground height and the robot height from ground zero to fill the relative body height field. This information can also be calculated from the joint positions, as long as the feet are on the ground. I chose the absolute height differences because it simplified the expressions. This absolute measure of body height would be difficult to implement without very precise altimeters, it serves the same purpose as the more direct method of calculating the height from joint angles. If the robot model introduces noise in sensors and actuators, the joint angle calculation must be used to be consistent with the model.

The adapt_control function also computes other information, useful to the controller and to the observer. The angle and distance of each foot relative to the body is updated here, as well as an estimate of the terrain slope and the coefficient of friction each foot is experiencing against the ground. While some of this information is not directly involved with the control, it is useful to have available when assessing how realistic the simulation was.

4.3.2 Initialization and Oscillations

States zero through three perform a series of set up and waiting functions which put the robot at rest before the walking algorithm begins. State zero performs all the one-time initializations, setting constants from the header file parameters and setting up the initial state. Setting globals to the #defined values seems redundant, but this allows initial values to be kept in the header file, away from the tangle of controller code. Making them globals allows them to be viewed and changed at any time in the simulation. This is an advantage when searching for correct parameter values to tune the behavior.

State one performs the detailed initialization and goes on to state two, which updates the knee commands to follow any user changes in the desired z height. After a brief delay, all joints are switched into force control mode, and the knee position gains are reduced, since position control is only used to move the knees when they are in the air from this point forward. The damping required to roughly achieve the desired damping ratio at each joint is calculated for the new gains. Finally, the desired Cartesian forces are calculated and then transformed into joint forces. This state waits for a moment so that any oscillations in the force control can settle out. Then walking can begin.

4.3.3 State 4: Double Support, Right Leg Forward

In the double support phase, Cartesian forces in X and Z are commanded. The lead leg's hip is position controlled to stabilize body pitch, but only while the lead foot is in contact with the ground. This check prevents hip torque from slamming the foot down if it bounces. The body tends to pitch forward in the single support phase as the swing leg is servo-ed forward. Thus, using the lead leg to correct this increases the downward pressure on the lead foot, while the trailing foot would lose contact pressure if that leg were trying to correct the positive pitch error. Therefore, the trailing leg's hip remains limp to keep the trailing foot on the ground while

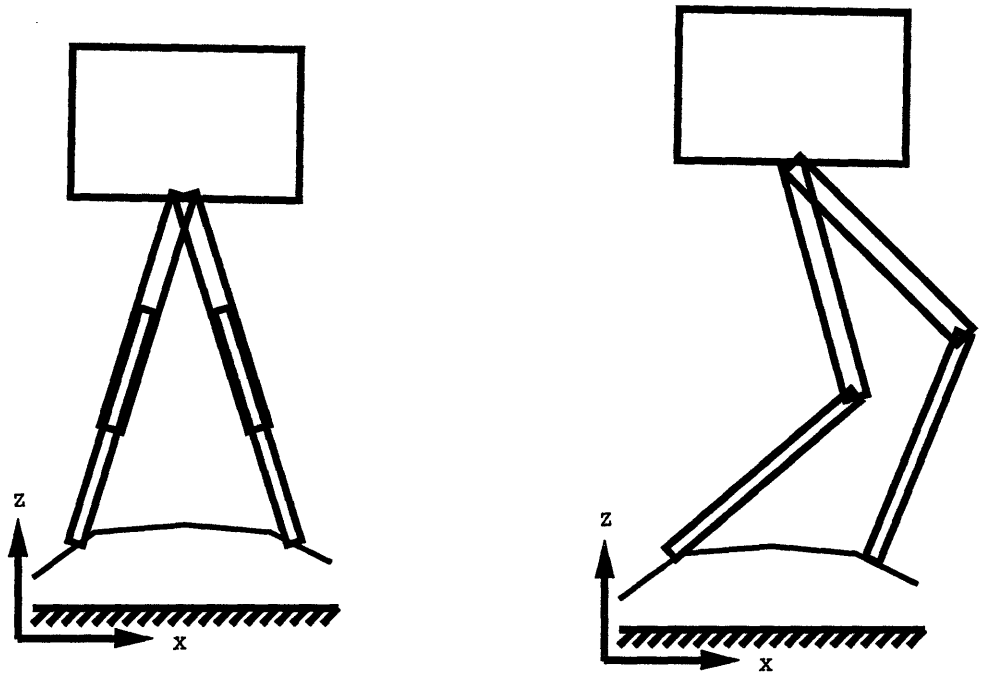

Figure 4-2: Double Support

preventing internal forces from horizontally loading the feet.

Both knees operate in feed forward mode throughout the double support phase. Having set the hip torques as above allows feed forward forces or torques for the knees to be calculated from the Cartesian force equations, completing the transformation from desired Cartesian forces on the body to joint commands.

The trailing leg is vital to contribute horizontal velocity during the double support phase. Once sufficient velocity exists to coast over the support leg with a specified minimum velocity, the trailing leg can be lifted. This condition is developed by integrating the horizontal force acting if the lead leg were to support the weight of the robot and rotate from its current position to vertical. This is the energy that will be subtracted from the current kinetic energy. Since the vertical height is held constant, the kinetic energy at vertical will be the current kinetic minus the work done by the lead leg.

This provides a predictor for the velocity at vertical if single support begins

immediately. Once this final velocity clears the minimum velocity parameter, the trailing leg should enter the swing phase so that it can be in position to begin the next double support phase before excessive velocity develops from having the single support leg behind the body. This calculation does not factor in the deceleration from driving the swing leg forward, so the actual minimum velocity dips below the minimum velocity parameter. It is a sufficiently small error, however, that the minimum parameter can be adjusted slightly higher to compensate.

Time spent in the double support phase can both accelerate or decelerate the body to bring it back to the desired velocity, depending upon which leg contributes more vertical force. As this happens, the minimum velocity condition becomes less demanding because the lead leg continues to rotate towards vertical. Eventually the minimum velocity condition should be satisfied, and control will switch to state 5. As a precaution, state five will take over if the support leg crosses vertical even if the condition is not satisfied. This case is only useful for real time control. It is likely to help the robot regain a stable gait if it has momentarily exceeded the algorithm's maximum velocity. It is quicker to skip to the next state after a direct comparison than to evaluate the minimum velocity expression first.

4.3.4 State 5: Double to Right Support Transition

The principle role of the double to single support transition is to lift the trailing leg off the ground so that it may enter the swing phase and servo forward to prepare for the next double support phase. The lead leg acts as if single support has begun. The horizontal force is ignored in order to maintain the proper vertical force. The body pitch is controlled by servoing the lead leg's hip, as in double support.

The trailing leg is servo-ed to raise the foot to a required ground clearance so that it will not hit the ground in the swing phase. With the prismatic knees, only the knee is position controlled to lift the foot and the hip is left limp. The revolute knee version uses action from both the hip and the knee to lift the foot off the ground along the angle of the virtual leg.

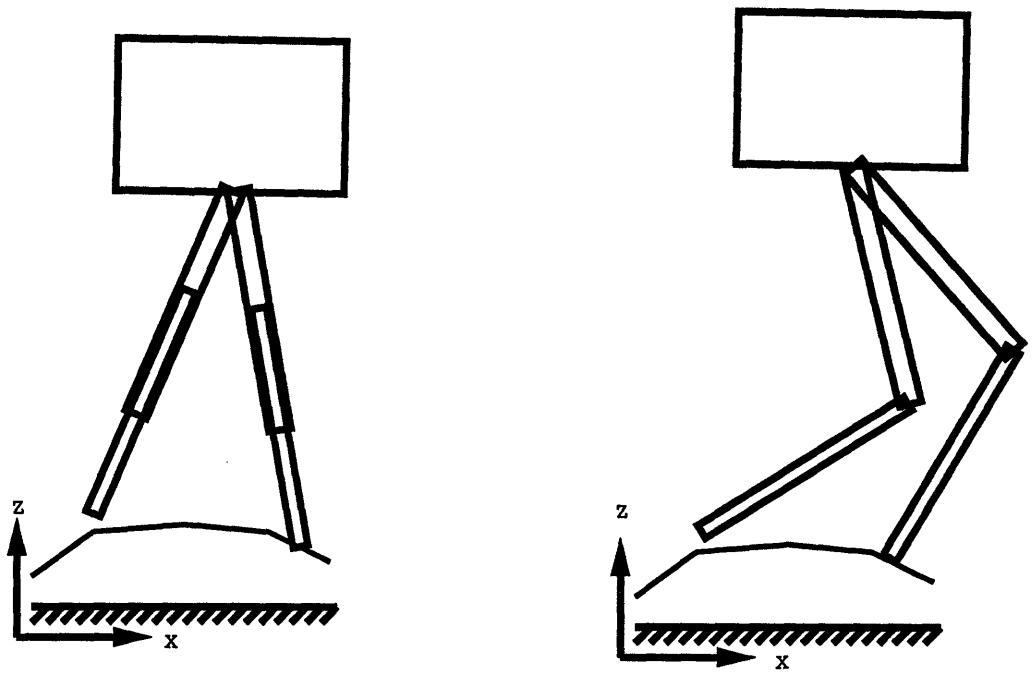

Figure 4-3: Transition to Single Support

The single support state is entered when the rear foot has been lifted higher than the larger of the lead foot height or the minimum ground clearance and the rear foot switch shows no ground contact.

4.3.5 State 6: Single Support on Right Leg

The single support phase simply rides out the transition of lead legs. The support leg's knee is force controlled to maintain body height, while the hip is position controlled to keep the body level. The swing leg is position controlled at both joints to keep the foot above the last ground contact or above the supporting foot by the ground clearance factor. The swing leg also positions the foot to mirror the angle to the support foot, plus a lead factor. This smoothes the transition from trailing leg to lead leg.

Once the swing leg reaches a minimum forward angle, the swing phase is complete, and the transition to double support happens in state 7. If the swing foot hits

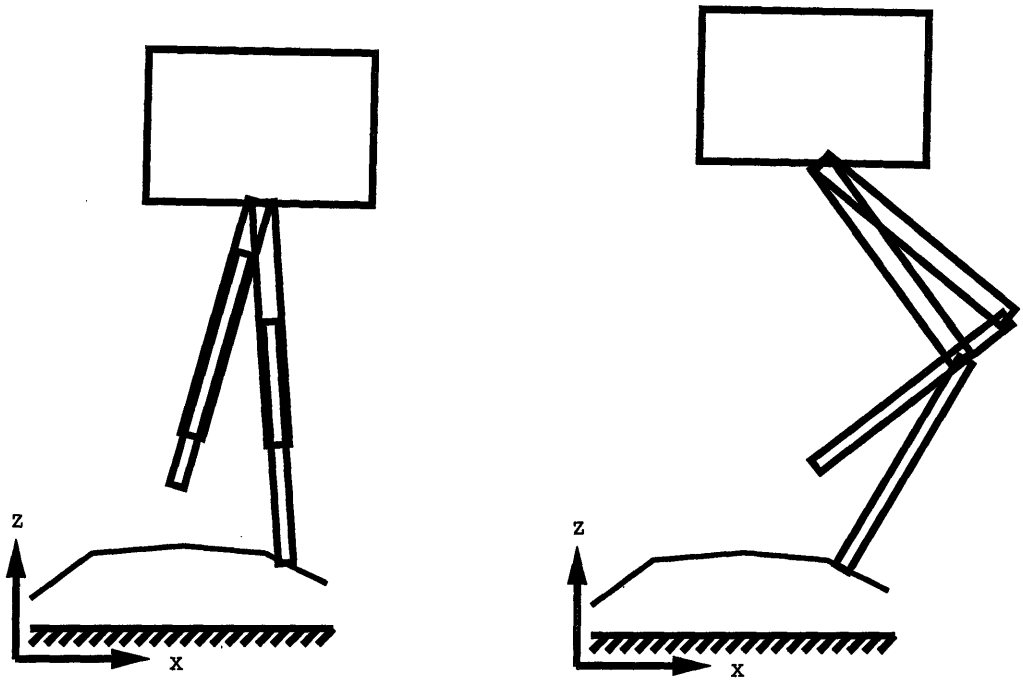

Figure 4-4: Single Support

the ground early, one of two actions is taken. If the swing leg is ahead of the support leg, this simply means that the transition to double support has already occured. The transition state is skipped, and the next double support phase is begun in state 8.

If the swing foot hits the ground while it is behind the support leg, a little stumble is executed by jumping back to state 4. A severe toe-stub can remove sufficient energy to make it impossible to maintain vertical height and rotate over the support foot. By returning to the double support phase, the controller can try ensure there is sufficient energy to coast across the support leg. If the foot hits too far forward of the body, the is no chance to inject additional energy. A backwards fall is then unavoidable without quickly repositioning the feet.

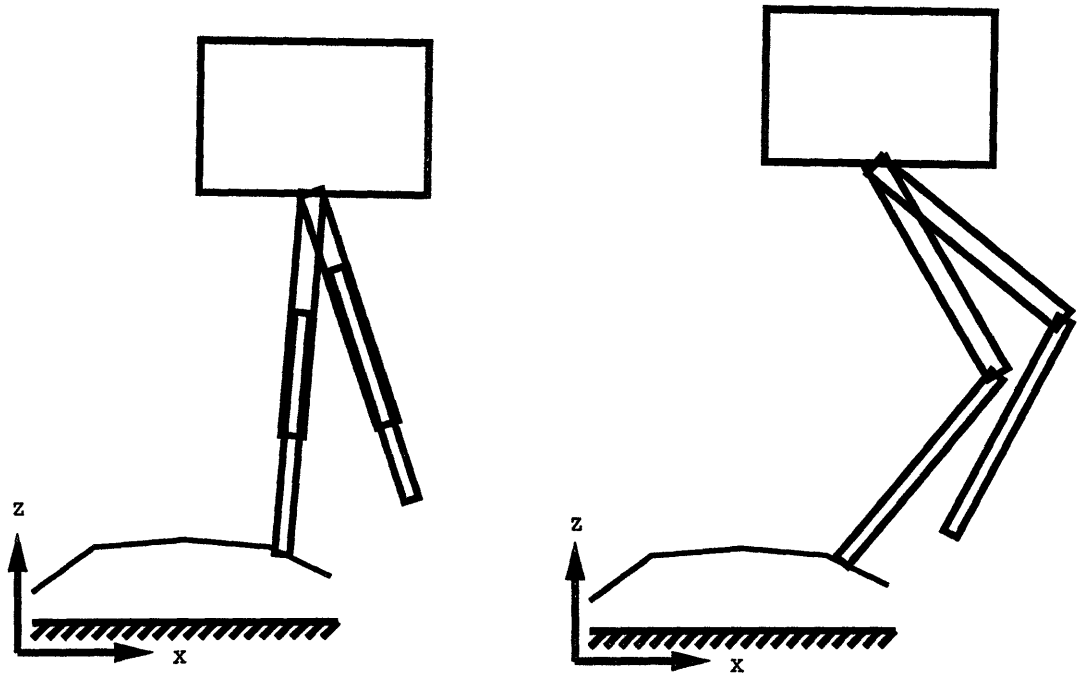

Figure 4-5: Transition to Double Support

4.3.6 State 7 : Right to Double Support Transition

This state is responsible for putting the swing leg back on the ground. This issue is somewhat complicated because the ground elevation beneath the lead foot is not known. To surmount this lacking information, a constant rate of descent is used to lower the foot until it hits the ground. The rate is chosen to be as quick as possible without excessive bouncing, and is set by the combination of a feed forward force and a damping constant. Once the foot registers ground contact, the next double support phase begins.

4.3.7 State 8 : Double Support, Left Leg Forward

This state repeats the calculations of state 4, but with the left and right roles exchanged. The left hip now controls the body angle while the right hip remains limp. More speed is provided by biasing the weight to the right foot, while deceleration is

caused by biasing to the left foot. The energy calculation triggers the transition to state 9.

4.3.8 State 9 : Double to Left Support Transition

While the right hip remains limp, the right foot is lifted from the ground. Once it sufficiently clears the ground or the left foot, single support on the left leg is controlled in state 10.

4.3.9 State 10: Single Support on Left Leg

Single support is executed on the left leg, simply mirroring the conditions and controls from the right leg support. The state can switch to 11 if the swing leg reaches the minimum swing angle, to 4 if the foot hits the ground in front of the support foot, or to 8 if it hits behind.

4.3.10 State 11: Left to Double Support Transition

As before with the right to double support transition, the swing foot is lowered to the ground, and double support in state 4 begins once contact is made.

Chapter 5

Results and Observations

The final performance of both models was quite surprising. Once the algorithms performed on smooth, flat terrain, bumps and significant hills required little more than tuning some parameters. A few new state transition conditions were added to remedy failings which were due to improper state assignment. These new conditions included issues like choosing which state to enter after single support if the swing foot hits the ground before reaching its goal angle. A few bugs in condition calculations even crept out towards the end when an improper variable was used in place of one that measured a similar, but different, value.

For this discussion I will divide the terrain into three main types: terrain with no net slope, terrain with a net up component, and terrain with a net down component. Each type can be smooth or rough to varying degrees. Data is shown from smooth, rough, and very rough versions of level terrain and smooth slopes. The up and down slopes represent the maximum steepness for which I could tune the individual models.

In general, the desired X velocity is kept low on all terrains. This always keeps the maximum forces lower, and it also keeps descents under better control. The double support phase attempts to prevent the velocity from ever falling below the minimum specified, but its prediction of the velocity after lifting the rear leg is based only on the robot's mass and the action of the support leg. There is a rather

significant reaction to the swing leg motion which makes the actual velocity in the swing phase drop lower than anticipated. The .3 m/s minimum I used in all cases proved adequate to prevent a backwards fall, but it was more like specifying the average velocity than the minimum.

5.1 Level Terrain

On level terrain, step length can be kept at a moderate size and the ground clearance variables are kept small. Short steps make for the most efficient motion, since the support leg must do little more than maintain its length, but the robot must then cope with more interactions with the uncertain terrain. If the step is too short, the robot can easily be upset by a drop in terrain, since that results in additional forward velocity that must be controlled by biasing weight to the lead leg in double support. If the lead leg is not sufficiently far ahead, the velocity cannot be reduced to the desired level. After a few steps this velocity can build out of control until the swing leg can no longer reach a position in front of the robot before it hits the ground.

If the lead leg swings too far forward, a down hill slope causes a fall. The leg cannot reach the ground after the swing phase, and the robot has to leave the leg extended and wait to fall onto it. This is usually catastrophic.

Keeping the swing leg close to the ground helps smooth out the transitions into and out of double support. This reduces the delay between the decision to pick up or put down a foot and the completion of the action. The conditions for lifting or putting down a foot neglect the transit time. At significant swing heights, this can become a significant factor in the ability to control the velocity.

An additional set of factors with great significance for the stability of the gait is the swing_lead and theta_min combination. If the swing lead is too large, the steps become very long. To accommodate long steps requires additional power or significant body bounce. When the lead is too small, the swing leg gets behind,

and the velocity goes out of control. Theta_min couples with swing_lead to find a balance between the control of the velocity and the efficiency. While no direct efficiency is calculated, the maximum torques and forces available from the joints prevent grossly inefficient gaits from being stable for long.

5.1.1 Smooth Surface Performance

Performance on smooth terrain was very good, but it did seem to suffer from the precautions present for handling rough terrain. Figure 5-1 presents the data summarizing the behavior of KWALKA on smooth, level terrain. The body exhibits small oscillations just above the set point. These are due to the simple PD height controller. It reacts to deviations caused from imperfect knee forces and commands more or less total vertical force to correct for deviations from the set-point.

The positive steady state error is due to an over estimate in the approximation used in calculating the force due to gravity. The thigh masses are being supported by the knee joints, but usually at an angle off vertical. Therefore, the linear joint structure is bearing some of the load, and the knees would not have to bear the entire weight. The calculations treat the force due to gravity as a constant and set the knee forces to counter it at all times.

The horizontal velocity exhibits oscillatory behavior for an unavoidable reason. The system is under actuated in single support. The vertical position control is given pressidence over the horizontal velocity, so each swing phase was expected to cause the velocity to fall and rise again. With no terrain disturbances and the small step length this run used, the double support states do not last long enough for the robot to achieve the desired horizontal velocity. Then, the velocity dips below the set minimum value because of the neglected swing leg interaction. This actual lower bound is a function of the swing leg mass properties and the forces applied to it. Since this is a constant offset, the minimum velocity set-point must be high enough to accommodate this loss.

The plot of the state shows the initial phases leading into the cycle of gait states.

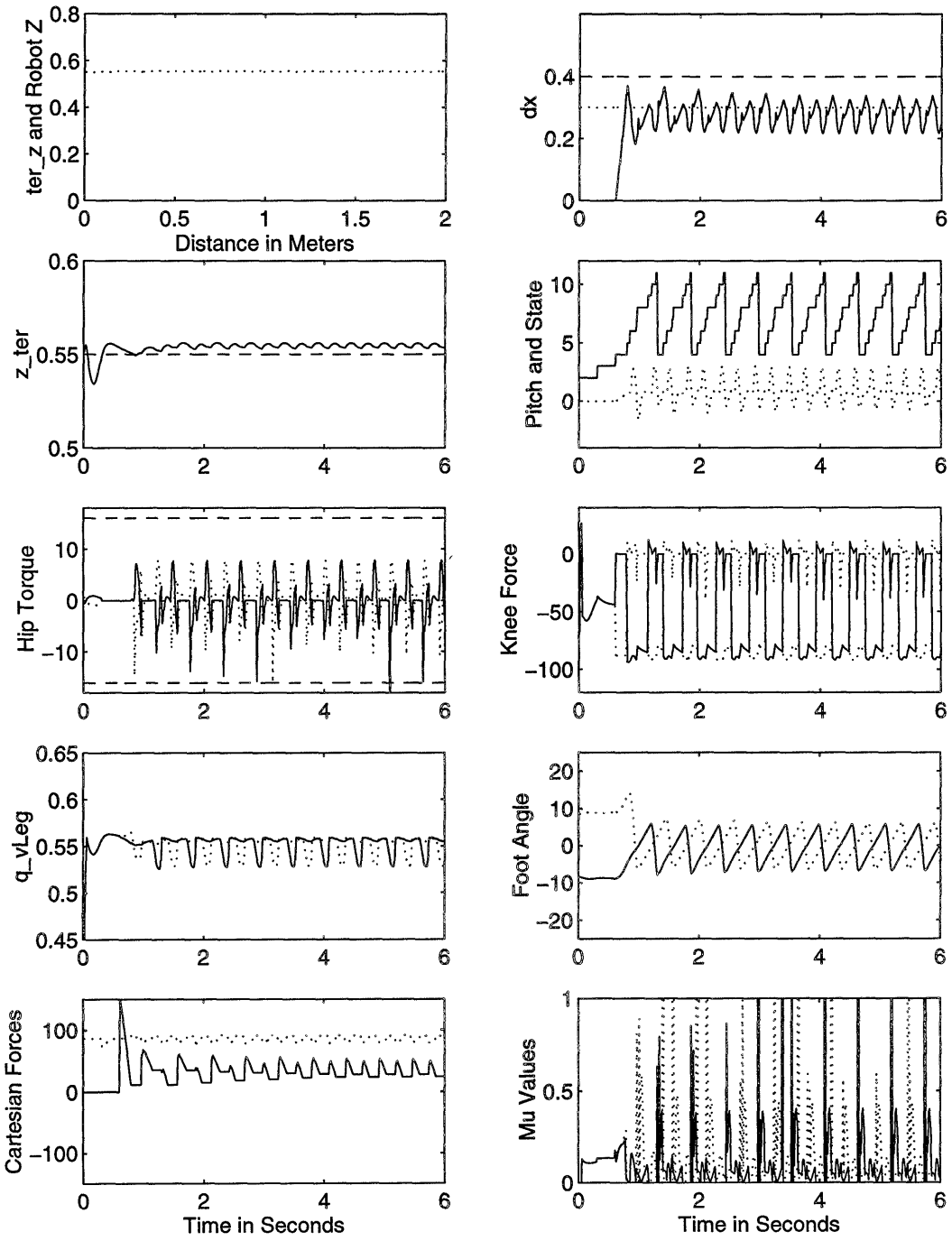

Figure 5-1: KWALKA on smooth, level ground. All units in MKS, with angles in degrees. Dotted traces indicate the left side for leg data graphs.

Each drop of the state from 11 to 4 represents two steps. In this trial, the double support periods account for nearly half of the cycle time. Over half of the remaining time is spent in the single support phases. The short stride length and low ground clearances used here make the transitions between support modes very short.

In order to obtain sufficient joint velocities, the joint damping had to be kept quite low. This results in pitch oscillations in response to the swing leg movements, which remain bounded between plus and minus 3 degrees.

The hip torques remain between the bounds, except for a single spike at the onset of a right foot swing phase. The hip torques are limited by clipping desired position commanded to keep the hip response within the torque limits. This does not account for the derivative term in the controller, which provide an opportunity for the hip torque to momentarily exceed the defined maximum limit. Adding the derivative term should keep the limits more strictly enforced. Since the knee forces are directly commanded, they are kept strictly within the defined limits. The forces are quite biased in the downward direction, which may make designs which take advantage of this more efficient. A symmetric drive costs additional weight for force potential in directions that are not used.

The parameters tracking the virtual leg—in this case, the actual leg—are also shown. The small deviations in leg length and small motions of the feet summarize this gait's small power expense.

The Cartesian forces act as expected. The vertical force hovers about the weight of the robot. Small deviations in vertical height require only minor adjustments in this direction. The horizontal forces peak at the onset of each double support state. This indicates that significant velocity is being lost in the swing phase each time. The magnitude decreases to a steady state level after the first few steps. A larger gain on the velocity controller could reduce the steady state error, but the step length is a primary limitation on the settling time.

The final plot displays the effective coefficients of friction for the left and right feet. It goes to zero when the foot is off the ground. This is monitored by the

foot switch variable for each foot. The forces seem to dissipate out of proportion, however, and a delay in registering foot separation from the ground causes spikes each time a foot is lifted. Without the spikes, this gait requires a friction coefficient of at least .2.

The results for KWALKB on smooth, level ground are shown in Figure 5-2. Better control of the swing leg angle gave this run a longer stride length. This improved the horizontal velocity control but degraded the vertical stabilization. The velocity control was nearly perfect, with the range only slightly exceeding the desired max and min settings. This is likely due to the slightly lower body height than desired. The anticipated loss of energy is not as great as actual, so the disparity between the actual minimum and the desired minimum goes away.

The torque limits on the knees cause a more stringent limitation on the maximum virtual leg force that becomes worse with knee flexion: The more the knee bends, the less force the virtual leg can produce. The stride length limits the body height set point, since the feet must reach the ground at the limits of the step. This forces the knees to operate at a large enough flexion to make the virtual leg much weaker than on the prismatic model.

The Cartesian forces also show greater variation due to the wider stance. While the vertical force changes significantly, the desired horizontal force eventually shows only small spikes to restore energy lost in the step. While the desired horizontal forces are smaller, the applied forces are larger because of the wider stance. This drives the required coefficient of friction up to at least .4 at the start.

5.1.2 Rough, Level Terrain

The next set of tests were run on a randomly generated terrain spanning a list of points 0, 1, 2, 3, and 4 cm high. This required additional swing heights and swing leads to prevent tripping or stubbing the feet on the bumps.

KWALKA showed significantly larger altitude variations on this terrain. The terrain changed faster than the altitude controller could respond since each step put

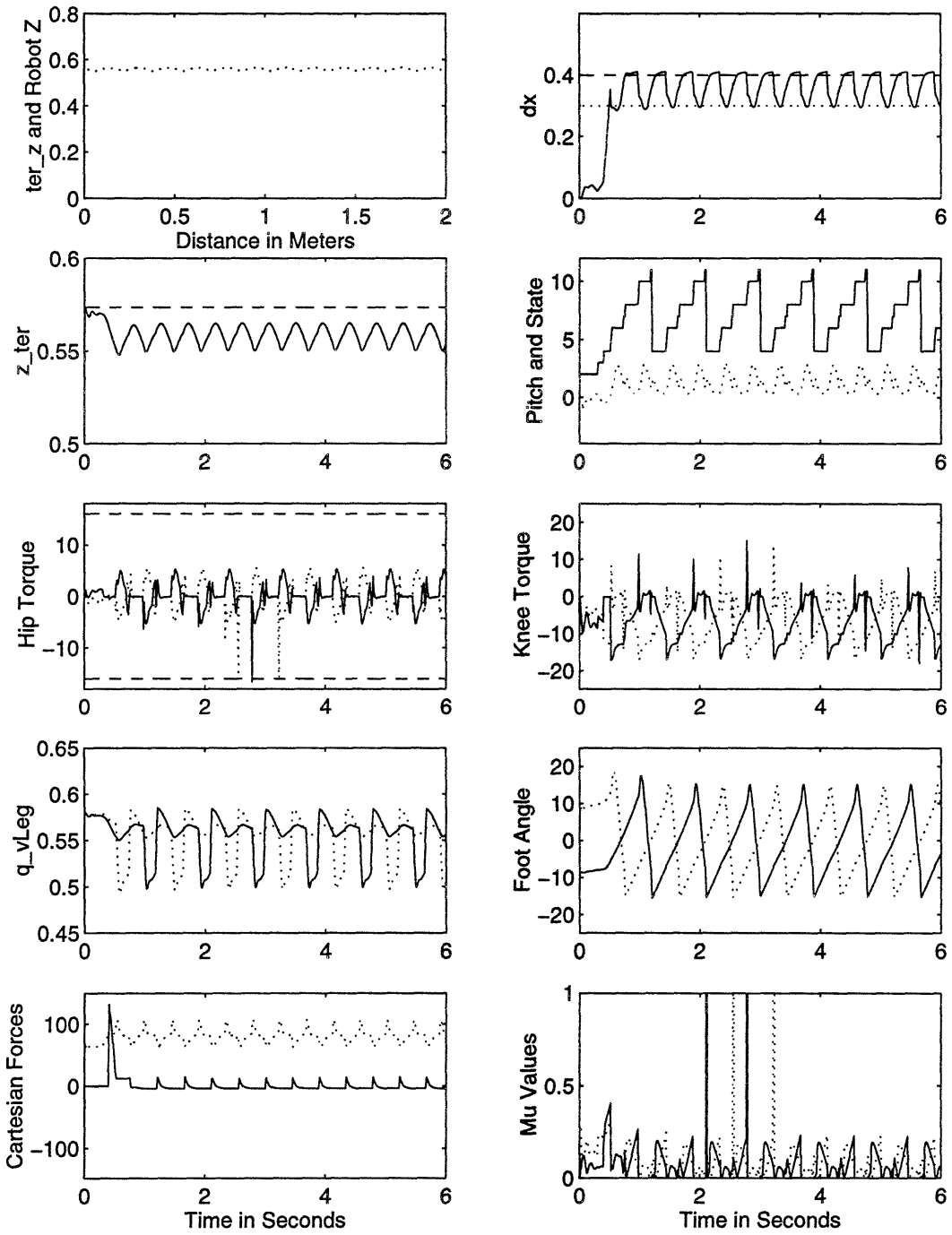

Figure 5-2: KWALKB performance on smooth, level ground. All units in MKS, with angles in degrees. Dotted traces indicate the left side for leg data graphs.

the foot at an unpredictable new height. The longer stride length kept the velocity under control, despite the sudden drops encountered in the terrain. The velocity regularly dips lower than it did on smooth terrain, but each double support phase manages to restore it to the set-point.

Double support takes up a longer segment of the stride cycle because the lead foot lands farther ahead. The pitch oscillations oscillate with the same magnitude, but are now biased slightly negative. Hip and knee torques are more widely varied than they were on smooth ground, but the average magnitudes only increase slightly. What changes significantly is the range of leg lengths. This is due in part to the longer stride length. The uneven ground causes additional modulation, making for marked increases in the motion of the legs. This test demanded much greater energy output from the robot.

Other new features this behavior exhibits are the zeroing of the desired horizontal force and the oscillation evident as the legs swing forward to -10 degrees. The longer strides cause both of these features to appear. The required friction has grown from the smooth ground case as well. There are fewer extraneous spikes, but the valid sections are higher.

Figure 5-4 shows the results of sending KWALKB across the rough terrain. Comparison with the smooth terrain test shows an increase in altitude variation, although it is always below the set-point. The gait cycles become less regular, and some of the double support phases lengthen considerably. The velocity remains very well controlled, with a slight variation beyond the bounds after a significant bump. Hip torques actually drop, and knee torques remain about the same. The last step with the right foot lands hard, as can be seen from the larger knee torque commanded at that point.

The range of virtual leg lengths and angles expands dramatically. This is mostly due to the lead leg taking a long time to come down when stepping into a depression, because the foot continues to move forward as it descends. The friction demands increase rather dramatically as well.

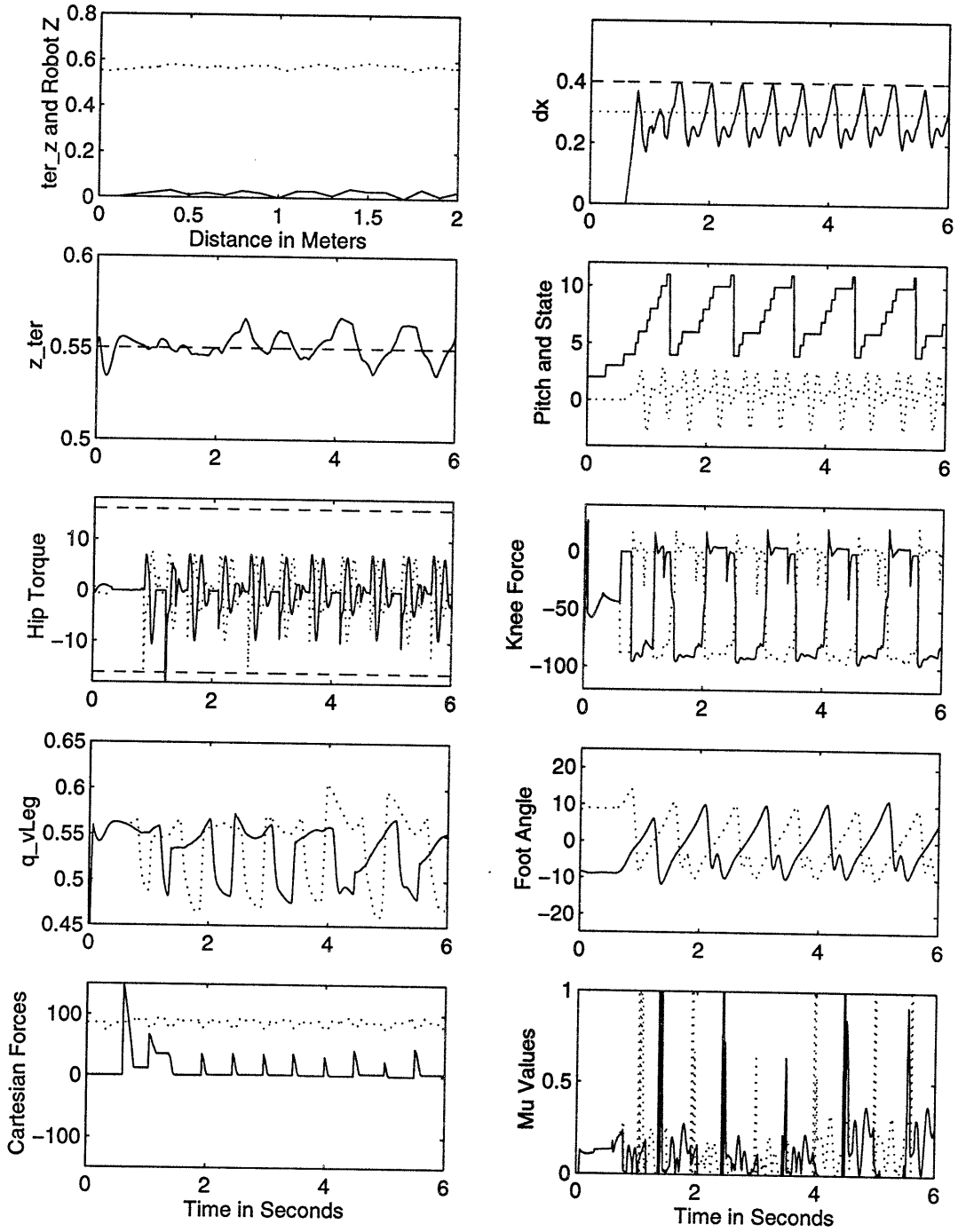

Figure 5-3: KWALKA performance on rough, level ground. All units in MKS, with angles in degrees. Dotted traces indicate the left side for leg data graphs.

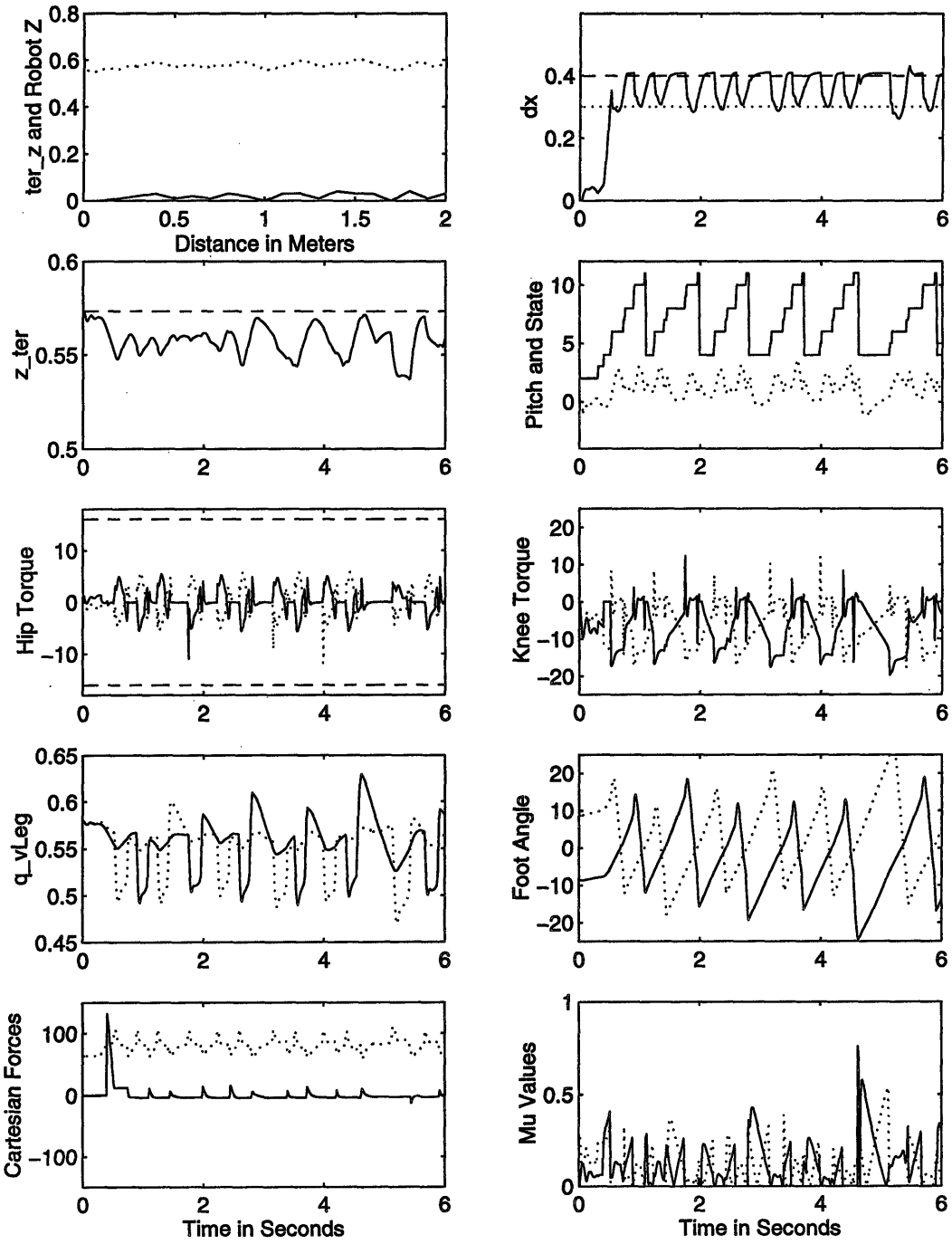

Figure 5-4: KWALKB performance on rough, level ground. All units in MKS, with angles in degrees. Dotted traces indicate the left side for leg data graphs.

Comparison of the two models on the rough terrain shows considerably less velocity and pitch deviation in KWALKB. The hip torques are lower, and the height varies by about the same amplitude. The revolute knee model shows greater variation of leg length and angle, however. This can cost additional energy. The efficiency comparison is somewhat clouded by the greater hip torque in the prismatic model, though. KWALKB's steps are longer and the average desired horizontal force is much lower. It does demand greater friction, however.

The next piece of terrain doubled all the heights of the rough terrain to produce very rough terrain. The grid points defining the terrain are 0, 2, 4, and 8 cm above the floor. KWALKA shows similar changes to those between the smooth and rough tests. The only notable increase is on body height control. The friction requirements grow noticeably as well. The commanded knee forces are slightly larger, and the virtual leg lengths and angles also increase by a small percentage. Figure 5-5 shows summarizes the results.

KWALKB shows a much more dramatic difference between its rough and very rough terrain performances. The vertical position error is much greater on the very rough terrain, and the knee torque increases enough to saturate on the last right foot step. The added vertical deviation aggravates the leverage disadvantage the knees experience. The velocity swings up by more than 40 percent stepping off a bump at the end.

Compared to the performance of KWALKA, the minimum velocity is better regulated, but the maximum velocity is less controlled. The vertical deviations are greater and still centered below the set-point. Hip torques are still significantly smaller, but virtual leg motion is considerably larger. More friction is required.

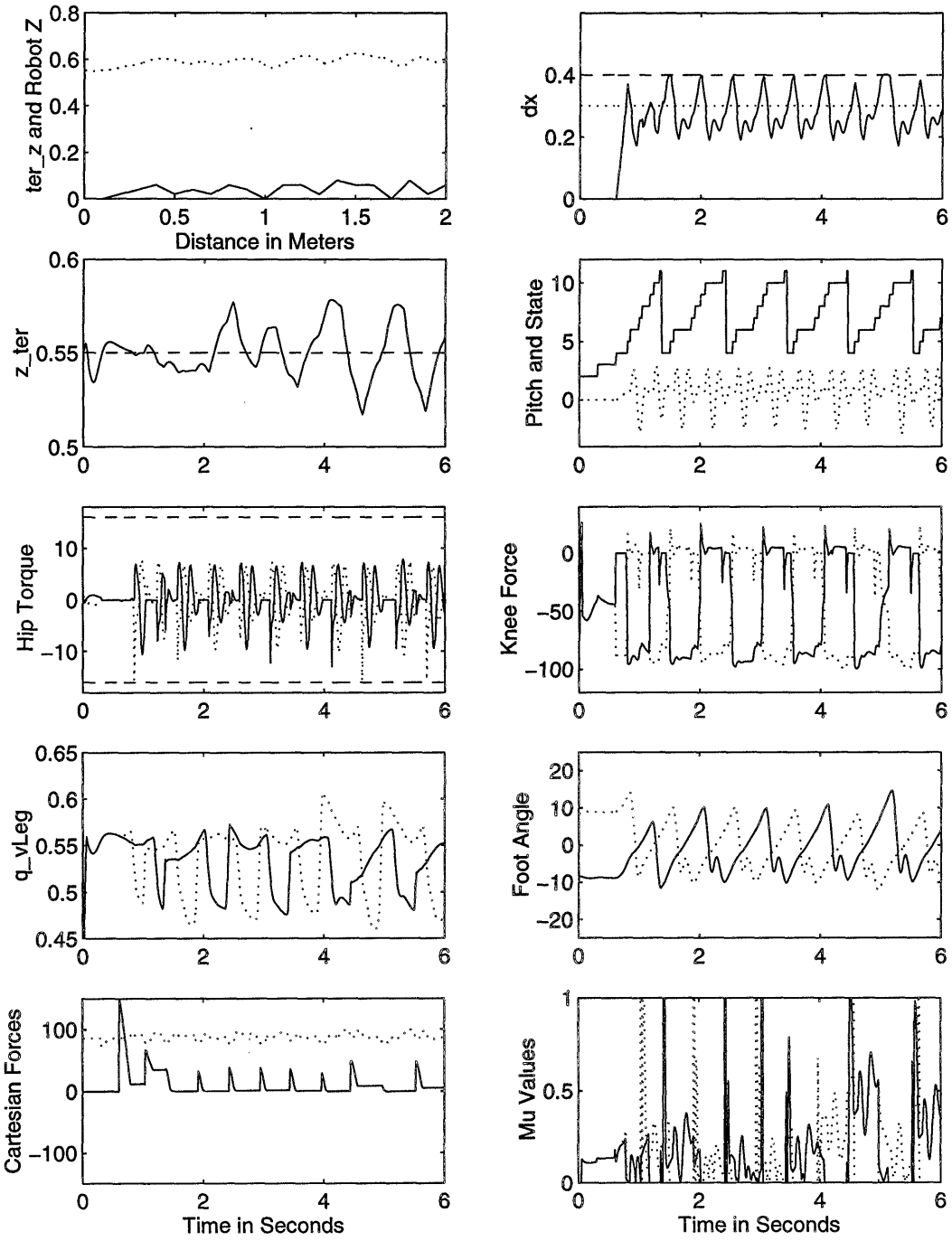

Figure 5-5: KWALKA performance on very rough, level ground. All units in MKS, with angles in degrees. Dotted traces indicate the left side for leg data graphs.

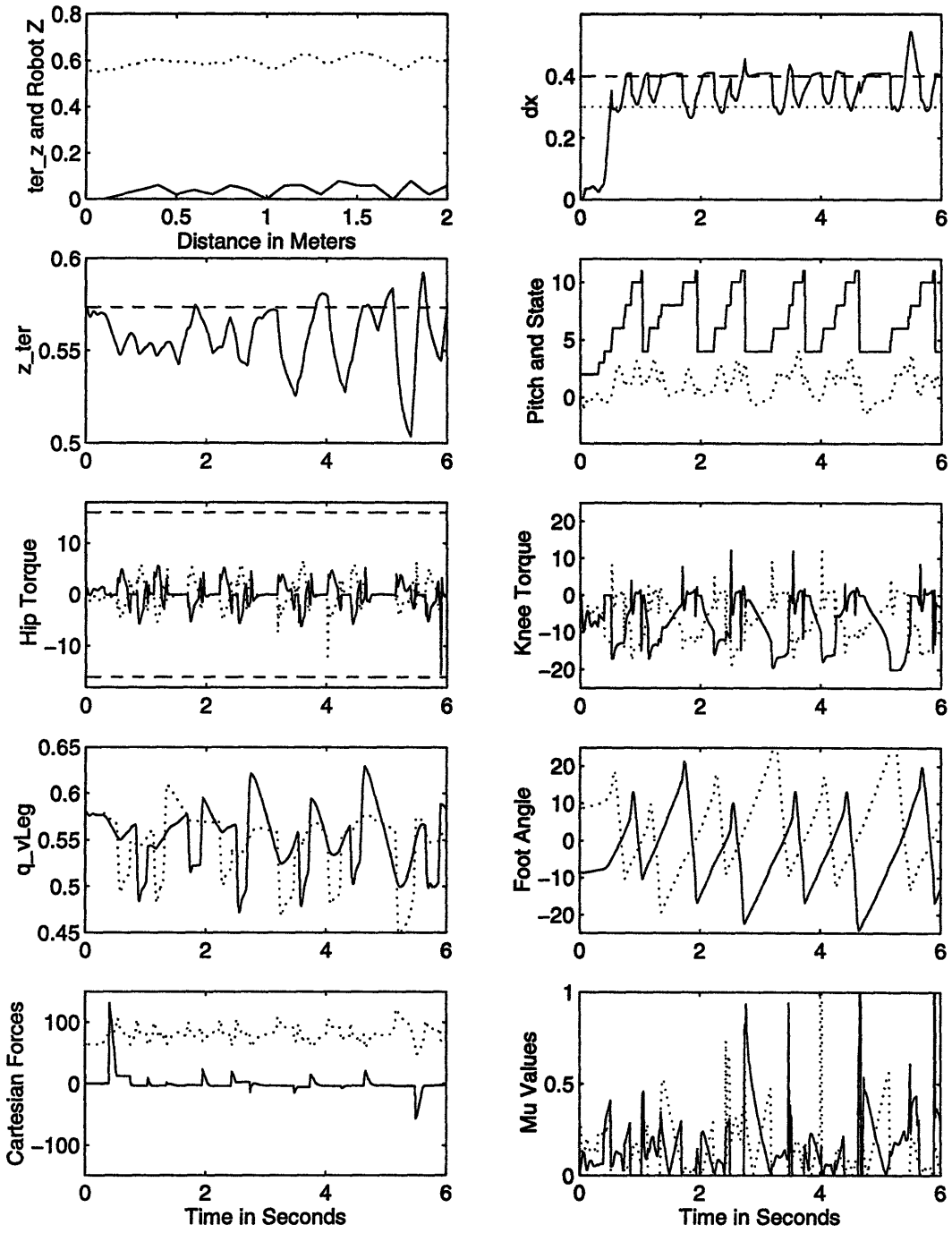

Figure 5-6: KWALKB performance on very rough, level ground. All units in MKS, with angles in degrees. Dotted traces indicate the left side for leg data graphs.

5.2 Sloping Terrain

5.2.1 Uphill

The next two groups of graphs represent KWALKA and KWALKB climbing the steepest slope they could. The up hill performance of KWALKA is so exceptional, the required friction became the limiting factor. The simulation could march right up a 50 degree incline, but the coefficient of friction would have to be well over 1. Unless the robot wears needle spikes and walks up a soft rubber slope, this kind of friction is unrealistic. For fear of having to equip a walking robot with weapons on its feet in order to verify these simulated results, I limited the slope to a 35 degree incline. This is the steepest hill which kept the coefficient of friction below 1. KWALKB was less agile on uphill grades, but still managed to climb a 26 degree incline.

KWALKA's 5 initial steps on level ground nicely demonstrate the changes in the behavior when the ground slopes up. The body height drops, the velocity stays lower, and the step rate doubles. Knee forces drop to a slightly lower minimum, and the friction requirements increase dramatically. The general motion improves dramatically on the incline. The feet swing forward and just land on the slope. States 7 and 11, the single to double transitions, take nearly no time. State 7 is too short to show up in the graph.

KWALKB's performance also smoothes dramatically on the incline. The range of X velocities drops by 40 percent, and the body height oscillation flattens to 25 percent of its level ground amplitude. The steady state body height error almost doubles, however, and the body takes on a three degree average forward pitch. As with the other model, the step rate also doubled.

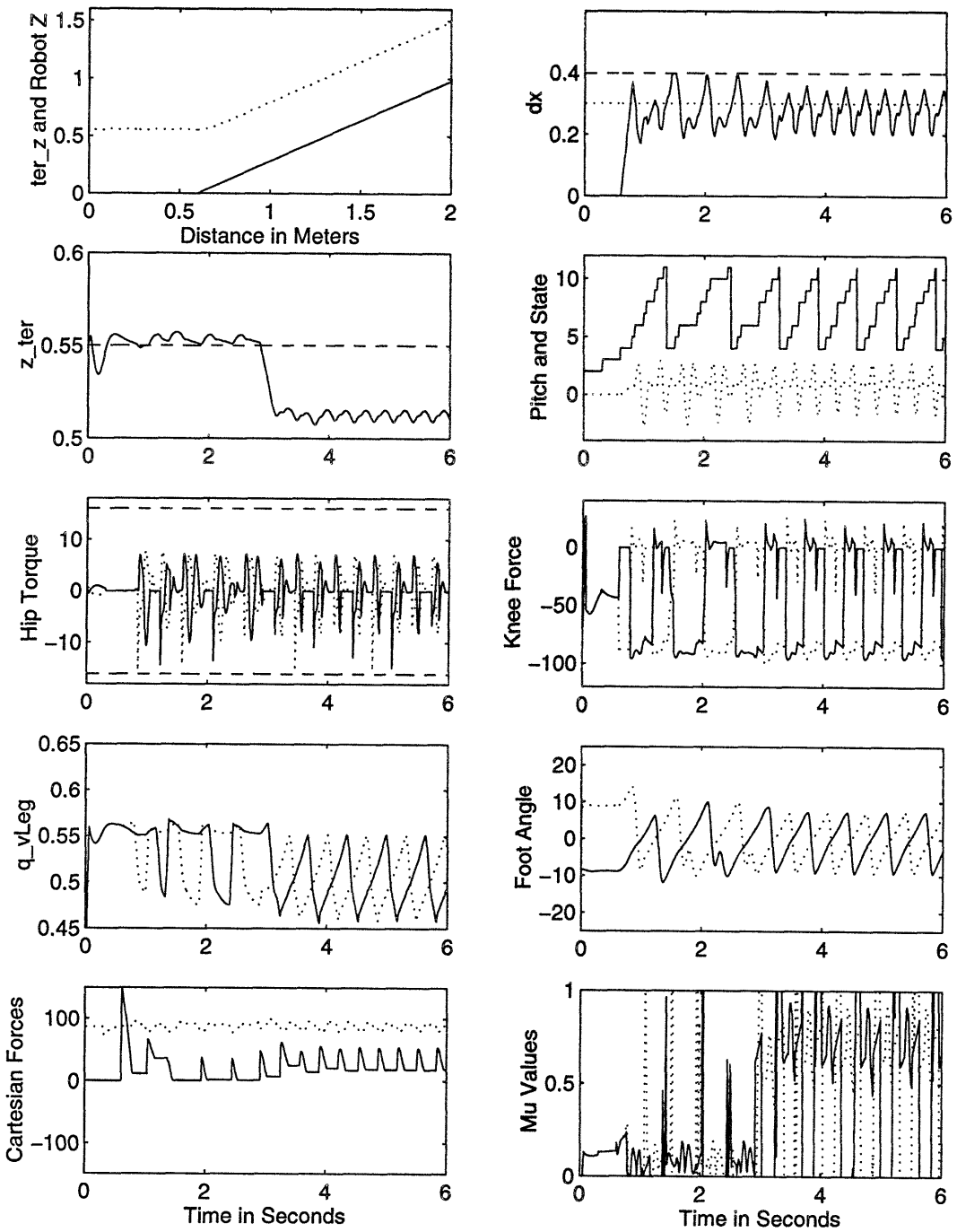

Figure 5-7: KWALKA performance on a smooth, uphill slope. All units in MKS, with angles in degrees. Dotted traces indicate the left side for leg data graphs.

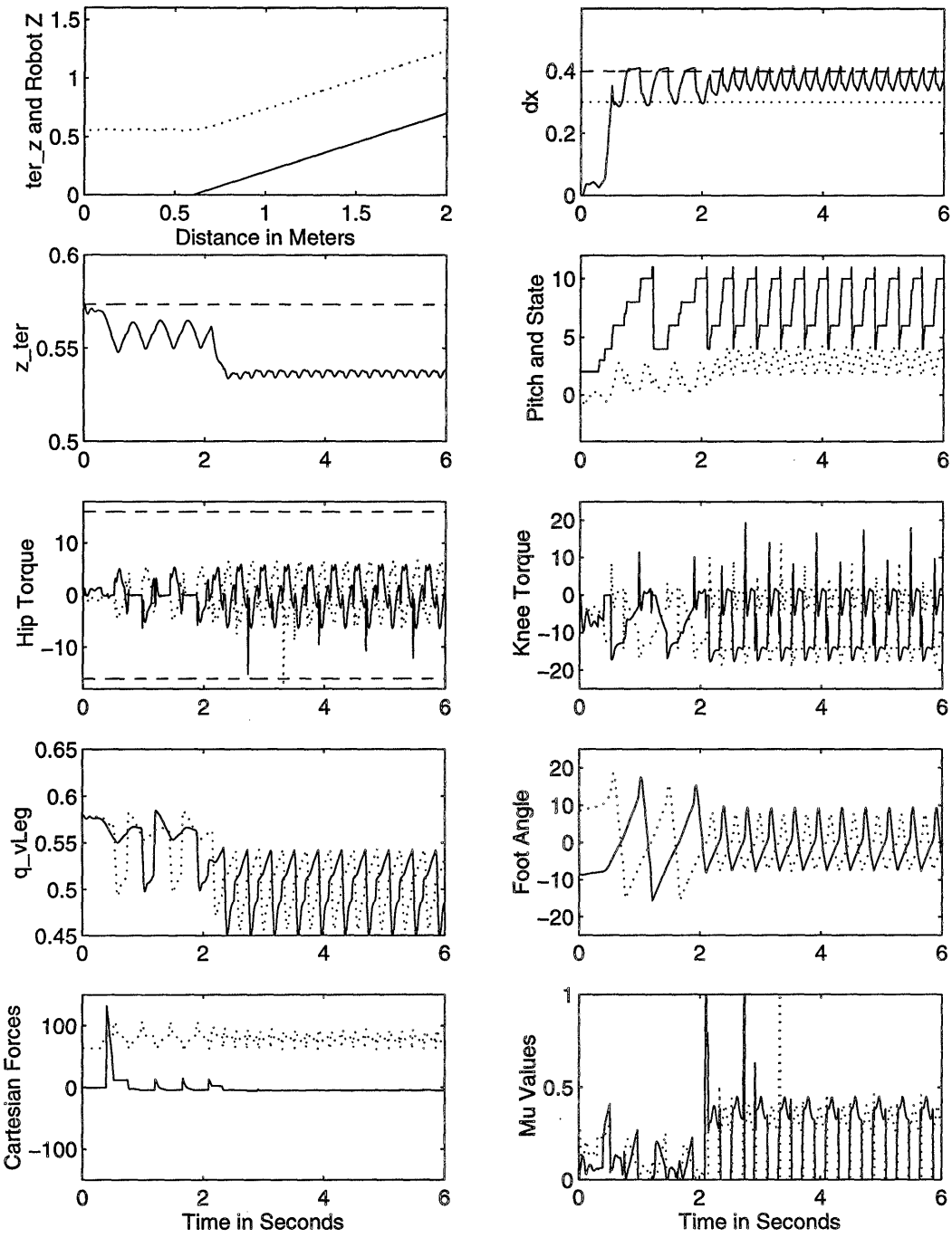

Figure 5-8: KWALKB performance on a smooth, uphill slope. All units in MKS, with angles in degrees. Dotted traces indicate the left side for leg data graphs.

5.2.2 Downhill

KWALKA performs fairly well on down hill slopes. It managed to descend a 35 degree incline in control. The body height set-point had to be reduced to .45 meters to give the legs sufficient room to extend down to the slope in front of the robot. A positive steady state error kept the body about 8 percent above the set-point. The legs extended over a larger range of lengths, biased shorter than on level ground by the lower body height. A much larger friction was required.

KWALKB lacked the ability to shorten its body height sufficiently. Every attempt at downhill slopes ended with the legs collapsing or flailing in space. While it is possible that the right combination of body height, stance width, velocity, etc. could make KWALKB descend a slope in control, I could not find one with the torque limits I set.

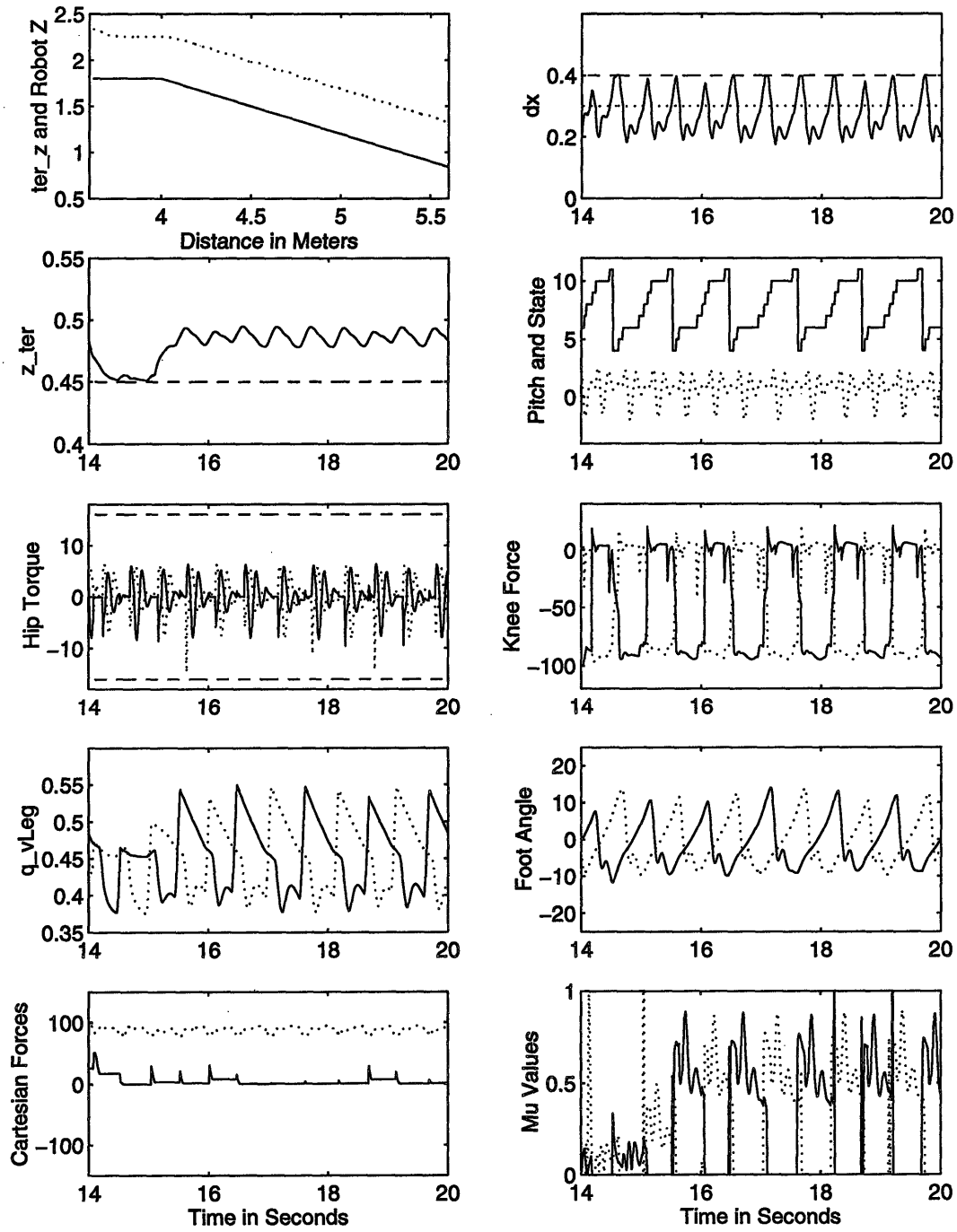

Figure 5-9: KWALKA performance on a smooth, downhill slope. All units in MKS, with angles in degrees. Dotted traces indicate the left side for leg data graphs.

Chapter 6

Conclusions and Future Work

The ability to walk across rough terrain on two legs without knowledge of the upcoming terrain is clearly demonstrated by both models. Dynamically adjusting for what each foot lands on can create a stable gait. The successes, failures, approximations, and data from this work lead to a number of conclusion and topics for other work.

6.1 Conclusions

The first discovery was that rough terrain is not as much harder than smooth terrain walking as smooth terrain walking is to begin with. Many smooth terrain algorithms can be easily adapted to deal with slight bumps and upward slopes. The level of difficulty seems to increase with decreasing slope. Uphill walking requires sufficient power, but is easy to control. Legs cannot easily swing too far forward, so a large number of swing techniques can work. As the slope becomes less steep, over-leading with the swing leg can cause problems as easily as under rotating it.

As the slope starts going down, it becomes more and more critical to place the lead foot at a point far enough forward to control the velocity; however, limited leg length constrains how far forward this point can be. At some point, the leg may not be able to extend far enough to reach a point suitable to control the velocity. The

body height becomes an important factor. The support leg must be able to operate at perhaps 50 percent maximum length in order to put the swing leg in reach of the ground head of it. This was the primary problem with KWALKB on downhill slopes. Controlled descent turns out to be the most demanding on a system.

A correlation between the severity of the terrain and power requirements was also evidenced in this work. A need for sufficient power over a broad portion of a limb's workspace is very important for dealing with unknown terrain. If the ground is extremely rough, high clearances are necessary, and it is likely that the feet will hit with the leg significantly shortened. With revolute joints, this can easily fall into the under powered region of the actuator. This interaction between the force available and the location in the work space makes it rather surprising that legs work as well as they do in nature. Artificial actuator efficiency has a long way to go before it can compare to natural systems.

The issue of revolute joints versus prismatic joints for legged locomotion is also incorporated in this work. Revolute knees work well for motions about their vertical point, since the structure takes most of the load and not the actuator. When significant flexion is required, they require considerable torque to produce weight-bearing support. Prismatic knees put the entire force on the joint's free axis, so the actuator must always be active to balance the weight of a load. For walking efficiency, the revolute joints are clearly the better system. With time spent standing around as well, there is no comparison. However, dynamic requirements and range and ease of effective control are issues which can justify prismatic devices. With pneumatic or hydraulic power, linear motion becomes preferable. This all sums up to the fact that the better system depends on the specific needs of the device.

Direct performance comparisons between the KWALKA and KWALKB models is not entirely possible from this work, due to an approach geared more for functionality than precision. This work represents a first cut at the issues, and makes some approximations which differ between the two models. Neglecting the swing leg reaction is one approximation which affects KWALKA much more than it does

KWALKB. An added factor in the effectiveness of the velocity control is the body height error. Because the actual body height is lower on KWALKB for the same set-point, the foot placement becomes less of a factor. Also, the swing leg has a smaller moment of inertia when it is raised higher.

6.2 Opportunities for Further Work

Some opportunities for further work involve refinements of the work presented here. The primary one is a more rigorous modeling approach to isolate the control system so that it deals only with virtual leg parameters. This work represents one control system adapted to two different models. The goal was to show that it can work and to simulate the systems on rough terrain. An approach directed at abstracting the models to a standard form and using the same form of the controller for each can draw stronger comparisons between the two models.

Another factor not addressed in the simulation but considered in the analysis is slippage. Interesting developments may come out of making the feet slide when the net force breaks the cone of friction. This would add another limit in the realm of actuator force and torque limits. Not only would the force on the feet be limited by what the leg can produce, but it would also be constrained by what the ground can resist.

Adaptive control issues lead to other possible extensions. Applying learning control or doing gain scheduling of control parameters based on current ground slope can add to the range of general slopes the algorithm can handle and improve the performance. If parameters never stray far enough for one step to cause a fall, the system can adapt itself to the terrain it encounters based on predetermined optimum values for the current slope. It could even find optimum values by itself if it can reset itself after falling. It could search the parameter space automatically, using velocity, altitude, and pitch deviations to evaluate the performance.

For any system which is a subset of another, larger system, the opportunity is

always available to expand. Both the terrain and the robot are essentially planar. While the robots legs are offset in the Y direction, no bank, yaw, or lateral motion is allowed. The terrain is also uniform along the Y direction. A natural extension would be to introduce terrain which varies in Y as well, and to add the other three degrees of freedom to the models. This is well beyond the scope of this project; however, it would prove to be interesting work.

Appendix A

Header Code Listings

A.1 Create Header Files

```
/*****************************************************************
 *  CREATE_KWALKA Header File  (same as CREATE_KWALKB)   *
 *  #DEFINEs for robot parameters    *
 *  and Macros    *
 *****************************************************************/

#define G  9.81

#define ID_RATIO 1.0   /* Makes X for right thigh this much thicker
  and Y for left thigh  */

/*  Density dividers to make volume of robot match approximate mass  */
#define BM_RATIO  7.48
#define TM_RATIO  0.65
#define SM_RATIO  1.70

#define BODY_X  .30
#define BODY_Y  .30
#define BODY_Z  .20

#define THIGH_X .025
#define THIGH_Y .035
#define THIGH_Z .32
```

```
#define SHANK_X .025
#define SHANK_Y .035
#define SHANK_Z .32
```

A.2 KWALKA Control Header File

```
/*****************************************************************
 *  KWALKA Header File   *
 *  #DEFINEs for control parameters    *
 *  and Macros   *
 *****************************************************************/

#define HIP_K       150.0
#define KNEE_K      800.0
#define KNEE_AIR_K  200.0
#define  DX_KV       400.0
#define  BODYZ_K     600.0

#define HIP_ZETA      0.85
#define KNEE_ZETA     0.85
#define BODYZ_ZETA    0.95
#define BODYDX_ZETA   0.95

#define FF_K          0.0
#define FF_B          0.0

#define F_KNEE_DOWN  -40.0
#define B_KNEE_DOWN   20.0

#define RIGHT        0
#define LEFT         1
#define PD_MODE      1
#define PD_FF_MODE   2
#define FF_MODE      3
#define FF_B_MODE    4
#define SINGLE       0
#define DOUBLE       1
#define SINGLE_DOWN  2

#define INIT_HIP        .15

#define F_KNEE_MAX       (-.2)
#define F_KNEE_MIN     (-100)
```

```
#define MAX_H_TORQ     15.0
#define MIN_H_TORQ   (-15.0)

#define Z_NOM           .55
#define DX_NOM          .4
#define DX_MIN          .3
#define THETA_MIN      (-.15)
#define SWING_LEAD     (-.10)
#define SWING_HI        .10
#define SWING_H         .10
#define G_CLEARANCE     .04

#define MAX_TERRAIN_ANGLE  (PI/4.0)

#define UP_Z_NOM          .50
#define UP_DX_NOM         .4
#define UP_DX_MIN         .3
#define UP_THETA_MIN     (-.15)
#define UP_SWING_LEAD    (-.10)
#define UP_SWING_HI       .20
#define UP_SWING_H        .20
#define UP_G_CLEARANCE    .04

#define MIN_TERRAIN_ANGLE  (-35.0*PI/180.0)

#define DN_Z_NOM          .50
#define DN_DX_NOM         .4
#define DN_DX_MIN         .3
#define DN_THETA_MIN     -.15
#define DN_SWING_LEAD    -.05
#define DN_SWING_HI       .20
#define DN_SWING_H        .20
#define DN_G_CLEARANCE    .04

#define THETA_R   (q.pitch+q.rhip)
#define THETA_L   (q.pitch+q.lhip)
#define Q_LKNEE   (THIGH_Z - q.lknee)
#define Q_RKNEE   (THIGH_Z - q.rknee)

#define ABS(P)  ((P) > 0 ? (P) : (-(P)))
#define SIGN(P) ((P) > 0 ? (1) : (-1))
#define MIN(P,Q) ((P) < (Q) ? (P) : (Q))
#define MAX(P,Q) ((P) > (Q) ? (P) : (Q))
```

A.3 KWALKB Control Header File

```
/*****************************************************************
 *  KWALKB Header File   *
 *  #DEFINEs for control parameters    *
 *  and Macros   *
 *****************************************************************/
#define G  9.81

#define HIP_K       150.0
#define KNEE_K      300.0
#define KNEE_AIR_K   50.0
#define DX_KV       400.0
#define BODYZ_K     600.0

#define HIP_ZETA      0.9
#define KNEE_ZETA     0.9
#define BODYZ_ZETA    0.6
#define BODYDX_ZETA   0.9

#define FF_K          0.0
#define FF_B          0.0

#define F_KNEE_DOWN (-100.0)
#define B_KNEE_DOWN    1.0

#define TOOSMALL      .001

#define RIGHT        0
#define LEFT         1
#define HIP          0
#define KNEE         1
#define FOOT         2
#define DOUBLE       0
#define SINGLE       1
#define SINGLE_DOWN  2

#define PD_MODE      1
#define PD_FF_MODE   2
#define FF_MODE      3
#define FF_B_MODE    4

#define DX_NOM        .4
#define DX_MIN        .3
```

```
#define INIT_FOOTA    .15
#define INIT_FOOTD    .58

#define F_KNEE_MAX   (-.2)
#define MIN_K_TORQ   (-20.0)
#define MAX_K_TORQ    20.0
#define MIN_H_TORQ   (-15.0)
#define MAX_H_TORQ    15.0

#define THETA_MIN      (-.10)
#define SWING_LEAD     (-.15)
#define SWING_HI        .05
#define SWING_H         .05
#define G_CLEARANCE     .02

#define MAX_TERRAIN_ANGLE  (PI/4.0)

#define UP_Z_NOM           .50
#define UP_DX_NOM          .4
#define UP_DX_MIN          .3
#define UP_THETA_MIN      (-.15)
#define UP_SWING_LEAD     (-.10)
#define UP_SWING_HI        .20
#define UP_SWING_H         .20
#define UP_G_CLEARANCE     .04

#define MIN_TERRAIN_ANGLE  (-35.0*PI/180.0)

#define DN_Z_NOM           .50
#define DN_DX_NOM          .4
#define DN_DX_MIN          .3
#define DN_THETA_MIN      (-.15)
#define DN_SWING_LEAD     (-.05)
#define DN_SWING_HI        .20
#define DN_SWING_H         .20
#define DN_G_CLEARANCE     .04


#define ABS(P)  ((P) > 0 ? (P) : (-(P)))
#define SIGN(P) ((P) > 0 ? (1) : (-1))
#define MIN(P,Q) ((P) < (Q) ? (P) : (Q))
#define MAX(P,Q) ((P) > (Q) ? (P) : (Q))
```

Appendix B

Creature Library Code

B.1 Create_KWALKA Code

```
/*****************************************************************
 *  create_kwalka.c    *
 *****************************************************************/

#include <cl_lib.h>
#include "create_kwalka.h"

main(argc,argv)
     int argc;
     char *argv[];
{
  command_line(argc, argv);

  begin_species("kwalka");

  use_density(ALUMINUM_DENSITY/BM_RATIO);

  new_joint("boom");
  set_planar_joint("x","z","pitch",'x','z','y');

  new_link("body");
  set_legplot_link_size(-BODY_X/2.0,BODY_X/2.0,
-BODY_Y/2.0,BODY_Y/2.0,
0.0,BODY_Z);
  begin_shape();
  translate(0.0, 0.0, -BODY_Z/2);
  shape(SBRICK, BODY_X, BODY_Y, BODY_Z);
  end_shape();
```

```
  joint_pin("lhip", 'y');
  servo(1,"lhip",PD_SERVO,"ls.k_lhip","ls.b_lhip");
  servo(2,"lhip",PD_FF_SERVO,"ls.k_lhip","ls.b_lhip","ls.ff_lhip");
  servo(3,"lhip",PD_FF_SERVO,"ls.k_ff","ls.b_ff","ls.ff_lhip");
  set_joint_offset(0.0, BODY_Y/2, 0.0);
  limit("lhip",-PI/2.0 , PI/2.0);

  use_density(ALUMINUM_DENSITY/TM_RATIO);

  new_link("lthigh");
  set_legplot_link_size(-THIGH_X/2.0,THIGH_X/2.0,
-ID_RATIO*THIGH_Y/2.0,ID_RATIO*THIGH_Y/2.0,
-THIGH_Z, 0.0);
  begin_shape();
  translate(0.0, 0.0, -THIGH_Z);
  shape(SBRICK, THIGH_X, ID_RATIO*THIGH_Y, THIGH_Z);
  end_shape();

  joint_slider("lknee",'z');
  servo(1,"lknee",PD_SERVO,"ls.k_lknee","ls.b_lknee");
  servo(2,"lknee",PD_FF_SERVO,"ls.k_lknee","ls.b_lknee"
,"ls.ff_lknee");
  servo(3,"lknee",PD_FF_SERVO,"ls.k_ff","ls.b_ff","ls.ff_lknee");
  servo(4,"lknee",PD_FF_SERVO,"ls.k_ff","ls.b_ff_b","ls.ff_lknee");
  set_joint_offset(0.0, 0.0, -THIGH_Z);
  limit("lknee",0.0,SHANK_Z);

  use_density(ALUMINUM_DENSITY/SM_RATIO);

  new_link("lshank");
  set_legplot_link_size(-SHANK_X/2.0,SHANK_X/2.0,
-SHANK_Y/2.0,SHANK_Y/2.0,
-SHANK_Z, 0.0);

  begin_shape();
  translate(0.0, 0.0, -SHANK_Z);
  shape(SBRICK, SHANK_X, SHANK_Y, SHANK_Z);
  end_shape();

  ground_contact("lfoot", 0.0, 0.0, -SHANK_Z);


  joint_pin("rhip", 'y');
  servo(1,"rhip",PD_SERVO,"ls.k_rhip","ls.b_rhip");
  servo(2,"rhip",PD_FF_SERVO,"ls.k_rhip","ls.b_rhip","ls.ff_rhip");
```

```
  servo(3,"rhip",PD_FF_SERVO,"ls.k_ff","ls.b_ff","ls.ff_rhip");
  set_parent("body");
  set_joint_offset(0.0,-BODY_Y/2, 0.0);
  limit("rhip",-PI/2.0 , PI/2.0);

  use_density(ALUMINUM_DENSITY/TM_RATIO);

  new_link("rthigh");
  set_legplot_link_size(-ID_RATIO*THIGH_X/2.0,ID_RATIO*THIGH_X/2.0,
-THIGH_Y/2.0,THIGH_Y/2.0,
-THIGH_Z, 0.0);

  begin_shape();
  translate(0.0, 0.0, -THIGH_Z);
  shape(SBRICK, ID_RATIO*THIGH_X, THIGH_Y, THIGH_Z);
  end_shape();

  joint_slider("rknee",'z');
  servo(1,"rknee",PD_SERVO,"ls.k_rknee","ls.b_rknee");
  servo(2,"rknee",PD_FF_SERVO,"ls.k_rknee","ls.b_rknee"
,"ls.ff_rknee");
  servo(3,"rknee",PD_FF_SERVO,"ls.k_ff","ls.b_ff","ls.ff_rknee");
  servo(4,"rknee",PD_FF_SERVO,"ls.k_ff","ls.b_ff_b","ls.ff_rknee");
  set_joint_offset(0.0,0.0,-THIGH_Z);
  limit("rknee",0.0,SHANK_Z);

  use_density(ALUMINUM_DENSITY/SM_RATIO);

  new_link("rshank");
  set_legplot_link_size(-SHANK_X/2.0,SHANK_X/2.0,
-SHANK_Y/2.0,SHANK_Y/2.0,
-SHANK_Z, 0.0);

  begin_shape();
  translate(0.0, 0.0, -SHANK_Z);
  shape(SBRICK, SHANK_X, SHANK_Y, SHANK_Z);
  end_shape();

  ground_contact("rfoot", 0.0, 0.0, -SHANK_Z);

  end_species();
}
```

B.2 Create_KWALKB Code

```
/*****************************************************************
 *  create_kwalkb.c      *
 *****************************************************************/

#include <cl_lib.h>
#include "create_kwalkb.h"

main(argc,argv)
     int argc;
     char *argv[];
{
  command_line(argc, argv);

  begin_species("kwalkb");

  use_density(ALUMINUM_DENSITY/BM_RATIO);

  new_joint("boom");
  set_planar_joint("x","z","pitch",'x','z','y');

  new_link("body");
  set_legplot_link_size(-BODY_X/2.0,BODY_X/2.0,
-BODY_Y/2.0,BODY_Y/2.0,
0.0,BODY_Z);
  begin_shape();
  translate(0.0, 0.0, -BODY_Z/2);
  shape(SBRICK, BODY_X, BODY_Y, BODY_Z);
  end_shape();

  joint_pin("lhip", 'y');
  servo(1,"lhip",PD_SERVO,"ls.k_lhip","ls.b_lhip");
  servo(2,"lhip",PD_FF_SERVO,"ls.k_lhip","ls.b_lhip","ls.ff_lhip");
  servo(3,"lhip",PD_FF_SERVO,"ls.k_ff","ls.b_ff","ls.ff_lhip");
  servo(4,"lhip",PD_FF_SERVO,"ls.k_ff","ls.b_ff_b","ls.ff_lhip");
  set_joint_offset(0.0, BODY_Y/2, 0.0);
  limit("lhip", -PI/2.0, PI/2.0);

  use_density(ALUMINUM_DENSITY/TM_RATIO);

  new_link("lthigh");
  set_legplot_link_size(-THIGH_X/2.0,THIGH_X/2.0,
-ID_RATIO*THIGH_Y/2.0,ID_RATIO*THIGH_Y/2.0,
-THIGH_Z, 0.0);
```

```
  begin_shape();
  translate(0.0, 0.0, -THIGH_Z);
  shape(SBRICK, THIGH_X, ID_RATIO*THIGH_Y, THIGH_Z);
  end_shape();

  joint_pin("lknee",'y');
  servo(1,"lknee",PD_SERVO,"ls.k_lknee","ls.b_lknee");
  servo(2,"lknee",PD_FF_SERVO,"ls.k_lknee","ls.b_lknee","ls.ff_lknee");
  servo(3,"lknee",PD_FF_SERVO,"ls.k_ff","ls.b_ff","ls.ff_lknee");
  servo(4,"lknee",PD_FF_SERVO,"ls.k_ff","ls.b_ff_b","ls.ff_lknee");
  set_joint_offset(0.0, 0.0, -THIGH_Z);
  limit("lknee",-PI/2 - INVERT*PI/2, PI/2 - INVERT*PI/2);

  use_density(ALUMINUM_DENSITY/SM_RATIO);

  new_link("lshank");
  set_legplot_link_size(-SHANK_X/2.0,SHANK_X/2.0,
-SHANK_Y/2.0,SHANK_Y/2.0,
-SHANK_Z, 0.0);

  begin_shape();
  rotate('y',180.0);
  translate(0.0, 0.0, 0.0);
  shape(SBRICK, SHANK_X, SHANK_Y, SHANK_Z);
  end_shape();

  ground_contact("lfoot", 0.0, 0.0, -SHANK_Z);


  joint_pin("rhip", 'y');
  servo(1,"rhip",PD_SERVO,"ls.k_rhip","ls.b_rhip");
  servo(2,"rhip",PD_FF_SERVO,"ls.k_rhip","ls.b_rhip","ls.ff_rhip");
  servo(3,"rhip",PD_FF_SERVO,"ls.k_ff","ls.b_ff","ls.ff_rhip");
  servo(4,"rhip",PD_FF_SERVO,"ls.k_ff","ls.b_ff_b","ls.ff_rhip");
  set_parent("body");
  set_joint_offset(0.0,-BODY_Y/2, 0.0);
  limit("rhip", -PI/2.0, PI/2.0);

  use_density(ALUMINUM_DENSITY/TM_RATIO);

  new_link("rthigh");
  set_legplot_link_size(-ID_RATIO*THIGH_X/2.0,ID_RATIO*THIGH_X/2.0,
-THIGH_Y/2.0,THIGH_Y/2.0,
-THIGH_Z, 0.0);

  begin_shape();
```

```
  translate(0.0, 0.0, -THIGH_Z);
  shape(SBRICK, ID_RATIO*THIGH_X, THIGH_Y, THIGH_Z);
  end_shape();

  joint_pin("rknee",'y');
  servo(1,"rknee",PD_SERVO,"ls.k_rknee","ls.b_rknee");
  servo(2,"rknee",PD_FF_SERVO,"ls.k_rknee","ls.b_rknee","ls.ff_rknee");
  servo(3,"rknee",PD_FF_SERVO,"ls.k_ff","ls.b_ff","ls.ff_rknee");
  servo(4,"rknee",PD_FF_SERVO,"ls.k_ff","ls.b_ff_b","ls.ff_rknee");
  set_joint_offset(0.0,0.0,-THIGH_Z);
  limit("rknee",-PI/2 - INVERT*PI/2, PI/2 - INVERT*PI/2);

  use_density(ALUMINUM_DENSITY/SM_RATIO);

  new_link("rshank");
  set_legplot_link_size(-SHANK_X/2.0,SHANK_X/2.0,
-SHANK_Y/2.0,SHANK_Y/2.0,
-SHANK_Z, 0.0);

  begin_shape();
  rotate('y',180.0);
  translate(0.0, 0.0, 0.0);
  shape(SBRICK, SHANK_X, SHANK_Y, SHANK_Z);
  end_shape();

  ground_contact("rfoot", 0.0, 0.0, -SHANK_Z);

  end_species();
}
```

Appendix C

Control Code

C.1 KWALKA Control Code

Following are the functions I wrote to implement the KWALKA control system within the structure provided by the Leg Lab's existing simulation package. User_gcontact is part of Peter Dilworth's terrain module.

```
/* ************************************************************
   control1()
     Called every control cycle.  Should contain or call the
     actual creature control code.
   ************************************************************ */
void
control1()
{

  adapt_control();

  switch ((int)ls.state) {

/*     Initialize robot    */
  case 0:

    onetime_init();

    ls.state=1;

    break;

  case 1:
```

```
    init_control();

    ls.state = 2;
    break;

/*  Wait for things to settle  */

  case 2:

    q_d.rknee = SHANK_Z - ((q_d.z)/cos(THETA_L) - THIGH_Z);
    q_d.lknee = SHANK_Z - ((q_d.z)/cos(THETA_R) - THIGH_Z);

    if(ls.t>0.3)
      {
ls.state = 3;
/* Soften position control for picking up and putting
   down legs  */
ls.k_lknee = KNEE_AIR_K;
ls.k_rknee = KNEE_AIR_K;
set_damping_from_k_zeta();

servosw.lhip = FF_MODE;
servosw.lknee = FF_MODE;
servosw.rhip = FF_MODE;
servosw.rknee = FF_MODE;
      }
    break;

/*     Switch to force control   */
  case 3:

    ls.Fz_d = ls.M*G+ls.k_bodyz*(q_d.z-ls.z_ter)
      +ls.b_bodyz*(qd_d.z - qd.z);

    ls.Fx_d = 0.0;

    compute_knee_ffs(DOUBLE,RIGHT);

/*  Wait for a bit to make sure this stays stable  */
    if(ls.t>0.6){
      ls.state = 4;
      servosw.rhip = PD_MODE;
      set_damping_from_k_zeta();
    }
    break;
```

```
/*     Double support, right leg forward   */
  case 4:

    ls.Fz_d = ls.M*G+ls.k_bodyz*(q_d.z-ls.z_ter)
      +ls.b_bodyz*(qd_d.z - qd.z);

    ls.Fx_d = ls.kvdx * (qd_d.x - qd.x);

    compute_knee_ffs(DOUBLE,RIGHT);

    /*  Stabilize pitch with front leg;  */
    if (gc_rfoot.fs){
      q_d.rhip  = MAX(MIN((q.rhip+MAX_H_TORQ/ls.k_rhip),THETA_R),
      (q.rhip+MIN_H_TORQ/ls.k_rhip));
    }

    if (THETA_R > 0.0 ||
(pow(DX_MIN,2.0) <=
 pow(qd.x,2.0) - ls.z_ter*G*pow(tan(THETA_R),2.0)))
      {
ls.state = 5;
ls.foot_min=q.lknee + ls.g_clearance;
q_d.lknee = q.lknee + ls.swing_hi;
ls.ff_lknee = -M_lshank*G*cos(THETA_L);
servosw.lknee = PD_FF_MODE;
      }
    break;

/*     Pick up the swing leg   */
  case 5:

    ls.Fz_d = ls.M1*G+ls.k_bodyz*(q_d.z-ls.z_ter)
      +ls.b_bodyz*(qd_d.z - qd.z);

    compute_knee_ffs(SINGLE,RIGHT);

/*  Now position control the right hip to keep the body level  */

    if (gc_rfoot.fs){
      q_d.rhip  = MAX(MIN((q.rhip+MAX_H_TORQ/ls.k_rhip),THETA_R),
      (q.rhip+MIN_H_TORQ/ls.k_rhip));
    }

    if (q_d.lknee < q.rknee){
      q_d.lknee = q.rknee + ls.swing_h;
    }
```

```
    if (q.lknee > q.rknee && q.lknee > ls.foot_min && !gc_lfoot.fs)
      {
ls.state = 6;
q_d.lhip  = MAX(MIN((q.lhip+MAX_H_TORQ/ls.k_lhip),
    (-THETA_R - q.pitch + ls.swing_lead)),
(q.lhip+MIN_H_TORQ/ls.k_lhip));
servosw.lhip = PD_MODE;
servosw.lknee = PD_MODE;
      }
    break;

/*     Move through single support on right leg  */
  case 6:

    ls.Fz_d = ls.M1*G+ls.k_bodyz*(q_d.z-ls.z_ter)
      +ls.b_bodyz*(qd_d.z - qd.z);

    compute_knee_ffs(SINGLE,RIGHT);

/*  Mirror global angle of support leg with swing leg */

    q_d.lhip  = MAX(MIN((q.lhip+MAX_H_TORQ/ls.k_lhip),
(-THETA_R - q.pitch + ls.swing_lead)),
    (q.lhip+MIN_H_TORQ/ls.k_lhip));

    if (q.lhip < 0 || q_d.lknee < q.rknee){
      q_d.lknee = q.rknee + ls.swing_h;
    }

/*  Now position control the right hip to keep the body level  */

    if (gc_rfoot.fs){
      q_d.rhip  = MAX(MIN((q.rhip+MAX_H_TORQ/ls.k_rhip),THETA_R),
      (q.rhip+MIN_H_TORQ/ls.k_rhip));
    }

    if (THETA_L < ls.theta_min && q.rhip > q.lhip){
      ls.state = 7;
      servosw.lknee = FF_B_MODE;
    }

    if (gc_lfoot.fs)
      {
if (q.rhip > q.lhip)
  {
```

```
    ls.state = 8;
    servosw.lknee = FF_MODE;
    servosw.rhip =  FF_MODE;
    adapt_control();
  }
else
  {
    ls.state = 4;
    ls.foot_min=q.lknee + ls.g_clearance;
    servosw.lhip = FF_MODE;
    servosw.lknee = FF_MODE;
  }
      }
    break;

/*  Put the left leg back on the ground  */
  case 7:

    ls.Fz_d = ls.M1*G+ls.k_bodyz*(q_d.z-ls.z_ter)
      +ls.b_bodyz*(qd_d.z - qd.z);

    compute_knee_ffs(SINGLE_DOWN,RIGHT);

    if (gc_rfoot.fs){
      q_d.rhip  = MAX(MIN((q.rhip+MAX_H_TORQ/ls.k_rhip),THETA_R),
      (q.rhip+MIN_H_TORQ/ls.k_rhip));
    }

    if (gc_lfoot.fs){
      ls.state=8;
      servosw.lknee = FF_MODE;
      servosw.rhip =  FF_MODE;
      adapt_control();
    }
    break;


/*     Double support, left leg forward   */
  case 8:

    ls.Fz_d = ls.M*G+ls.k_bodyz*(q_d.z-ls.z_ter)
      +ls.b_bodyz*(qd_d.z - qd.z);

    ls.Fx_d = ls.kvdx * (qd_d.x - qd.x);

    compute_knee_ffs(DOUBLE,LEFT);
```

```
    if (gc_lfoot.fs){
      q_d.lhip  = MAX(MIN((q.lhip+MAX_H_TORQ/ls.k_lhip),THETA_L),
      (q.lhip+MIN_H_TORQ/ls.k_lhip));
    }

    if (THETA_L > 0.0 ||
(pow(DX_MIN,2.0) <=
 pow(qd.x,2.0) - ls.z_ter*G*pow(tan(THETA_L),2.0)))
      {
ls.state = 9;
ls.foot_min=q.rknee + ls.g_clearance;
q_d.rknee = q.rknee + ls.swing_hi;
ls.ff_rknee = -M_rshank*G*cos(THETA_R);
servosw.rknee = PD_FF_MODE;
      }
    break;

/*     Pick up the swing leg (right leg)  */
  case 9:

    ls.Fz_d = ls.M1*G+ls.k_bodyz*(q_d.z-ls.z_ter)
      +ls.b_bodyz*(qd_d.z - qd.z);

    compute_knee_ffs(SINGLE,LEFT);

/*  Now position control the left hip to keep the body level  */

    if (gc_lfoot.fs){
      q_d.lhip  = MAX(MIN((q.lhip+MAX_H_TORQ/ls.k_lhip),THETA_L),
      (q.lhip+MIN_H_TORQ/ls.k_lhip));
    }

    if (q_d.rknee < q.lknee){
      q_d.rknee = q.lknee + ls.swing_h;
    }

    if (q.rknee > q.lknee && q.rknee > ls.foot_min && !gc_rfoot.fs)
      {
ls.state = 10;
q_d.rhip  = MAX(MIN((q.rhip+MAX_H_TORQ/ls.k_rhip),
    (-THETA_L-q.pitch + ls.swing_lead)),
(q.lhip+MIN_H_TORQ/ls.k_lhip));
servosw.rhip =  PD_MODE;
servosw.rknee = PD_MODE;
      }
```

```
    break;

/*     Move through single support on left leg  */
  case 10:

    ls.Fz_d = ls.M1*G+ls.k_bodyz*(q_d.z-ls.z_ter)
      +ls.b_bodyz*(qd_d.z - qd.z);

    compute_knee_ffs(SINGLE,LEFT);

/*  Mirror global angle of support leg with swing leg */

    q_d.rhip  = MAX(MIN((q.rhip+MAX_H_TORQ/ls.k_rhip),
(-THETA_L-q.pitch + ls.swing_lead)),
    (q.rhip+MIN_H_TORQ/ls.k_rhip));

    if (q.rhip < 0 || q_d.rknee < q.lknee){
      q_d.rknee = q.lknee + ls.swing_h;
    }

/*  Now position control the left hip to keep the body level  */

    if (gc_lfoot.fs){
      q_d.lhip  = MAX(MIN((q.lhip+MAX_H_TORQ/ls.k_lhip),THETA_L),
      (q.lhip+MIN_H_TORQ/ls.k_lhip));
    }

    if (q.lhip>q.rhip && THETA_R < ls.theta_min)
      {
ls.state = 11;
servosw.rknee = FF_B_MODE;
      }

    if (gc_rfoot.fs)
      {
if (q.lhip > q.rhip)
  {
    ls.state = 4;
    servosw.rknee = FF_MODE;
    servosw.lhip =  FF_MODE;
    adapt_control();
  }
else
  {
    ls.state = 8;
    q_d.rknee = q.rknee + ls.swing_hi;
```

```
    servosw.rhip = FF_MODE;
    servosw.rknee = FF_MODE;
  }
      }
    break;

/*  Put the right leg back on the ground  */
  case 11:

    ls.Fz_d = ls.M1*G+ls.k_bodyz*(q_d.z-ls.z_ter)
      +ls.b_bodyz*(qd_d.z - qd.z);

    compute_knee_ffs(SINGLE_DOWN,LEFT);

    if (gc_lfoot.fs){
      q_d.lhip  = MAX(MIN((q.lhip+MAX_H_TORQ/ls.k_lhip),THETA_L),
      (q.lhip+MIN_H_TORQ/ls.k_lhip));
    }

    if (gc_rfoot.fs)
      {
ls.state=4;
servosw.lhip = FF_MODE;
servosw.rknee = FF_MODE;
adapt_control();
      }
    break;

  default:

    ls.k_lhip  = HIP_K;
    ls.k_lknee = KNEE_K;
    ls.k_rhip  = HIP_K;
    ls.k_rknee = KNEE_K;

    q_d.lknee = -0.1;
    q_d.rknee = -0.1;
    q_d.lhip  = -0.13;
    q_d.rhip  = 0.13;
    break;
  }
}


/* ************************************************************
   adapt_control()
```

```
   Called to calculate terrain data and virtual leg data
   ********************************************************** */
void adapt_control()
{
  double lfootx,lfootz,rfootx,rfootz,factor;

  ls.ter_z = terrain_z_at_xy(q.x,0.0);

  ls.z_ter = q.z - ls.ter_z;

  ls.q_rvleg = THIGH_Z+SHANK_Z-q.rknee;

  ls.q_lvleg = THIGH_Z+SHANK_Z-q.lknee;

  ls.q_rfoota = THETA_R;

  ls.q_lfoota = THETA_L;

  lfootx = -(ls.q_lvleg)*sin(THETA_L);
  rfootx = -(ls.q_rvleg)*sin(THETA_R);
  lfootz =  (ls.q_lvleg)*cos(THETA_L);
  rfootz =  (ls.q_rvleg)*cos(THETA_L);

  if (gc_rfoot.fs){
    ls.r_mu=ABS((gc_rfoot.f_z*sin(ls.ground_angle)
 +gc_rfoot.f_x*cos(ls.ground_angle))/
(-gc_rfoot.f_z*cos(ls.ground_angle)
 +gc_rfoot.f_x*sin(ls.ground_angle)));
  }
  else ls.r_mu = 0.0;

  if (gc_lfoot.fs){
    ls.l_mu=ABS((gc_lfoot.f_z*sin(ls.ground_angle)
 +gc_lfoot.f_x*cos(ls.ground_angle))/
(-gc_lfoot.f_z*cos(ls.ground_angle)
 +gc_lfoot.f_x*sin(ls.ground_angle)));
  }
  else ls.l_mu = 0.0;
  if (gc_rfoot.fs && gc_lfoot.fs && ls.state>1)
    {
      if (ls.q_rfoota < ls.q_lfoota){
ls.ground_angle = atan2(rfootz-lfootz,lfootx-rfootx);}
      else{
ls.ground_angle = atan2(lfootz-rfootz,rfootx-lfootx);}
    }
  if (ls.ground_angle > 0.0)
```

```
    {
      factor = MAX(0,(MAX_TERRAIN_ANGLE
      - ls.ground_angle)/MAX_TERRAIN_ANGLE);
    }
}


/* ************************************************************
   set_damping_from_k_zeta()
   Called to take ls.k_((l/r)(knee/hip)/bodyz) and
   ls.zeta_(hip/knee/bodyz) and set b_(knee/hip/bodyz)
   to required values.
   Using CG's of thigh and shank at half extension for MoI,
   body plus thigh mass for bodyz controller, and body plus thigh
   mass for bodydx controller
   ************************************************************ */

void set_damping_from_k_zeta()
{
  double i=(M_lthigh*pow(THIGH_Z/2.0,2.0)
    + M_lshank*pow(THIGH_Z,2.0))/3.0;

  ls.b_lhip  = 2.0*ls.zeta_hip*sqrt(ls.k_lhip*i);
  ls.b_lknee = 2.0*ls.zeta_knee*sqrt(ls.k_lknee*M_lshank);
  ls.b_rhip  = 2.0*ls.zeta_hip*sqrt(ls.k_rhip*i);
  ls.b_rknee = 2.0*ls.zeta_knee*sqrt(ls.k_rknee*M_rshank);
  ls.b_bodyz = 2.0*ls.zeta_knee*sqrt(ls.k_bodyz*ls.M);

}


/* ************************************************************
   compute_knee_ffs(support,supside)
   Called to set ls.ff_rknee and ls.ff_lknee from ls.Fx_d
   and ls.Fz_d
   ************************************************************ */

void compute_knee_ffs(support, supside)
     int support;
     int supside;
{
  int i;
  double ff[2];
  double theta[2],hipdif[2];

  theta[LEFT] = THETA_L;
```

```
  theta[RIGHT] = THETA_R;

  hipdif[LEFT] = q.lhip-q.rhip;
  hipdif[RIGHT] = q.rhip-q.lhip;

  if (support == DOUBLE)
    {

      /* Calculate desired feed forward forces */

      ff[!supside] = (ls.Fz_d * sin(theta[supside])
      - ls.Fx_d * cos(theta[supside]))
/sin(hipdif[!supside]);

      ff[supside] = (ls.Fz_d * sin(theta[!supside])
     - ls.Fx_d * cos(theta[!supside]))
/sin(hipdif[supside]);

      /* Check to make sure the feet aren't being pulled up */
      for (i=0;i<2;i++)
{
  if (ff[i] > F_KNEE_MAX)
    {
      ff[i]  = F_KNEE_MAX;
      ff[!i] = (F_KNEE_MAX*cos(theta[!i])
-ls.Fz_d)/cos(theta[i]);
    }
}
    }
  else
    {
      ff[supside] = MIN((-ls.Fz_d/cos(theta[supside])),F_KNEE_MAX);

      if (support==SINGLE_DOWN) {
ff[!supside] = F_KNEE_DOWN;
      }
      else{
ff[!supside] = M_lshank*G*cos(theta[!supside]);
      }
    }
  ls.ff_lknee = MIN(MAX(ff[LEFT],F_KNEE_MIN),-F_KNEE_MIN);
  ls.ff_rknee = MIN(MAX(ff[RIGHT],F_KNEE_MIN),-F_KNEE_MIN);
}


/* ************************************************************
```

```
   onetime_init()
   Called to do one-time setup for control algorithm
   ************************************************************ */
void onetime_init()
{
  ls.t = 0.0;
  ls.M = M_body+M_lthigh+M_rthigh;
  ls.M1= ls.M + M_lshank;

  ls.kvdx = DX_KV;

  gc_rfoot.model = 101.0;
  gc_lfoot.model = 101.0;

  ls.theta_min  = THETA_MIN;
  ls.swing_lead = SWING_LEAD;
  ls.swing_hi   = SWING_HI;
  ls.swing_h    = SWING_H;
  ls.g_clearance = G_CLEARANCE;

  ls.k_lhip =  HIP_K;
  ls.k_lknee = KNEE_K;
  ls.k_rhip =  HIP_K;
  ls.k_rknee = KNEE_K;
  ls.k_bodyz = BODYZ_K;

  ls.zeta_knee = KNEE_ZETA;
  ls.zeta_hip  = HIP_ZETA;
  ls.zeta_bodyz= BODYZ_ZETA;
  ls.zeta_bodydx = BODYDX_ZETA;

  set_damping_from_k_zeta();

  ls.b_ff_b = B_KNEE_DOWN;

  ls.ground_angle = 0.0;

  gc_rfoot.k_z = 1000.0;
  gc_lfoot.k_z = 1000.0;
  gc_rfoot.k_x = gc_rfoot.k_z;
  gc_lfoot.k_x = gc_lfoot.k_z;
  gc_rfoot.k_y = 0.0;
  gc_lfoot.k_y = 0.0;

  gc_rfoot.b_z = 500.0;
  gc_lfoot.b_z = 500.0;
```

```
  gc_rfoot.b_x = 500.0;
  gc_lfoot.b_x = 500.0;
  gc_rfoot.b_y = 0.0;
  gc_lfoot.b_y = 0.0;

  ls.k_ff = FF_K;
  ls.b_ff = FF_B;

  q_d.z  = Z_NOM;
  ls.dx_min = DX_MIN;
  qd_d.x = DX_NOM;
  qd_d.z = 0.0;
}


/* ************************************************************
   init_control()
   Called to initialize or reset the control algorithm.
   ************************************************************ */
void init_control()
{
  q.z = Z_NOM;
  q.x = 0.0;
  qd.x = 0.0;
  qd.z = 0.0;
  ls.Fz_d = ls.M*G;
  ls.Fx_d = 0.0;

  q_d.lhip =  INIT_HIP;
  q_d.lknee = (Z_NOM)/cos(THETA_L) - THIGH_Z;
  q_d.rhip = -INIT_HIP;
  q_d.rknee = (Z_NOM)/cos(THETA_R) - THIGH_Z;

  q.lhip =  INIT_HIP;
  q.lknee = (Z_NOM)/cos(THETA_L) - THIGH_Z;
  q.rhip = -INIT_HIP;
  q.rknee = (Z_NOM)/cos(THETA_R) - THIGH_Z;

  ls.ff_lhip = 0.0;
  ls.ff_rhip = 0.0;

  compute_knee_ffs(DOUBLE,RIGHT);

  servosw.lhip = PD_FF_MODE;
  servosw.lknee = PD_FF_MODE;
  servosw.rhip = PD_FF_MODE;
```

```
  servosw.rknee = PD_FF_MODE;
}



/* ************************************************************
   user_gcontact()

     called with a contact point.  This is to allow user-defined
     ground contact - set the contact type to something over 100,
     like with servos.

     The parameters are the ground contact structure, the
     position of the ground contact in global coordinates, and
     the velocity of the ground contact in global coordinates.

   ************************************************************ */

void
user_gcontact(contact, x, y, z, dx, dy, dz)
     basic_gcontact *contact;
     double x, y, z, dx, dy, dz;
{

  double f[3], p[3], g[3], v[3], theta, dtheta;

  g[0] = z;
  g[1] = x;
  g[2] = y;

/*  sdpos(link, p, g);
  sdvel(link, p, v);  */

  /* Now you have g = position (z,x,y) of the contact point
     in the ground frame */
  contact->at_x = g[1];
  contact->at_y = g[2];
  contact->at_z = g[0];
  contact->at_th = 0.0;

  switch ((int)(contact->model))
    {
    case 101:
      linear_spring_damper_terrain(contact, x, y, z, dx, dy, dz);
      /* or DOGHNUT */
      break;
```

```
    default:
      contact->f_x = 0.0;
      contact->f_y = 0.0;
      contact->f_z = 0.0;
      break;
    }
}
```

C.2 KWALKB Control Code

Following are the functions I wrote to implement the KWALKB control system within the structure of the existing simulation package. The user_gcontact is part of Peter Dilworth's terrain module.

```
/* ************************************************************
   control1()
     Called every control cycle.  Should contain or call the actual
     creature control code.
   ************************************************************ */
void
control1()
{

  adapt_control();

  switch ((int)ls.state) {

/*     Initialize robot    */
  case 0:

    onetime_init();

    ls.state=1;

    break;

  case 1:

    init_control();

    ls.state=2;
```

```
    break;

  case 2:

/*  Wait here for transients to settle out  */

    solve_leg(LEFT);
    solve_leg(RIGHT);

    if(ls.t>0.3){
      ls.state = 3;
      ls.ff_lhip = 0.0;
      ls.ff_lknee = 0.0;
      ls.ff_rknee = 0.0;
      servosw.lknee = FF_MODE;
      servosw.rknee = FF_MODE;
      servosw.lhip = FF_MODE;
      servosw.rhip = PD_MODE;
    }
    break;

  case 3:

    /*  Command  knee torques to stay upright, keep zero torque
at left hip, and position control the right hip to keep
pitch at zero  */

    ls.Fz_d = ls.M*G+ls.k_bodyz*(q_d.z-ls.z_ter)
      +ls.b_bodyz*(qd_d.z - qd.z);

    ls.Fx_d = 0.0;

    if (gc_rfoot.fs){
      q_d.rhip  = THETA_R;
    }

    torques_from_forces(DOUBLE,RIGHT);

    if(ls.t>0.4)
      {
ls.state = 4;

/*  Soften position control for picking up and putting
    down legs  */
ls.k_lknee = KNEE_AIR_K;
ls.k_rknee = KNEE_AIR_K;
```

```
set_damping_from_k_zeta();
servosw.lhip = FF_MODE;
servosw.lknee = FF_MODE;
servosw.rknee = FF_MODE;
servosw.rhip = PD_MODE;
      }
    break;


/*     Double support, right leg forward   */
  case 4:

    ls.Fz_d = ls.M*G+ls.k_bodyz*(q_d.z-ls.z_ter)
      +ls.b_bodyz*(qd_d.z - qd.z);

    ls.Fx_d = ls.kvdx * (qd_d.x - qd.x);

    /*  Stabilize pitch with front leg;  */
    if (gc_rfoot.fs){
      q_d.rhip  = THETA_R;
    }

    torques_from_forces(DOUBLE,RIGHT);

    if ( ls.q_rfoota > 0.0 ||
(pow(ls.dx_min,2.0) <=
 pow(qd.x,2.0) - ls.z_ter*G*pow(tan(ls.q_rfoota),2.0)))
      {
ls.state = 5;
ls.foot_min=ls.q_lvleg - ls.g_clearance;
ls.q_d_lvleg = ls.q_lvleg - ls.swing_hi;
solve_leg(LEFT);
servosw.lknee = PD_MODE;
servosw.lhip  = PD_MODE;
      }
    break;

/*     Pick up the swing leg   */
  case 5:

    ls.Fz_d = ls.M1*G+ls.k_bodyz*(q_d.z-ls.z_ter)
      +ls.b_bodyz*(qd_d.z - qd.z);

/*  Now position control the right hip to keep the body level  */

    if (gc_rfoot.fs){
```

```
      q_d.rhip  = THETA_R;
    }

    if (ls.q_d_lvleg > ls.q_rvleg){
      ls.q_d_lvleg = ls.q_rvleg - ls.swing_h;
    }

    solve_leg(LEFT);

    torques_from_forces(SINGLE,RIGHT);

    if (((ls.q_lvleg < ls.q_rvleg && ls.q_lvleg < ls.foot_min) ||
 ls.q_lfoota < ls.q_rfoota) && !gc_lfoot.fs){
      ls.state = 6;
      ls.q_d_lfoota  = -ls.q_rfoota + ls.swing_lead;
      solve_leg(LEFT);
    }
    break;

/*     Move through single support on right leg  */
  case 6:

    ls.Fz_d = ls.M1*G+ls.k_bodyz*(q_d.z-ls.z_ter)
      +ls.b_bodyz*(qd_d.z - qd.z);

/*  Mirror global angle of support leg with swing leg */

    ls.q_d_lfoota  = MAX(-ls.q_rfoota+ls.swing_lead, 1.2*ls.theta_min);

/*  Make sure swing foot stays above support foot  */

    if (ls.q_d_lfoota < 0 || ls.q_d_lfoota > ls.q_rfoota){
      ls.q_d_lvleg = ls.q_rvleg - ls.swing_h;
    }

/*  Now position control the right hip to keep the body level  */

    if (gc_rfoot.fs){
      q_d.rhip  = THETA_R;
    }

    solve_leg(LEFT);

    torques_from_forces(SINGLE,RIGHT);

    if (ls.q_lfoota < ls.theta_min && ls.q_rfoota > ls.q_lfoota){
```

```
      ls.state = 7;
      servosw.lknee = FF_B_MODE;
      servosw.lhip  = FF_B_MODE;
    }

    if (gc_lfoot.fs){
      if (ls.q_rfoota > ls.q_lfoota){
ls.state = 8;
servosw.lknee = FF_MODE;
servosw.rhip =  FF_MODE;
      }
      else {
ls.state = 5;
ls.foot_min=ls.q_lvleg - ls.g_clearance;
      }
    }
    break;

/*  Put the left leg back on the ground  */
  case 7:

    ls.Fz_d = ls.M1*G+ls.k_bodyz*(q_d.z-ls.z_ter)
      +ls.b_bodyz*(qd_d.z - qd.z);

    if (gc_rfoot.fs){
      q_d.rhip  = THETA_R;
    }

    torques_from_forces(SINGLE,RIGHT);
    torques_from_forces(SINGLE_DOWN,RIGHT);

    if (gc_lfoot.fs){
      ls.state=8;
      servosw.lknee = FF_MODE;
      servosw.rhip =  FF_MODE;
      servosw.lhip =  PD_MODE;
    }
    break;


/*     Double support, left leg forward   */
  case 8:

    ls.Fz_d = ls.M*G+ls.k_bodyz*(q_d.z-ls.z_ter)
      +ls.b_bodyz*(qd_d.z - qd.z);
```

```
    ls.Fx_d = ls.kvdx * (qd_d.x - qd.x);

    torques_from_forces(DOUBLE,LEFT);

    if (gc_lfoot.fs){
      q_d.lhip  = THETA_L;
    }

    if (ls.q_lfoota > 0.0 ||
(pow(ls.dx_min,2.0) <=
 pow(qd.x,2.0) - ls.z_ter*G*pow(tan(ls.q_lfoota),2.0)))
      {
ls.state = 9;
ls.foot_min=ls.q_rvleg - ls.g_clearance;
ls.q_d_rvleg = ls.q_rvleg - ls.swing_hi;
solve_leg(RIGHT);
servosw.rknee = PD_MODE;
servosw.rhip  = PD_MODE;
      }
    break;

    /*     Pick up the swing leg (right leg)  */
  case 9:

    ls.Fz_d = ls.M1*G+ls.k_bodyz*(q_d.z-ls.z_ter)
      +ls.b_bodyz*(qd_d.z - qd.z);

    /*  Now position control the left hip to keep the body level  */

    if (gc_lfoot.fs){
      q_d.lhip  = THETA_L;
    }

    torques_from_forces(SINGLE,LEFT);

    if (ls.q_d_rvleg < ls.q_lvleg){
      ls.q_d_rvleg = ls.q_lvleg - ls.swing_h;
    }

    solve_leg(RIGHT);

    if (((ls.q_rvleg < ls.q_lvleg && ls.q_rvleg < ls.foot_min) ||
 (ls.q_rfoota<ls.q_lfoota)) && !gc_rfoot.fs)
      {
ls.state = 10;
ls.q_d_rfoota  = -ls.q_lfoota + ls.swing_lead;
```

```
solve_leg(RIGHT);
      }
    break;

/*     Move through single support on left leg  */
  case 10:

    ls.Fz_d = ls.M1*G+ls.k_bodyz*(q_d.z-ls.z_ter)
      +ls.b_bodyz*(qd_d.z - qd.z);

/*  Mirror global angle of support leg with swing leg */

    ls.q_d_rfoota  = MAX(-ls.q_lfoota+ls.swing_lead,1.2*ls.theta_min);

    if (ls.q_rfoota < 0 || ls.q_d_rfoota < ls.q_lfoota){
      ls.q_d_rvleg = ls.q_lvleg - ls.swing_h;
    }

/*  Now position control the left hip to keep the body level  */

    if (gc_lfoot.fs){
      q_d.lhip  = THETA_L;
    }

    solve_leg(RIGHT);

    torques_from_forces(SINGLE,LEFT);

    if (ls.q_lfoota > ls.q_rfoota && ls.q_rfoota < ls.theta_min)
      {
ls.state = 11;
servosw.rknee = FF_B_MODE;
servosw.rhip  = FF_B_MODE;
      }

    if (gc_rfoot.fs)
      {
if (ls.q_lfoota > ls.q_rlfoota)
  {
    ls.state = 4;
    servosw.rknee = FF_MODE;
    servosw.lhip =  FF_MODE;
  }
else
  {
    ls.state = 9;
```

```
    ls.q_d_rvleg = ls.q_rvleg - ls.swing_hi;
  }
      }
    break;

/*  Put the right leg back on the ground  */
  case 11:

    ls.Fz_d = ls.M1*G+ls.k_bodyz*(q_d.z-ls.z_ter)
      +ls.b_bodyz*(qd_d.z - qd.z);

    if (gc_lfoot.fs){
      q_d.lhip  = THETA_L;
    }

    torques_from_forces(SINGLE,LEFT);
    torques_from_forces(SINGLE_DOWN,LEFT);

    if (gc_rfoot.fs)
      {
ls.state=4;
servosw.lhip  = FF_MODE;
servosw.rknee = FF_MODE;
servosw.rhip  = PD_MODE;
      }
    break;

  default:

    ls.k_lhip  = HIP_K;
    ls.k_lknee = KNEE_K;
    ls.k_rhip  = HIP_K;
    ls.k_rknee = KNEE_K;

    q_d.lknee = -0.1;
    q_d.rknee = -0.1;
    q_d.lhip  = -0.13;
    q_d.rhip  = 0.13;
    break;
  }
}

/* ************************************************************
   set_damping_from_k_zeta()
   Called to take ls.k_((l/r)(knee/hip)/bodyz) and
   ls.zeta_(hip/knee/bodyz) and set b_(knee/hip/bodyz) to
```

```
   required values.
   Using CG's of thigh and shank at half extension for MoI,
   body plus thigh mass for bodyz controller, and body plus thigh
   mass for bodydx controller
   ********************************************************** */
void set_damping_from_k_zeta()
{
  double i=(M_lthigh*pow(THIGH_Z/2.0,2.0)
    + M_lshank*pow(THIGH_Z+SHANK_Z/2.0,2.0))/3.0;
  double i1 = M_lshank*pow(SHANK_Z/2.0,2.0)/3.0;

  ls.b_lhip  = 2.0*ls.zeta_hip*sqrt(ls.k_lhip*i);
  ls.b_lknee = 2.0*ls.zeta_knee*sqrt(ls.k_lknee*i1);
  ls.b_rhip  = 2.0*ls.zeta_hip*sqrt(ls.k_rhip*i);
  ls.b_rknee = 2.0*ls.zeta_knee*sqrt(ls.k_rknee*i1);
  ls.b_bodyz = 2.0*ls.zeta_knee*sqrt(ls.k_bodyz*ls.M);

}

/* ************************************************************
   adapt_control()
   Called to calculate data from terrain and virtual model
   ********************************************************** */
void adapt_control()
{
  double lfootx,lfootz,rfootx,rfootz,factor;

  ls.ter_z = terrain_z_at_xy(q.x,0.0);

  ls.z_ter = q.z - ls.ter_z;

/*  Calculate the virtual leg length and angle  */
  ls.q_lvleg = MIN(THIGH_Z+SHANK_Z,
   pow(THIGH_Z*THIGH_Z+SHANK_Z*SHANK_Z
       +2.0*THIGH_Z*SHANK_Z*cos(q.lknee),0.5));

  ls.q_rvleg = MIN(THIGH_Z+SHANK_Z,
   pow(THIGH_Z*THIGH_Z+SHANK_Z*SHANK_Z
       +2.0*THIGH_Z*SHANK_Z*cos(q.rknee),0.5));

  ls.q_lfoota = THETA_L -
    ls.invert*acos((ls.q_lvleg*ls.q_lvleg+THIGH_Z*THIGH_Z
    -SHANK_Z*SHANK_Z)/(2.0*THIGH_Z*ls.q_lvleg));

  ls.q_rfoota = THETA_R -
    ls.invert*acos((ls.q_rvleg*ls.q_rvleg+THIGH_Z*THIGH_Z
```

```
    -SHANK_Z*SHANK_Z)/(2.0*THIGH_Z*ls.q_rvleg));

  ls.theta_r = THETA_R;
  ls.theta_l = THETA_L;

  lfootx = -(ls.q_lvleg)*sin(THETA_L);
  rfootx = -(ls.q_rvleg)*sin(THETA_R);
  lfootz =  (ls.q_lvleg)*cos(THETA_L);
  rfootz =  (ls.q_rvleg)*cos(THETA_L);

  if (gc_rfoot.fs){
    ls.r_mu=ABS((gc_rfoot.f_z*sin(ls.ground_angle)
 +gc_rfoot.f_x*cos(ls.ground_angle))/
(-gc_rfoot.f_z*cos(ls.ground_angle)
 +gc_rfoot.f_x*sin(ls.ground_angle)));
  }
  else ls.r_mu = 0;

  if (gc_lfoot.fs){
    ls.l_mu=ABS((gc_lfoot.f_z*sin(ls.ground_angle)
 +gc_lfoot.f_x*cos(ls.ground_angle))/
(-gc_lfoot.f_z*cos(ls.ground_angle)
 +gc_lfoot.f_x*sin(ls.ground_angle)));
  }
  else ls.l_mu = 0;

  if (gc_rfoot.fs && gc_lfoot.fs && ls.state>1)
    {
      if (ls.q_rfoota < ls.q_lfoota){
ls.ground_angle = atan2(rfootz-lfootz,lfootx-rfootx);}
      else{
ls.ground_angle = atan2(lfootz-rfootz,rfootx-lfootx);}
    }
}


/* ************************************************************
   torques_from_forces() takes the support type, support leg,
   and calculates the required knee torques
   ************************************************************ */
void torques_from_forces(support,supside)
     char support;
     char supside;
{

  int i;
```

```
  double theta[2][3];
  double alpha[2],footdif[2],ff[2];
  double q_vleg[2];

  theta[RIGHT][KNEE] = q.rknee;
  theta[LEFT][KNEE]  = q.lknee;
  theta[RIGHT][HIP]  = THETA_R;
  theta[LEFT][HIP]   = THETA_L;
  theta[RIGHT][FOOT]  = ls.q_rfoota;
  theta[LEFT][FOOT]   = ls.q_lfoota;

  q_vleg[RIGHT]=ls.q_rvleg;
  q_vleg[LEFT]=ls.q_lvleg;

  footdif[LEFT] = ls.q_lfoota-ls.q_rfoota;
  footdif[RIGHT] = ls.q_rfoota-ls.q_lfoota;

  alpha[LEFT] =  2*asin(ls.q_lvleg/(THIGH_Z+SHANK_Z));
  alpha[RIGHT] = 2*asin(ls.q_rvleg/(THIGH_Z+SHANK_Z));

  /*  Calculate feed forward forces for virtual leg  */

  if (support==DOUBLE)
    {
      /* Calculate desired feed forward forces */

      ff[!supside] = (ls.Fz_d * sin(theta[supside][FOOT])
      - ls.Fx_d * cos(theta[supside][FOOT]))
/sin(footdif[!supside]);

      ff[supside] = (ls.Fz_d * sin(theta[!supside][FOOT])
     - ls.Fx_d * cos(theta[!supside][FOOT]))
/sin(footdif[supside]);

      /* Check to make sure the feet aren't being pulled up */
      for (i=0;i<2;i++)
{
  if (ff[i] > F_KNEE_MAX)
    {
      ff[i]  = F_KNEE_MAX;
      ff[!i] = (F_KNEE_MAX*cos(theta[!i][FOOT])-ls.Fz_d)
/cos(theta[i][FOOT]);
    }
}
    }
  else
```

```
    {
      ff[supside] = MIN((-ls.Fz_d/cos(theta[supside][FOOT]))
,F_KNEE_MAX);
      if (support==SINGLE_DOWN) {
ff[!supside] = F_KNEE_DOWN;
      }
      else{
ff[!supside] = M_lshank*G*cos(theta[!supside][FOOT]);
      }
    }

  /* Now convert to knee torques, keeping in the actuator limits */

  ls.ff_lknee = MAX(ls.min_k_torq,
    MIN(ls.max_k_torq,
-ls.invert*ff[LEFT]*(THIGH_Z+SHANK_Z)
*cos(alpha[LEFT]/2.0)/2.0));

  ls.ff_rknee = MAX(ls.min_k_torq,
    MIN(ls.max_k_torq,
-ls.invert*ff[RIGHT]*(THIGH_Z+SHANK_Z)
*cos(alpha[RIGHT]/2.0)/2.0));
  if (supside==RIGHT){
    ls.ff_lhip = 0.0;
  }
  else{
    ls.ff_rhip = 0.0;
  }
}

/* ************************************************************
   solve_leg(side)
   sets q.knee and q.hip for the desired leg to put the foot in place
   ************************************************************ */
void solve_leg(side)
     char side;
{
  double calpha, salpha;  /* cos of angle between knee and thigh */
  double cbeta;   /* cos of angle between virtual leg and thigh */
  double knee,hip,vleg2,foota,footd;

  if (side==LEFT)
    {
      foota = ls.q_d_lfoota;
      footd = ls.q_d_lvleg;
    }
```

```
  else
    {
      foota = ls.q_d_rfoota;
      footd = ls.q_d_rvleg;
    }

  vleg2 = footd*footd;

  cbeta = -(SHANK_Z*SHANK_Z-vleg2-THIGH_Z*THIGH_Z)
    /(2*THIGH_Z*footd);
  if (ABS(cbeta) > 1 ) cbeta = SIGN(cbeta);

  hip =  ls.invert*acos(cbeta) + foota;

  calpha = -(vleg2-THIGH_Z*THIGH_Z-SHANK_Z*SHANK_Z)
    /( 2*THIGH_Z*SHANK_Z);
  if (ABS(calpha)>1.0) calpha = SIGN(calpha);

  salpha = -ls.invert*pow((1-pow(calpha,2.0)),0.5);

  knee = (atan2(salpha,-calpha));


  /* Now command the new positions, or the closest you
     can without going over the torq limits  */
  if (side==LEFT)
    {
      if (ls.state > 1.0)
{
  q_d.lhip = MAX( MIN((q.lhip+ls.max_h_torq/ls.k_lhip),hip),
 (q.lhip+ls.min_h_torq/ls.k_lhip));
  q_d.lknee = MAX(MIN((q.lknee+ls.max_k_torq/ls.k_lknee)
      ,knee),
  (q.lknee+ls.min_k_torq/ls.k_lknee));
}
      else
{
  q_d.lhip = hip;
  q_d.lknee = knee;
}
    }
  else
    {
      if (ls.state > 1.0)
{
  q_d.rhip = MAX(MIN((q.rhip+MAX_H_TORQ/ls.k_rhip),hip),
```

```
 (q.rhip+MIN_H_TORQ/ls.k_rhip));
  q_d.rknee = MAX(MIN((q.rknee+MAX_K_TORQ/ls.k_rknee),knee),
  (q.rknee+MIN_K_TORQ/ls.k_rknee));
}
      else
{
  q_d.rhip = hip;
  q_d.rknee = knee;
}
    }
}


/* ************************************************************
   onetime_init()
   Called to initialize parameters for first run.
   ************************************************************ */
void onetime_init()
{
  ls.M = M_body;
  ls.M1= ls.M + M_lshank + M_lthigh;

  ls.kvdx = DX_KV;

  gc_rfoot.model = 101.0;
  gc_lfoot.model = 101.0;

  ls.theta_min  = THETA_MIN;
  ls.swing_lead = SWING_LEAD;
  ls.swing_hi   = SWING_HI;
  ls.swing_h    = SWING_H;
  ls.g_clearance = G_CLEARANCE;

  ls.invert = INVERT; /* Knee inversion -1=forward, 1 backward */

  ls.k_lhip =  HIP_K;
  ls.k_lknee = KNEE_K;
  ls.k_rhip =  HIP_K;
  ls.k_rknee = KNEE_K;
  ls.k_bodyz = BODYZ_K;

  ls.max_k_torq = MAX_K_TORQ;
  ls.min_k_torq = MIN_K_TORQ;
  ls.max_h_torq = MAX_H_TORQ;
  ls.min_h_torq = MIN_H_TORQ;
```

```
  ls.zeta_knee = KNEE_ZETA;
  ls.zeta_hip  = HIP_ZETA;
  ls.zeta_bodyz= BODYZ_ZETA;
  ls.zeta_bodydx = BODYDX_ZETA;

  set_damping_from_k_zeta();

  ls.b_ff_b = B_KNEE_DOWN;

  gc_rfoot.k_z = 1000.0;
  gc_lfoot.k_z = 1000.0;
  gc_rfoot.k_x = gc_rfoot.k_z;
  gc_lfoot.k_x = gc_lfoot.k_z;
  gc_rfoot.k_y = 0.0;
  gc_lfoot.k_y = 0.0;

  gc_rfoot.b_z = 500.0;
  gc_lfoot.b_z = 500.0;
  gc_rfoot.b_x = 500.0;
  gc_lfoot.b_x = 500.0;
  gc_rfoot.b_y = 0.0;
  gc_lfoot.b_y = 0.0;

  ls.k_ff = FF_K;
  ls.b_ff = FF_B;
}

/* ************************************************************
   init_control()
   Called to initialize the control algorithm.
   ************************************************************ */
void init_control()
{
  ls.t = 0.0;

  ls.Fz_d = ls.M*G;
  ls.Fx_d = 0.0;

  ls.q_d_rfoota=-INIT_FOOTA;
  ls.q_d_rvleg=INIT_FOOTD;
  ls.q_d_lfoota=INIT_FOOTA;
  ls.q_d_lvleg=INIT_FOOTD;

  solve_leg(LEFT);
  solve_leg(RIGHT);
```

```
  q_d.z = INIT_FOOTD*cos(INIT_FOOTA);

  q.lhip  =  q_d.lhip;
  q.lknee =  q_d.lknee;
  q.rhip  =  q_d.rhip;
  q.rknee =  q_d.rknee;
  q.z     =  q_d.z;

  ls.dx_min = DX_MIN;
  qd_d.x = DX_NOM;
  qd_d.z = 0.0;

  ls.ff_lhip = 0.0;
  ls.ff_lknee = 0.0;

  ls.ff_rhip = 0.0;
  ls.ff_rknee = 0.0;

  servosw.lhip = PD_FF_MODE;
  servosw.lknee = PD_FF_MODE;
  servosw.rhip = PD_FF_MODE;
  servosw.rknee = PD_FF_MODE;
}

/* ************************************************************
   user_gcontact()
   called with a contact point.  This is to allow user-defined
   ground contact - set the contact type to something over 100,
   like with servos.

   The parameters are the ground contact structure, the position
   of the ground contact in global coordinates, and the velocity
   of the ground contact in global coordinates.
   ************************************************************ */

void user_gcontact(contact, x, y, z, dx, dy, dz)
     basic_gcontact *contact;
     double x, y, z, dx, dy, dz;
{

  double f[3], p[3], g[3], v[3], theta, dtheta;

  g[0] = z;
  g[1] = x;
  g[2] = y;
```

```
  /* Now you have g = position (z,x,y) of the contact point
     in the ground frame */
  contact->at_x = g[1];
  contact->at_y = g[2];
  contact->at_z = g[0];
  contact->at_th = 0.0;

  switch ((int)(contact->model))
    {
    case 101:
      linear_spring_damper_terrain(contact, x, y, z, dx, dy, dz);
      /* or DOGHNUT */
      break;

    default:
      contact->f_x = 0.0;
      contact->f_y = 0.0;
      contact->f_z = 0.0;
      break;
    }
}
```

Appendix D

Simulation Code

The simulation system in the M.I.T. Leg Lab automatically produces the file main.c. As this is unpublished proprietary information, I will list only Peter Dilworth's modifications made to add his rough terrain module.

D.1 Modifications to Main.c

```
/* BEGIN USER CODE 1 */
/* To read in a terrain file */
#define TERRAIN (LAST_SIM_COMMAND + 1)

terrain_data td;  /* PCD */

/* END USER CODE 1 */


/* BEGIN USER CODE 2 */
"TER",     /* 14 */
/* END USER CODE 2 */


/* BEGIN USER CODE 3 */
  goalfile_initialize();
/* END USER CODE 3 */


/* BEGIN USER CODE 4 */
    } else {
      block = 1;
      Display (vars); /* library */
```

```
    }
/* END USER CODE 4 */


/* BEGIN USER CODE 5 */
/* END USER CODE 5 */


/* BEGIN USER CODE 6 */
    case TERRAIN:
      if (nwords == 1)
{
  Err_Display("load what terrain file?");
}
      else
{
  if (load_terrain(words[1]))
    {
      Err_Display("Terrain file loaded");
    }
}
      break;

/* END USER CODE 6 */

/* BEGIN USER CODE 7 */

/*  Code from here to end is compliments of Peter Dilworth,
    staff of the MIT Leg Lab since June 1994.  */

/*
  this returns:
      0 -- if the foot did not hit the ground,
      1 -- if the foot landed on terrain,
     -1 -- if the foot landed outside terrain


  there are two modes:
      td.mapmode = REGIONMODE --> foot may land outside terrain
      (z = 0 in all cases)

      td.mapmode = DOGHNUTMODE --> foot always lands inside terrain,
      edges wrap around
*/
double check_for_terrain_hit( x, y, z )
     double x;
```

```
    double y;
    double z;
{
  double offsetx, offsety;
  double xslope, yslope;
  double deltay, deltax, deltaz;
  double Ox, Oy, Xx, Xy, Yx, Yy;
  int result = 0.0;
  int which;

  /* if region mode, see if leg is over terrain patch at all */
  if((td.mapmode == REGIONMODE) &&
     ((x < (td.xoffset*td.xres)) ||
      (x >= ((td.xoffset+td.xdim)*td.xres)) ||
      (y < (td.yoffset*td.yres)) ||
      (y >= ((td.yoffset+(td.ydim-1.0))*td.yres))))
    {
      /* foot is over flat default ground in region mode,
 flat ground z = zero everywhere */
      if(z < 0.0)
        {
          result = -1.0;
        }
    }
  else
    {
      /* either REGION or DOGHNUT mode (it doesn't matter which)
 , but foot is over terrain map */
      /* if in REGION mode, convert absolute foot coords into
 relative offset coords for use with terrain array check */
      if(td.mapmode == REGIONMODE)
        {
          /* convert point from linear world coords to linear
     terrain coords */
          offsetx = (x - (td.xoffset*td.xres));
          offsety = (y - (td.yoffset*td.yres));

          which = get_terrain_corner_points
    (offsetx, offsety, &Ox, &Oy, &Xx, &Xy, &Yx, &Yy);
        }
      else
        {
          /* td.mapmode == DOGHNUTMODE */
          /* wrap hit point around terrain using modulus function
     and resolve into linear terrain coords */
          offsetx = fmod(x, (td.xdim*td.xres));
```

```
        offsety = fmod(y, ((td.ydim-1.0)*td.yres));

        which = get_terrain_corner_points
    (offsetx, offsety, &Ox, &Oy, &Xx, &Xy, &Yx, &Yy);
      }

     td.Oz = td.terrain_array[((int)Oy*(int)td.xdim)+(int)Ox];
     td.Xz = td.terrain_array[((int)Xy*(int)td.xdim)+(int)Xx];
     td.Yz = td.terrain_array[((int)Yy*(int)td.xdim)+(int)Yx];

     /* get z values for corners of (possible) hit triangle */
     /* see if foot is actually touching ground */
     xslope = ((td.Xz-td.Oz)*td.zres)/td.xres;
     yslope = ((td.Yz-td.Oz)*td.zres)/td.yres;

     if(which == LOWER)
       {
         /* origin point of triangle is corner of grid square
    closest to world origin */
         deltax = fmod(offsetx, td.xres);
         deltay = fmod(offsety, td.yres);
       }
     else /* UPPER triangle */
       {
         /* origin point of triangle is corner of grid square on
    opposite side from world origin, hence flip */
         deltax = td.xres - fmod(offsetx, td.xres);
         deltay = td.yres - fmod(offsety, td.yres);
       }

     deltaz = (xslope*deltax) + (yslope*deltay);
     /* z of point right below foot on terrain */
     td.Tz = (td.Oz*td.zres) + deltaz;

     /* if it is, return 1.0 */
     if(z < td.Tz)
       {
         result = 1.0;
       }
   }

  return(result);
}

/*
  like check_for_terrain_hit, but it just returns z height
```

```
  of terrain below x,y
*/
double terrain_z_at_xy( x , y )
     double x;
     double y;
{
  double offsetx, offsety;
  double xslope, yslope;
  double deltay, deltax, deltaz;
  double Ox, Oy, Xx, Xy, Yx, Yy;
  double tdXz, tdOz, tdYz;
  double result = 0.0;
  int which;

  /* if region mode, see if leg is over terrain patch at all */
  if((td.mapmode == REGIONMODE) &&
     ((x < (td.xoffset*td.xres)) ||
      (x >= ((td.xoffset+td.xdim)*td.xres)) ||
      (y < (td.yoffset*td.yres)) ||
      (y >= ((td.yoffset+(td.ydim-1.0))*td.yres))))
    {
      result = 0.0;
    }
  else
    {
      /* either REGION or DOGHNUT mode (it doesn't matter which),
 but foot is over terrain map */

      /* if in REGION mode, convert absolute foot coords into
 relative offset coords for use with terrain array check */
      if(td.mapmode == REGIONMODE)
        {
          /* convert point from linear world coords to
     linear terrain coords */
          offsetx = (x - (td.xoffset*td.xres));
          offsety = (y - (td.yoffset*td.yres));

          which = get_terrain_corner_points
    (offsetx, offsety, &Ox, &Oy, &Xx, &Xy, &Yx, &Yy);
        }
      else
        {
          /* td.mapmode == DOGHNUTMODE */
          /* wrap hit point around terrain using modulus function
     and resolve into linear terrain coords */
          offsetx = fmod(x, (td.xdim*td.xres));
```

```
        offsety = fmod(y, ((td.ydim-1.0)*td.yres));

        which = get_terrain_corner_points
    (offsetx, offsety, &Ox, &Oy, &Xx, &Xy, &Yx, &Yy);
      }

     tdOz = td.terrain_array[((int)Oy*(int)td.xdim)+(int)Ox];
     tdXz = td.terrain_array[((int)Xy*(int)td.xdim)+(int)Xx];
     tdYz = td.terrain_array[((int)Yy*(int)td.xdim)+(int)Yx];

     /* get z values for corners of (possible) hit triangle */
     /* see if foot is actually touching ground */
     xslope = ((tdXz-tdOz)*td.zres)/td.xres;
     yslope = ((tdYz-tdOz)*td.zres)/td.yres;

     if(which == LOWER)
       {
         /* origin point of triangle is corner of grid square
    closest to world origin */
         deltax = fmod(offsetx, td.xres);
         deltay = fmod(offsety, td.yres);
       }
     else /* UPPER triangle */
       {
         /* origin point of triangle is corner of grid square on
    opposite side from world origin, hence flip */
         deltax = td.xres - fmod(offsetx, td.xres);
         deltay = td.yres - fmod(offsety, td.yres);
       }

     deltaz = (xslope*deltax) + (yslope*deltay);

     /* z of point right below foot on terrain */
     result = (tdOz*td.zres) + deltaz;
   }

  return(result);
}

/*
  determines the ordinates of the three corner points that
  define the triangle of the terrain array that contain the
  possible foot landing point Note: point defined by x, y
  should be GUARANTEED to be in range
  of terrain array, if it is not, something is terribly, terribly
  wrong...
```

```
  returns: UPPER or LOWER to specify which triangle point
  is inside of
*/

int get_terrain_corner_points( x, y, Ox, Oy, Xx, Xy, Yx, Yy )
     double x;
     double y;
     double *Ox;
     double *Oy;
     double *Xx;
     double *Xy;
     double *Yx;
     double *Yy;
{
  double Sx, Sy;
  int i1, i2;
  int result;
  double lower_dist, upper_dist;

  /* first find out which grid square you are in */
  /* find corner of this grid closest to world origin */
  i1 = (int)(x/td.xres);
  i2 = (int)(y/td.yres);
  Sx = (double)i1;
  Sy = (double)i2;

  /* then find out which of the two possible origin corners is
     closest to point, this defines UPPER vrs LOWER corner */
  lower_dist = dist((Sx*td.xres), (Sy*td.yres), x, y);
  upper_dist = dist(((Sx+1.0)*td.xres), ((Sy+1.0)*td.yres), x, y);
  if(lower_dist < upper_dist)
    result = LOWER;
  else
    result = UPPER;



  /* fill in three corner points of triangle */
  if(result == LOWER)
    {
      *Ox = Sx;
      *Oy = Sy;
      *Xx = Sx+1.0;
      *Xy = Sy;
      *Yx = Sx;
```

```
      *Yy = Sy+1.0;
    }
  else
    {
      *Ox = Sx+1.0;
      *Oy = Sy+1.0;
      *Xx = Sx;
      *Xy = Sy+1.0;
      *Yx = Sx+1.0;
      *Yy = Sy;
    }

  return(result);
}

/*
  just finds the distance between two points
*/
double dist( x1, y1, x2, y2 )
     double x1;
     double y1;
     double x2;
     double y2;
{
  double result;

  result = sqrt(((x1-x2)*(x1-x2))+((y1-y2)*(y1-y2)));

  return(result);
}


/*
  first pass at custom terrain, implements normal linear
  spring damper gc code, but uses terrain loaded into terrain
  structure instead of flat ground.

  there are two modes:
         td.mapmode = REGIONMODE, where terrain maps onto
 a patch whose origin is located at #xoffset #yoffset

 td.mapmode = DOGHNUTMODE, where terrain wraps around
 at the edges like a toroid
*/
void linear_spring_damper_terrain(gc, x, y, z, dx, dy, dz)
  basic_gcontact *gc;
```

```
  double x, y, z, dx, dy, dz;
{
  double new_fs;

  /* get current state of foot switch */
  new_fs = check_for_terrain_hit(x, y, z);

  if(new_fs == 0.0)
    {
      /* foot is off ground do nothing */
      gc->fs = 0.0;
      gc->f_x = 0.0;
      gc->f_y = 0.0;
      gc->f_z = 0.0;
      gc->t_th = 0.0;
    }
  else
    {
      /* initial touchdown coordinate record */
      if(gc->fs == 0.0)
        {
          gc->td_x = x;
          gc->td_y = y;
          gc->td_z = z;
        }

      /* record that foot has touched down */
      gc->fs = new_fs;

      /* Foot switch is on, need to apply forces. */
      if((td.mapmode == REGIONMODE) && (gc->fs == -1.0))
        {
          /* foot landed outside of terrain region, use
     standard ground contact model */
          /* x direction */
          gc->f_x = (gc->td_x - x);
          gc->f_x = (gc->k_x)*gc->f_x - gc->b_x*dx;



          /* y direction */
          gc->f_y = (gc->td_y - y);
          gc->f_y = (gc->k_y)*gc->f_y - gc->b_y*dy;

          /* z direction */
          if(gc->nomlen_z + z > 0.002)
```

```
            gc->f_z = -(gc->k_z)*z/(gc->nomlen_z + z) - gc->b_z*dz;
          else
            gc->f_z = -(gc->k_z)*z/(0.002) - gc->b_z*dz;

          /* Zero z direction if it would result in sticky ground. */
          if(gc->f_z < 0.0)
            gc->f_z = 0.0;

          gc->t_th = 0.0;
        }
       else
        {
          /* foot landed inside terrain region, use terrain
      ground contact model */
          /* test forces, always vertical direction */
          /* x direction */
          gc->f_x = (gc->td_x - x);
          gc->f_x = (gc->k_x)*gc->f_x - gc->b_x*dx;

          /* y direction */
          gc->f_y = (gc->td_y - y);
          gc->f_y = (gc->k_y)*gc->f_y - gc->b_y*dy;

          /* z direction */
          if(gc->nomlen_z + (td.Tz-z) > 0.002)
            gc->f_z = (gc->k_z)*(td.Tz-z)
       /(gc->nomlen_z + (td.Tz-z)) - gc->b_z*dz;
          else
            gc->f_z = (gc->k_z)*(td.Tz-z)/(0.002) - gc->b_z*dz;

          /* Zero z force if it would result in sticky ground. */
          if(gc->f_z < 0.0)
            gc->f_z = 0.0;

          gc->t_th = 0.0;
        }
    }
}

/*
  requests terrain file to load, and loads it into memory
*/
int load_terrain(tfilename)
     char *tfilename;
{
  FILE *fp;
```

```
double keyword, value, orientmode, x, y, num;
double xmax, ymax;
int result;
int error;

if((fp = fopen(tfilename, "r+")) == NULL)
  {
    error = 1;
  }
else
  {
    /* set default values for td eles */
    td.xres = -1.0;
    td.yres = -1.0;
    td.zres = -1.0;
    td.xdim = -1.0;
    td.ydim = -1.0;
    orientmode = XDOWN;
    td.mapmode = REGIONMODE;
    td.xoffset = 0.0;
    td.yoffset = 0.0;
    x = y = 0.0;

    /* loop thru and get keywords up to #beginterrain */
    keyword = get_keyword(fp, &value);
    while((keyword != BEGINTERRAIN) && (keyword != BADKEY))
      {
        if(keyword == XRESOLUTION)
          td.xres = value;

        if(keyword == YRESOLUTION)
          td.yres = value;

        if(keyword == ZRESOLUTION)
          td.zres = value;

        if(keyword == DOWNDIMENSION)
          td.xdim = value;

        if(keyword == ACROSSDIMENSION)
          td.ydim = value;

        if(keyword == XDOWN)
          orientmode = XDOWN;

        if(keyword == YDOWN)
```

```
      orientmode = YDOWN;

    if(keyword == REGIONMODE)
      td.mapmode = REGIONMODE;

    if(keyword == DOGHNUTMODE)
      td.mapmode = DOGHNUTMODE;

    if(keyword == XOFFSET)
      td.xoffset = value;

    if(keyword == YOFFSET)
      td.yoffset = value;

    keyword = get_keyword(fp, &value);
  }

xmax = ymax = 0.0;
error = 0;

if((td.xres == 0.0) || (td.yres == 0.0))
  {
    Err_Display("cannot have zero for axis resolution value");
    error = 1;
  }
else if((keyword == BADKEY) ||
   (td.xres == -1.0) ||
   (td.yres == -1.0) ||
   (td.zres == -1.0) ||
   (td.xdim == -1.0) ||
   (td.ydim == -1.0))
  {
    Err_Display("bad or missing keyword in terrain file");
    error = 1;
  }
else if(keyword == BEGINTERRAIN)
  {
    /* ready to load terrain data into mem */
    /* free array if there is already one loaded */
    if(td.tloaded == TLOADED)
      free(td.terrain_array);
    /* allocate and fill in the terrain array */
    if((td.terrain_array =
(double *)malloc( ((int)(td.ydim)+1)*
 ((int)(td.xdim)+1)*
 sizeof(double) )) == NULL)
```

```
        {
          Err_Display("Not enough memory for terrain array");
          error = 1;
        }
      else
        {
          /* loop thru and collect grid data points */
          while(((num = fgetdigit(fp, orientmode, &x, &y))
  != -1.0) && (error == 0))
            {
              if((x > td.xdim) || (y > td.ydim))
                {
                  Err_Display("overran terrain array size",
"in terrain date file");
                  error = 1;
                }
              else
                {
                  if(x > xmax) xmax = x;
                  if(y > ymax) ymax = y;
                  td.terrain_array[((int)y*(int)td.xdim)+
    (int)x] = num;
                }
            }

          /*
          debug_printf("xmax is", xmax);
          debug_printf("equals:", (td.xdim-1.0));
          debug_printf("ymax is", ymax);
          debug_printf("equals:", (td.ydim-1.0));
          */

          if((xmax != (td.xdim-1.0)) || (ymax != td.ydim))
            {
              Err_Display("wrong number of data points to",
   "fill specified terrain array");
              error = 1;
            }
        }
    }

  if(!error)
    td.tloaded = TLOADED;

  fclose(fp);
}
```

```
  return(!error);
}
double get_keyword( fp, value )
     FILE *fp;
     double *value;
{
  char line[1000];
  char keyword[100];
  double result = BADKEY;

  fgets(line, 90, fp);

  sscanf(line, "#%s %lf\n", keyword, value);

  if(!stricmp(keyword, "beginterrain"))
    result = BEGINTERRAIN;

  if(!stricmp(keyword, "endterrain"))
    result = ENDTERRAIN;

  if(!stricmp(keyword, "xresolution"))
    result = XRESOLUTION;

  if(!stricmp(keyword, "yresolution"))
    result = YRESOLUTION;

  if(!stricmp(keyword, "zresolution"))
    result = ZRESOLUTION;

  if(!stricmp(keyword, "downdimension"))
    result = DOWNDIMENSION;

  if(!stricmp(keyword, "acrossdimension"))
    result = ACROSSDIMENSION;

  if(!stricmp(keyword, "xdown"))
    result = XDOWN;

  if(!stricmp(keyword, "ydown"))
    result = YDOWN;

  if(!stricmp(keyword, "regionmode"))
    result = REGIONMODE;

  if(!stricmp(keyword, "doghnutmode"))
```

```
    result = DOGHNUTMODE;

  if(!stricmp(keyword, "xoffset"))
    result = XOFFSET;

  if(!stricmp(keyword, "yoffset"))
    result = YOFFSET;

  return(result);
}

double fgetdigit( fp, mode, x, y )
     FILE *fp;
     double mode;
     double *x;
     double *y;
{
  char num[100], c;
  int newline = 0;
  double result;
  int i;
  char fred[10];

  /* check previously saved delim to see if it was a newline */
  if(td.fputc_hack == 10.0)
    {
      newline = 1;
    }

  /* first eat leading white space */
  c = fgetc(fp);
  while((c == 0x0a) || (c == 0x20) || (c == ' ') || (c == '\t'))
    {
      if(c == 0x0a)
        {
          newline = 1;
        }

      c = fgetc(fp);
    }


  if(c == EOF)
    {
      result = -1.0;
    }
```

```
 else
   {
     i = 0;

     /* get the next series of digits */
     while((c != EOF) && (c != '\n') && (c != ' ') && (c != '\t'))
       {
         num[i++] = c;
         c = fgetc(fp);
       }
     num[i] = 0x00;

     /* save right hand delim in case it was a newline
(so you can sense it next time around) */
     td.fputc_hack = (double)c;
   }

 /* figure out x and y */
 if(mode == XDOWN)
   {
     /* each new number up to newline increments y, and newline
increments x */      if(newline)
       {
         ++(*x);
         (*y) = 0.0;
       }
     else
       {
         ++(*y);
       }
   }
 else if(mode == YDOWN)
   {
     /* each new number up to newline increments x, and a newline
increments y */
     if(newline)
       {
         ++(*y);
         (*x) = 0.0;
       }
     else
       {
         ++(*x);
       }
   }
 else
```

```
    {
      result = -1.0;
    }

  /* sense if num was end data flag */
  if((num[0] == '#') || (num[1] == '#') || (num[2] == '#'))
    {
      result = -1.0;
    }
  else
    {
      /* num has the digit, and newline tells if there was a
 newline encountered before num */
      result = (double)atoi(num);
    }

  return(result);
}

void debug_printf( str, num )
     char *str;
     double num;
{
  FILE *fp;

  fp = fopen("stuff", "a");
  fprintf(fp, "%s %lf\n", str, num);
  fclose(fp);
}

/* END USER CODE 7 */
```

Appendix E

Terrain Files

E.1 Smooth Terrain and Hills

The terrain files consist of a number of dimensioning and mode commands followed by a matrix of ground heights. Smooth terrain is just a list of zeros for level ground or a list of numbers which ascend in the X direction for upward slopes. For downward slopes, I used a mound. This went up from ground level, sloped up to a plateau, and then turned into a downhill slope. This way the robot had to walk up the slope before going down, but the state at the top could be saved. This way additional runs could be done down the slope with different parameter values.

The rough terrain file following this section is representative of the file format.

E.2 Rough Terrain

This is the terrain file I generated to test rough terrain performance. It is written in the format Peter Dilworth defined as input for the terrain model.

The very rough terrain was a copy of this file with the zresolution increased to 0.02.

```
#xresolution 0.10
#yresolution 0.10
#zresolution 0.01
#downdimension 56
#acrossdimension 5
#xdown
#regionmode
#xoffset -3.0
#yoffset -2.0
#beginterrain
00 00 00 00 00
```

00 00 00 00 00
00 00 00 00 00
00 00 00 00 00
00 00 00 00 00
01 01 01 01 01
02 02 02 02 02
03 03 03 03 03
01 01 01 01 01
02 02 02 02 02
01 01 01 01 01
03 03 03 03 03
02 02 02 02 02
00 00 00 00 00
03 03 03 03 03
03 03 03 03 03
01 01 01 01 01
04 04 04 04 04
03 03 03 03 03
03 03 03 03 03
00 00 00 00 00
04 04 04 04 04
01 01 01 01 01
03 03 03 03 03
00 00 00 00 00
03 03 03 03 03
03 03 03 03 03
00 00 00 00 00
03 03 03 03 03
02 02 02 02 02
00 00 00 00 00
02 02 02 02 02
03 03 03 03 03
00 00 00 00 00
01 01 01 01 01
02 02 02 02 02
01 01 01 01 01
03 03 03 03 03
03 03 03 03 03
03 03 03 03 03
03 03 03 03 03
03 03 03 03 03
01 01 01 01 01
03 03 03 03 03
00 00 00 00 00
00 00 00 00 00
02 02 02 02 02

```
04 04 04 04 04
01 01 01 01 01
02 02 02 02 02
01 01 01 01 01
04 04 04 04 04
00 00 00 00 00
02 02 02 02 02
02 02 02 02 02
00 00 00 00 00
###
```

Bibliography

[1] S. Hirose and Y. Umetani. The Basic Motion Regulation System for a Quadruped Walking Vehicle. Contributed for presentation at the Design Engineering Technical Conference, Beverly Hills, CA, September 1980.

[2] Jessica K. Hodgins and Marc H. Raibert. Adjusting Step Length for Rough Terrain Locomotion. *IEEE Transactions on Robotics and Automation*, 7(3):289–298, June 1991.

[3] Shuuji Kajita, Tomio Yamaura, and Akira Kobayashi. Dynamic Walking Control of a Biped Robot Along a Potential Energy Conserving Orbit. *IEEE Transactions on Robotics and Automation*, 8(4):431–438, August 1992.

[4] Robert B. McGhee and Geoffery I. Iswandhi. Adaptive Locomotion of a Multilegged Robot over Rough Terrain. *IEEE Transactions on Systems, Man, and Cybernetics Journal*, 9(4):176–182, April 1979.

[5] Thomas A. McMahon. Mechanics of Locomotion. *The International Journal of Robotics Research*, 3(2):4–28, Summer 1984.

[6] Aftab E. Patla, Caroylin Robinson, Michelle Samways, and Crystal J. Armstrong. Visual Control of Step Length During Overground Locomotion: Task-Specific Modulation of the Locomotor Synergy. *Journal of Experimental Psychology: Human Perception and Performance*, 15(3):603–617, 1989.

[7] Cary B. Phillips and Norman I. Badler. Interactive Behaviors for Bipedal Articulated Figures. *Computer Graphics*, 25(4):359–362, July 1991.

[8] Marc H. Raibert and Jessica K. Hodgins. Animation of Dynamic Legged Locomotion. *Computer Graphics*, 25(4):349–358, July 1991.

[9] Jr. William H. Warren, David S. Young, and David N. Lee. Visual Control of Step Length During Running Over Irregular Terrain. *Journal of Experimental Psychology: Human Perception and Performance*, 12(3):259–266, 1986.