Bucknell University
Bucknell Digital Commons

Master's Theses | Student Theses

2011

Energy-Economical Heuristically Based Control of Compass Gait Walking on Stochastically Varying Terrain

Christian Hubicki
Bucknell University

Follow this and additional works at: https://digitalcommons.bucknell.edu/masters_theses

Part of the Mechanical Engineering Commons

Recommended Citation
Hubicki, Christian, "Energy-Economical Heuristically Based Control of Compass Gait Walking on Stochastically Varying Terrain" (2011). *Master's Theses*. 26.
https://digitalcommons.bucknell.edu/masters_theses/26

This Masters Thesis is brought to you for free and open access by the Student Theses at Bucknell Digital Commons. It has been accepted for inclusion in Master's Theses by an authorized administrator of Bucknell Digital Commons. For more information, please contact dcadmin@bucknell.edu.

ENERGY-ECONOMICAL HEURISTICALLY BASED CONTROL OF COMPASS GAIT WALKING ON STOCHASTICALLY VARYING TERRAIN

by

Christian Michael Hubicki

(A Thesis)

Presented to the Faculty of
Bucknell University
In Partial Fulfillment of the Requirements for the Degree of
Master of Science in Mechanical Engineering

Approved:

Adviser

Department Chairperson

Engineering Thesis Committee Member

Engineering Thesis Committee Member

May 2017
(Date: month and year)

I, Christian Hubicki, do grant permission for my thesis to be copied.

ACKNOWLEDGEMENTS

To Mom, Dad, and Alexa, who have always supported me in all my pursuits,

Who always lend a caring ear and never miss an opportunity to feed me,

Who sacrifice routinely for their family in ways I can never repay.

To Keith, who took a chance on an overly confident student five years ago,

Who always made room in his daunting schedule if I needed help,

Who never failed to prove that he was looking out for me.

To Emily, who always makes a rainy day a joy,

Who has dinner cooked when I come back from the lab,

Who could not possibly know how happy she makes me every day.

Thank you all.

Table of Contents

Page

- Acknowledgements i
- Table of Contents ii
- List of Tables vi
- List of Figures vii
- Abstract xi
- Chapter 1: Introduction 1
  - Controlling Walking Robots 2
  - Zero-Moment Point Control 2
  - Passive-Dynamic Walking 3
  - Underactuated Systems 4
  - Limit-Cycle Stability and Robustness 6
  - Robust Biped Control 7
  - Metastability 9
  - Performance Tradeoffs 11
  - Goal Statement 12
- Chapter 2: Simulation Model 13
  - Hybrid Continuous/Discrete Dynamics 13
  - Poincare Section 14
  - Terrain-Crossing Conditions 15

- Compass Gait Continuous Dynamics 17
- Collisions 21
- Impulse 23
- Stochastic Terrain Model 24
- Validation 27
- Actuation and Control 27

- Chapter 3: Genetically Optimized Gain Scheduled Control 29
  - Proportional-Derivative Control 29
  - Gain-Scheduled Control 31
  - Control Parameter Set 34
  - Genetic Optimization 34
  - “Methinks it is like a weasel” 35
  - Mutation 36
  - Fitness Function 37
  - Robustness Cost Function 38
  - Failure Modes 39
  - Energy-Speed Weighting Factor 41
  - Reproduction 42
  - Convergence 43
  - Data Collection 44
  - Genetic Algorithm Results 47

- Conclusions 48
- Chapter 4: Heuristic Control 50
  - Optimization-Inspired Heuristic 50
  - Selection Pressures 54
  - Random Walk 54
  - Impulse Selection Pressures 56
  - Gain Schedule Selection Pressures 59
  - Mean Gain Schedule 62
  - Tradeoff-Conducive Control Heuristic 64
  - Heuristic Bounding Parameters 66
  - Heuristic Parameter Tuning 67
  - Gradient-Descent Algorithm 68
  - Tradeoff Curve Metrics 69
  - Approximated Gradient 71
  - Heuristic Tradeoff Curve Results 73
  - Conclusions 79
- Chapter 5: Reinforcement Learning 80
  - Artificial Intelligence 81
  - Machine Learning 81
  - Reinforcement Learning 82
  - State and Action Value Functions 82

- Value Iteration 84
- Mean First-Passage Time 88
- Approximate Optimal Robustness 89
- Value-Iteration Robustness Results 90

- Chapter 6: Simulated Walking Experiment 93
  - Value-Iteration Cost Function 93
  - Action Space 94
  - Discrete Dynamics 96
  - Walking Experiment Results 97
  - Walking Experiment Conclusions 100
- Chapter 7: Conclusions and Future Work 102
  - Summary 102
  - Conclusions 103
  - Future Work 104
- Bibliography 106
- Appendix 110
  - Simulated Walking Experiment Code 110
  - Approximate Optimal Robustness Code 169
  - Genetic Optimization Algorithm Code 196
  - Gradient-Descent Heuristic Parameter Tuning Code 222

List of Tables

Page

Table 3.1: Parameters used in the genetic algorithm for mutating and reproducing the control parameter sets 43

Table 3.2: The maximum and minimum values denoting the range of state space used to generate the reported data with the genetic algorithm 45

Table 4.1: State variables used for testing the gradient-descent algorithm 71

Table 4.2: Gradient-descent exploration values by which heuristic bounding parameters are changed in order to find the path of greatest descent; the initial bounding parameter values for the algorithm are also included in this table 73

Table 4.3: State variables used for a second run of the gradient-descent algorithm 76

Table 5.1: Discretization of variables for approximating the system states, actions, and terrain heights 83

Table 6.1: State variables used for gradient-descent algorithm to obtain the six heuristic bounding parameters for the simulated walking experiment 95

Table 6.2: Heuristic bounding parameters resulting from gradient-descent algorithm for the simulated walking experiment 96

Table 6.3: Discretization of variables for approximating the system states, actions, and terrain heights for simulated walking experiment 97

List of Figures

Page

Figure 1.1: Honda Motor Company's prototype humanoid robot, ASIMO, which is likely the most well-known example of bipedal robot control using zero-moment point methods 3

Figure 1.2: Dynamic bipedal robots built by Collins and Ruina at Cornell University; Collins robot (left) and Cornell Ranger (right). 4

Figure 1.3: A visualization of the "Acrobot" (left) and a stroboscopic sequence of various attempts to balance it (Wiklendt 2009) using a spiked neural network approach (right). 5

Figure 1.4: Visuals of the first reference (Espiau 1994) to the compass gait walking model (left) and its current implementation (Byl 2008) with a more obvious resemblance to the Acrobot (right). 6

Figure 1.5: Basin of attraction depicted by shaded region (left) for pictured compass gait model (right) (Byl 2009) 7

Figure 1.6: Illustration of the concept of "capture regions" (Pratt 2006), which are regions which to place the foot center of pressure to recover from a push 8

Figure 1.7: The M2V2 humanoid robot developed by the Institute for Human and Machine Cognition (Pratt 2008) 9

Figure 1.8: Visualization of stochastic terrain for the compass gait model 10

Figure 1.9: The results of control of the compass gait model on rough terrain using a Value-Iteration Reinforcement Learning Algorithm (Byl 2009) 11

Figure 2.1: Five stages of the single-step transfer function beginning at Poincare section i and terminating at section $i+1$: detect terrain crossing of lead leg, apply instantaneous impulse in line with trailing leg, compute plastic collision at leading leg, swap ground revolute joint and state variables, compute continuous dynamics with hip-torque actuation until terrain cross is detected 17

Figure 2.2: A diagram of the utilized compass gait model 19

Figure 2.3: Rigid body representation of the compass gait for collision computations 22

Figure 2.4: Illustration of the core concept of the stochastic terrain model: a ground height which varies in accordance to a given probability distribution 25

Figure 2.5: Frame of animation of walking sequence picturing a representative terrain roughness (Note: smoothness of terrain in animation is purely aesthetic) 27

Figure 3.1: Compass gait model discretized by interleg angle for gain-scheduled control 32

Figure 3.2: The surviving control parameter set's fitness value plotted over 80 generations, indicating that the convergence criterion is met at generation 59 44

Figure 3.3: 55 solutions generated by the genetic algorithm (one data set) plotted in energy-speed tradeoff space with a quadratic data fit 48

Figure 4.1: All impulse magnitude values for 51 genetic optimizations plotted with a linear trend line, revealing a strong linear correlation 51

Figure 4.2: All proportional gain schedule values for 51 genetic optimizations, revealing no obvious trend 52

Figure 4.3: All derivative gain schedule values for 51 genetic optimizations, revealing no obvious trend 53

Figure 4.4: The percentile values of 5000 normal ($\sigma = 0.125$) random walks over time (50% indicating the median, 75% denoting the third quartile, etc.) 55

Figure 4.5: A sample genetic optimization following the change in impulse magnitude over 80 generations. 56

Figure 4.6: Plot of impulse magnitude drift due to the genetic algorithm against the random walk probability profiles (i.e., at generation 15, over 99% of random walks produced drift numbers greater than the impulse magnitude drift at that time, meaning that less than 1% of random walks produced such extreme values) 57

Figure 4.7: A statistical analysis of 5000 random walks with $\sigma = 0.125$ (identical to impulse mutation rate), observing the percentage of random walks which remained within 0.08 of their starting value over several generations 58

Figure 4.8: A sample genetic optimization following the change in the proportional gains in the gain schedule over 80 generations 59

Figure 4.9: A sample genetic optimization following the change in the derivative gains in the gain schedule over 80 generations 60

Figure 4.10: Beginning at the generation of convergence, the drift in proportional gain is plotted against the percentile ranges of random walks (i.e., the 50% line indicates the median value, 75% is the third quartile value). The dotted lines from bottom to top are the following percentages: 1%, 5%, 10%, 25%, 50%, 75%, 90%, 95%, and 99%. 61

Figure 4.11: Beginning at the generation of convergence, the drift in derivative gain is plotted against the percentile ranges of random walks (i.e., the 50% line indicates the median value, 75% is the third quartile value). The dotted lines from bottom to top are the following percentages: 1%, 5%, 10%, 25%, 50%, 75%, 90%, 95%, and 99%. 62

Figure 4.12: Mean values for 51 optimized gain schedules produced by the genetic algorithm for various weighting factors. Each point indicates a gain associated with a lower-bound swing angle ratio range in the gain schedule. 63

Figure 4.13: Several genetically optimized values (all 51 genetic optimizations) of a single, sample sector of the proportional gain schedule plotted against the resulting controller speed (essentially a single gain schedule entry from Figure 4.2) 65

Figure 4.14: Several genetically optimized values (all 51 genetic optimizations) of a single, sample sector of the derivative gain schedule plotted against the resulting controller speed (essentially a single gain schedule entry from Figure 4.3) 65

Figure 4.15: A visualization of the gradient descent process on a contour plot, progressing from initial guess ($\boldsymbol{x_0}$) to the most recent approximate minimum ($\boldsymbol{x_4}$) by traversing the maximum gradient. 69

Figure 4.16: Illustrations of energy-speed tradeoff curves highlighting examples of varying performance in mean energy and speed range 70

Figure 4.17: An illustration of a hypothetical one-dimensional (left) and two-dimensional performance curves. The one-dimensional case shows how moving in the two possible directions (left and right) yields predictions of the gradient. The two-dimensional case indicates the diagonal motion used to explore and approximate a higher dimensional gradient. 72

Figure 4.18: Tradeoff curves for genetically optimized solutions and the tuned heuristic tradeoff controller 74

Figure 4.19: Tradeoff curves for genetically optimized solutions and the tuned heuristic tradeoff controller for the second set of state variables and desired step angle 77

Figure 4.20: Comparison of the mean "ramped" and "flat" gain profiles on their performance in energy-speed tradeoffs 78

Figure 5.1: Visualization of state-value functions (V) and action-value functions (Q) as vectors indexed by i (enumerating distinct states) and k (enumerating distinct actions) 84

Figure 5.2: Markov Decision Process (MDP) Matrix 85

Figure 5.3: A visualization of the relationship between the action-value function (Q), transition matrix ($T_{ij}^{a_k}$), and state-value function (V) where n represents the current step number, i is the current state number, j is the state number potentially occupied for the next step, and k is an index enumerating all of the available state actions 87

Figure 5.4: Block diagram of the value-iteration reinforcement learning process 88

Figure 5.5: Visualization of the updating process for the state-value function using the value of the best action ($min(Q_n)$) 88

Figure 5.6: Comparison of resulting mean-first passage times of the value-iteration algorithm with published data (Byl 2009) 91

Figure 6.1: Energy-Speed Tradeoff Curve for 1 km Walk using Value-Iteration compared to the Tuned Single-Step Curve 99

Abstract

The field of legged robotics has been long anticipated in the popular media to herald a revolution in both civilian and military life. From mechanical fire fighters barreling through burning apartments with minimal regard for self-preservation to nimble explorers bounding up Martian ridges who never complain about the cold, finding applications for bipedal machines requires little imagination. Despite their promised dexterity and overall popular appeal, in the early 21st century, bipedal robots are seldom sighted outside of university research labs or cutting-edge technology firms.

The absence of these legged machines in our daily lives can be attributed to significant technical barriers in performance. The largely untold flaw of Honda's flagship robotic humanoid, ASIMO, is that its exorbitant energy consumption drains its generously sized battery pack in roughly 30 minutes, nullifying its utility outside of relatively short public demonstrations. Recognizing that this energy limitation is not unique to ASIMO but common among current-generation walking robots, academic researchers have recently pushed to develop highly energy-economical bipeds. The consequence has been a series of prototypes which trade an abundance of actuation and control authority for an underactuated approach dubbed Dynamic Walking. Specifically, Cornell University developed two internationally publicized walking machines; one which boasted energy economy on par with human walking (for short distances) and the Cornell Ranger which set a world record for walking 5.6 miles on a single battery charge.

While delivering such significant advances in energy economy, dynamic walking robots have still largely fallen short in applications with high speed requirements or

rough terrain. This investigation uses simulation to explore the inherent tradeoffs of controlling high-speed and highly robust walking robots while minimizing energy consumption. Using a novel controller which optimizes robustness, energy economy, and speed of a simulated robot on rough terrain, the user can adjust their priorities between these three outcome measures and systematically generate a performance curve assessing the tradeoffs associated with these metrics.

The novel robot controller is a two-tiered hierarchical system consisting of a tradeoff-conducive control heuristic used for individual steps and an overseeing Artificial Intelligence algorithm to decide which step to take. The tradeoff-conducive control heuristic is shown to have marked advantage over traditional proportional-derivative controllers. This control heuristic rapidly generates controllers which span a wide range of step speed and energy economy for the simulated biped. Generated controllers are also shown to produce the same step speed while using smaller energy budgets than their traditional counterparts. The overseeing algorithm (a value-iteration reinforcement learning algorithm) is demonstrated to be capable of selecting these single-step controllers in a manner resulting in sustained walks over a kilometer in length while producing the desired energy-speed tradeoffs.

Chapter 1: Introduction

Ever since 1959, robots have been built to assist humans in a variety of dull, dirty, and dangerous jobs (Kurfess 2005). From the earliest industrial robots which were used in such applications as painting wheelbarrows, applications for robots have ballooned into countless sectors of research, industrial, and military enterprises. Robots assemble our cars, inspect for bombs, perform surgery, explore Mars, and sweep our floors. Despite all these advances in technology, robots still struggle to do what many people consider to be trivial. Robots cannot yet walk like humans.

While many robots have been built which can repeatedly place one foot in front of the other, none can do so on the same energy budget as humans without sacrificing the stability and agility of which humans are capable. To emphasize the point, arguably the world's most famous bipedal robot, ASIMO, consumes an estimated 16 times the amount of energy that a human requires to walk (Collins 2005). The problem is profound and high-profile enough that a $200,000 "W-Prize" has been offered for a robot capable of traversing a ten-kilometer obstacle course with limited time and a strict energy budget. This prize remains unclaimed as it is simply very difficult to make a robot so robust to avoid falling, economical in energy consumption, and sufficiently speedy to meet the requirements on an obstacle course.

This problem for walking robots is disappointing as human-like locomotion is a critical means of navigating urban environments. Humans can bound up stairs, step over obstacles, squeeze into elevators, and dart around other humans. Humans are also capable of handling extreme natural terrains like cliff walls, thick forests, mountains, and

sandy deserts. Before even beginning to address these extreme conditions, solutions must be found for designing robots which can walk with performance on par with humans. Much of this room for improvement may be filled by advances in robot control.

Controlling Walking Robots

Walking robots are plagued with some significant technical barriers for entry into military, industrial, and consumer markets. With exception to some recently-developed robust prototypes such as the M2V2 (Pratt 2008), bipedal robots simply fall too easily to be left unattended even in the absence of significantly challenging terrain or antagonistic agents. Compensating for this lack of robustness, many prototypes have traditionally employed fully-actuated control systems to dominate the dynamics and eliminate falls. By using such heavy actuation, these control strategies inherently constrain the overall robot agility and require extravagant energy budgets to implement (Collins 2005).

Zero-Moment Point Control

The origins of modern, formalized bipedal robot control date back as far as 1968. Miomir Vukobratovic produced a number of papers which acted as the foundation for Zero-Moment Point (ZMP) Control (Vukobratovic 2004). In effect, the ZMP approach to locomotive control preserves the dynamic balance of the biped for the entirety of each stride. This approach regulates the motion of the biped's mechanical linkages such that the biped's weight and reaction forces can be counteracted by a single point load applied at a point (the zero-moment point) by the foot. If this force is applied within the foot

area, it ensures that the sole of the robotic foot is in full, flat contact with the surface. This balanced gait and "stable" ground-foot contact eliminates many of the dynamic challenges associated with bipedal gait control. The ZMP is a concept utilized pervasively in the field of bipedal robotic control. Perhaps the most notable instance of ZMP implementation is the Honda Motor Company's flagship humanoid robot, ASIMO, which is pictured in Figure 1.1.

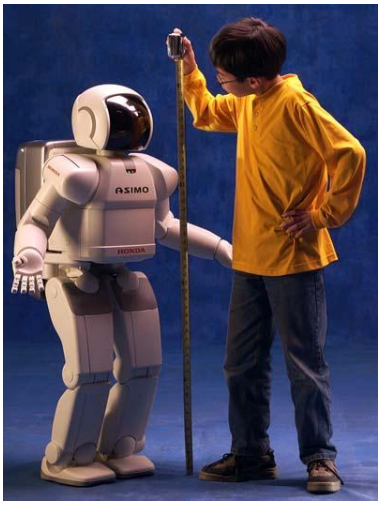

Figure 1.1: Honda Motor Company's prototype humanoid robot, ASIMO, which is likely the most well-known example of bipedal robot control using zero-moment point methods

Passive-Dynamic Walking

Since 1990 (McGeer 1990), there has been a push by researchers to use the inherent dynamics of legged systems, not an abundance of actuation, to facilitate forward motion and stability. Sacrificing the luxury of complete control authority over the physical state of the robot resulted in a considerable alleviation of its energy burden. This finding gave rise to the field of passive-dynamic walking, an approach which inspired walking machines capable of achieving stable gaits using a shallow downward

slope as its solitary energy source, as shown for example on the left in Figure 1.2. Requiring such meager resources, these bipeds became the mold for so-called *dynamic walking* robots, which seek to minimize actuation costs of level-terrain walkers (Collins 2005). Such robots include Cornell's "Ranger", shown on the right in Figure 1.2, which currently holds the record for walking 5.6 miles, the longest distance walked by a machine without being touched or refueled (Karssen 2007).

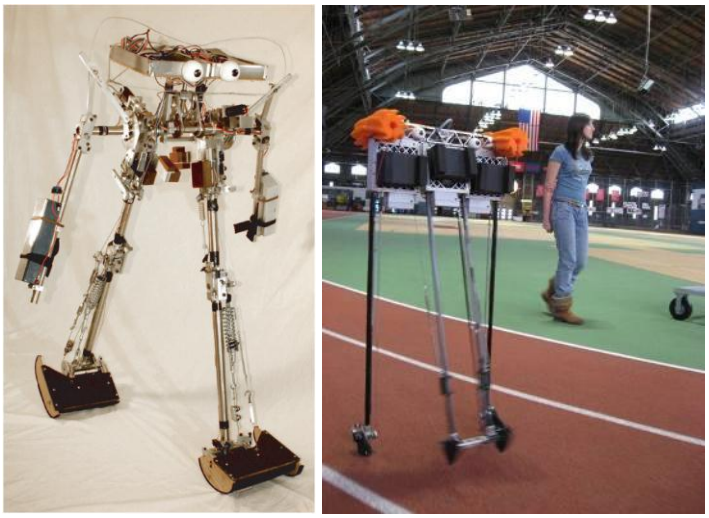

Figure 1.2: Dynamic bipedal robots built by Collins and Ruina at Cornell University; Collins robot (left) and Cornell Ranger (right).

Underactuated Systems

While the energetic performance of dynamic walking robots is promising and their gaits are technically stable, relatively small disturbances can force the robot into an irrecoverable state. Furthermore, the relinquishing of control authority that allowed for the development of such economical machines has moved these bipeds firmly into the category of *underactuated mechanical systems* (Spong 1998). An underactuated system

is one that lacks the proper number of actuators to control the number of degrees of freedom in the system.

A classic and relevant example of underactuation is the gymnastic “acrobot”. Figure 1.3 (left) depicts the acrobot as a double pendulum with a single actuator providing a torque at the distal joint. The challenge of this system is to design a controller to balance the acrobot upright in a “headstand” as shown in a stroboscopic representation in Figure 1.3. Despite having a mere two degrees of freedom, controlling this system proves to be deceptively complex and has been approached using techniques as sophisticated as spiked neural networks and genetic algorithms (Wiklendt 2008).

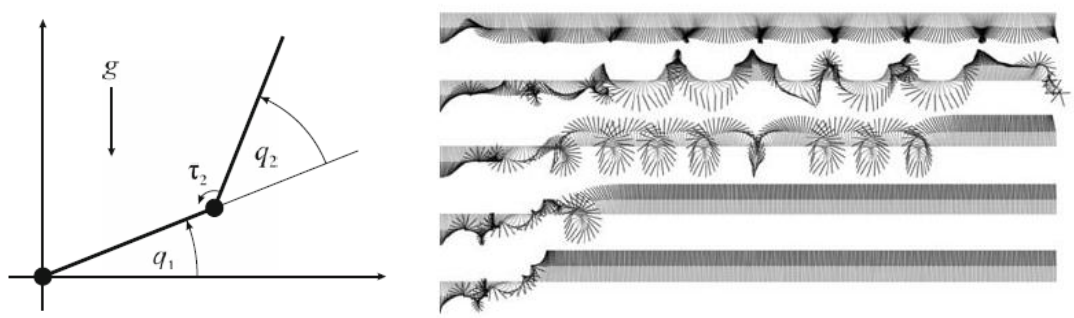

Figure 1.3: A visualization of the “Acrobot” (left) and a stroboscopic sequence of various attempts to balance it (Wiklendt 2008) using a spiked neural network approach (right).

The acrobot provides a particularly apt example for not only under-actuated systems, but also a simple model for walking machines called the *compass gait*. The original compass gait walking model (Espiau 1994) as shown in Figure 1.4 (left), like the acrobot, is a double pendulum actuated only through a torque applied at the revolute-jointed hip. It is casually noted in recent papers (Byl 2008) that the compass gait model is dynamically equivalent to the acrobot, a comparison which is more obvious when

viewing the compass gait visualized in Figure 1.4 (right). While the goals of compass gait control are dissimilar to the common acrobot balancing challenge, the comparison reveals the need for implementation of complex control systems for even a single leg swing, let alone a series of steps. This underscores the nonlinear nature of the compass gait model and the consequent challenges associated with its control.

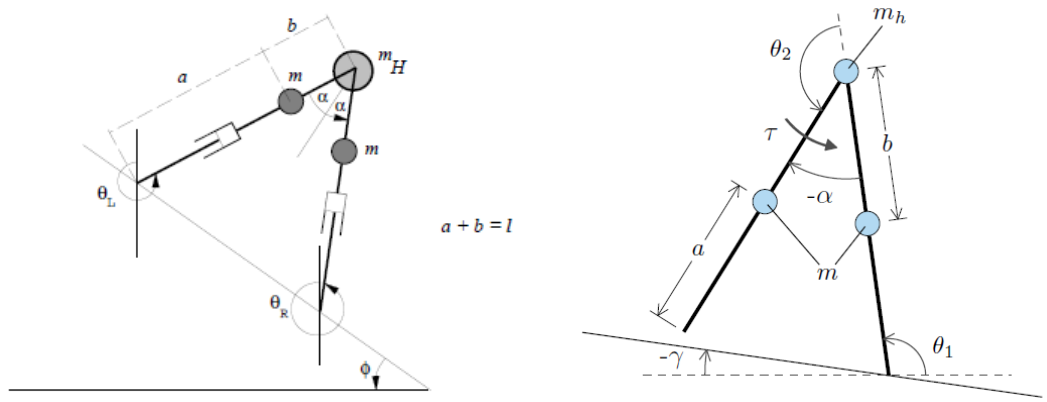

Figure 1.4: Visuals of the first reference (Espiau 1994) to the compass gait walking model (left) and its current implementation (Byl 2008) with a more obvious resemblance to the Acrobot (right).

Limit-Cycle Stability and Robustness

A number of investigations have been published studying the compass gait in the purely passive case, i.e. zero hip torque. The literature regarding the stability analysis of the two-dimensional passive walker has been numerously replicated and the methods are well-established within the dynamic walking community. In such a system where there is no active controller, the most common means of achieving a self-perpetuating gait is through limit-cycle walking (Hobbelen 2007). Limit-cycle walking is achieved when each step is dynamically identical to the previous step, resulting in a sustained (but not

necessarily stable) gait. The gait is deemed "stable" if, when given a small perturbation, the system converges back to a limit cycle gait.

Furthermore, the ability of the machine to reject larger disturbances reflects the system's robustness. The region of the walker's state-space over which the system converges to a sustained gait is dubbed the *basin of attraction*. The size of this basin acts as an indicator of system robustness, as shown in Figure 1.5 (left) for a passive compass-gait walker. Figure 1.5 (right) defines the state variables for the compass gait used on the axes for the plotted basin of attraction. It has been an ongoing goal for dynamic walking researchers to increase the size of the attractive basin as currently-sized basins often result in generally poor disturbance rejection in practice (Byl 2009).

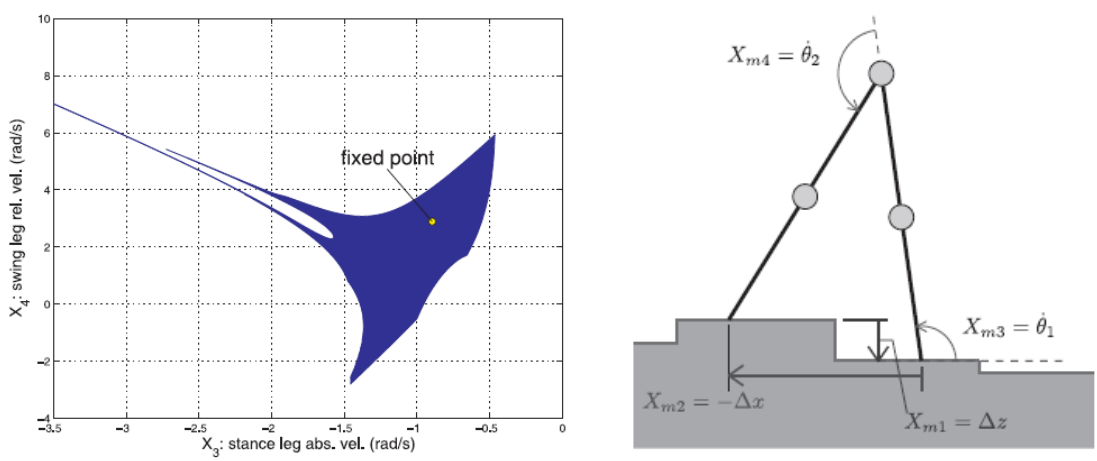

Figure 1.5: Basin of attraction depicted by shaded region (left) for pictured compass gait model (right) (Byl 2009)

Robust Biped Control

Various approaches are under investigation to satisfy the demand for more robust walking bipeds. One such method approaches robustness as a push recovery problem

(Pratt 2006). Balancing a biped on one foot is the archetypal problem for push recovery. If push recovery were implemented on a ZMP controlled robot such as ASIMO, small pushes require only an adjustment of the standing foot's center of pressure (CoP) to sufficiently maintain balance. However, for larger disturbances, it may be necessary to take additional steps to avoid falling. To assess whether such a step needs to be taken to regain balance after a push, a *capture region* is computed. A capture region is an area on the ground a foot's CoP must occupy to avoid a fall. If the capture region does not intersect the standing foot, a step must be taken by the raised foot which lands in the capture region as illustrated in Figure 1.6.

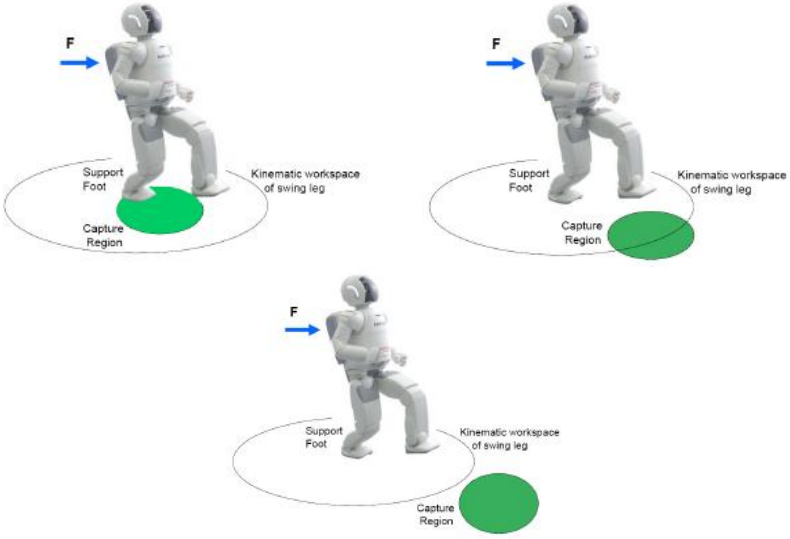

Figure 1.6: Illustration of the concept of "capture regions" (Pratt 2006), which are regions in which to place the foot center of pressure to recover from a push

Furthermore, capture regions have been expanded upon to solve more problems than simple push recovery. Capture regions have been used as a means of solving

intermittent terrain problems (i.e., stepping stones) via multiple capture regions. If a capture region is not reachable in one step, perhaps it is possible to reach by taking more steps. Intermediate capture regions are then defined to reach the next capture region. Capture regions can even be used as a generalized walking approach, treating a sequence of steps as a series of forward falls, as utilized for the control of the IHMC M2V2 (Pratt 2008) shown in Figure 1.7. Using capture points has shown superior robustness to traditional ZMP approaches, which makes for an excellent safe-guard against falling when large disturbances are detected. However, the method lacks the utilization of inherent dynamics that make dynamic walkers energetically economical.

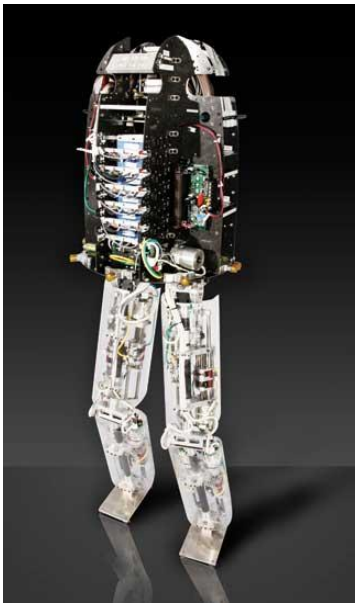

Figure 1.7: The M2V2 humanoid robot developed by the Institute for Human and Machine Cognition (Pratt 2008)

Metastability

An alternative approach to robust walking has been recently developed for dynamic walkers using the concept of “metastability” (Tedrake 2006). While dynamic

walkers have been traditionally controlled with a limit cycle gait in mind, taking a metastability-based approach allows for the robot states to “wander” around a much larger region in state-space, so long as the transitory states do not lead to walking failure. Furthermore, a metastable approach does not require a deterministic model of reality. The dynamics can be modeled probabilistically which allows for the addition of stochastic disturbances. As such, stochastic terrain can be incorporated in the walking model (as depicted in Figure 1.8) and can be approached using metastability methods.

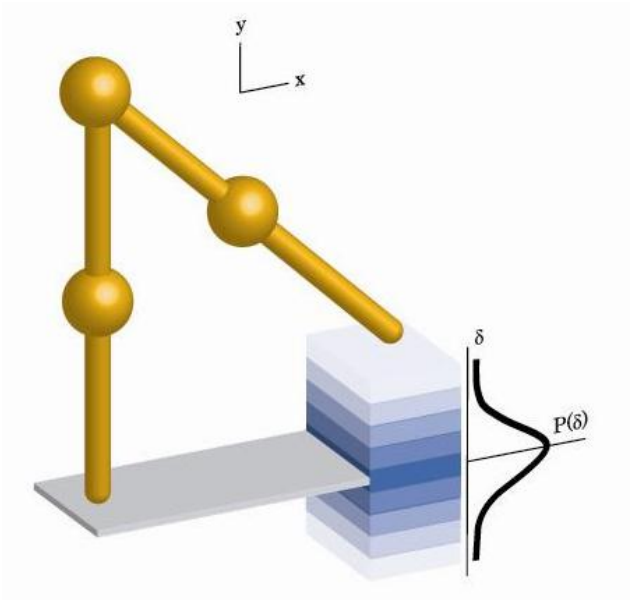

Figure 1.8: Visualization of stochastic terrain for the compass gait model

In essence, if the system is controlled in a manner that is highly metastable (walks for many steps without falling) on rough terrain, then such an approach would be considered highly robust. By using an artificially intelligent algorithm, “approximate optimal control” (Byl 2008) of the compass gait on rough terrain was developed to maximize the number of steps to failure. The results of research by Byl and Tedrake at MIT for controlling of the compass gait model on rough terrain using this method are

shown in Figure 1.9. This approach has resulted in simulated walkers which take as many as an estimated 10^{14} steps (denoted by a metastability metric called *mean first-passage time* or MFPT) before falling on rough terrain.

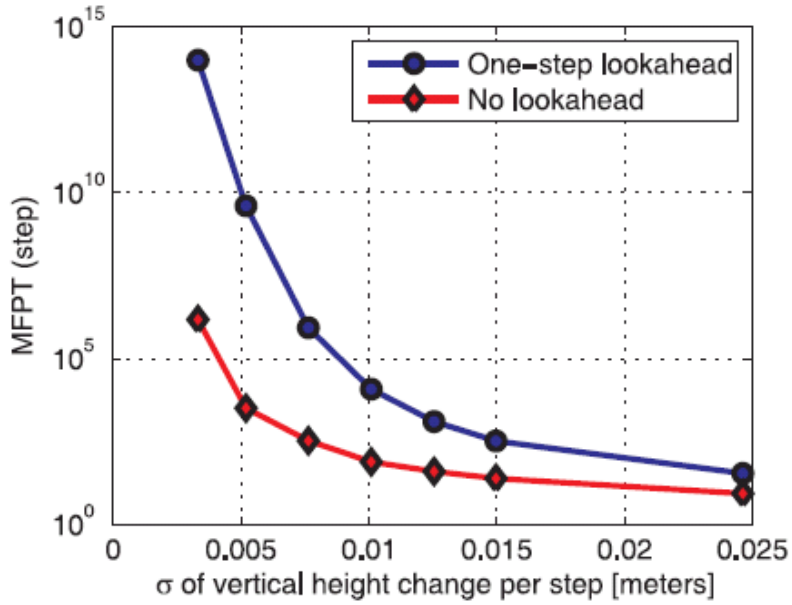

Figure 1.9: The results of control of the compass gait model on rough terrain using a Value-Iteration Reinforcement Learning Algorithm (Byl 2009)

Performance Tradeoffs

It is clear that the metastability approach to handling rough terrain walking is quite powerful in developing highly-robust controllers. While an impressive result for robustness, the actuation utilized in these highly robust simulations are far from economical in regard to energy consumption. Energy economy is a significant motivator for the development of dynamic walking methods and should be kept in focus.

Furthermore, decreases in energy consumption are likely to result in a loss in walking speed. It also remains unknown how changing walking speed will impact walker's robustness and vice versa. The result of these possibly synergistic or

antagonistic relationships may result in an interesting tradeoff problem. To investigate any such relationship, a means must be developed to synthesize controllers which can optimally meet demands of robustness, energy economy, and speed to the desires of a user.

Goal Statement

The goal of this thesis is to produce a method of synthesizing controllers capable of controlling a simulated walking robot on rough terrain. Furthermore, the aim is to traverse such terrain while being able to produce a wide range of performance over three key parameters: robustness, energy economy, and speed. Using the techniques employed for metastable walking as a starting point, supplemental methods for controlling single steps with high speed or low energy cost must be developed via optimization techniques. In turn the metastability methods must be modified to accommodate more than the single robustness metric. Accomplishing such a feat would be a novel contribution to the field of dynamic walking.

Chapter 2: Simulation Model

Central to any simulation-based investigation is the definition of the system model. In dynamic walking, a number of models have been used in the study of gait control. Some models have complex kinematic layouts incorporating feet, knees, an upper body, and sometimes arms (Yin 2007). Models have more recently begun to incorporate springs which may produce dynamics more advantageous to walking (Hurst 2008). Some models are so rudimentary that their relationship to walking is less intuitive, as is the case in the example of the rimless wheel (McGeer 1990).

Among the simpler of the proposed frameworks is the compass gait model. Ignoring effects such as three-dimensional dynamics, foot-slipping, and collision elasticity, the compass gait model provides a platform upon which the most fundamental principles of bipedal walking can be isolated and probed. Variations upon the compass gait have been used as the basis for foundational research on the stability (Espiau 1994), energy economy (Kuo 2002), and terrain robustness (Byl 2009) of dynamic bipedal locomotion. Its relative simplicity and considerable precedence render the compass gait most conducive to investigation into the control of performance tradeoffs in dynamic bipedal robots.

Hybrid Continuous/Discrete Dynamics

On the most basic level, this simulation uses a hybrid system of continuous and discrete dynamics: the compass gait walking model being modeled as a discrete series of dynamically continuous steps. Governed by Newtonian mechanics for the swing of each

leg, the continuity is punctuated by a series of impact events resulting from the swing leg colliding with the terrain. This discretization, in addition to being necessary in the modeling of ground impacts, has advantages in the analysis of the long-term gait. Instantaneously prior to these impacts, a conceptual snapshot is taken called a Poincare section. This concept is vital to the analysis of dynamic walking.

Poincare Section

A critical tool for analyzing continuous systems on a discrete level, a Poincare section is a representative snapshot of the system states. If the system state variables are an accurate and sufficient representation of the dynamics, these recorded state variables taken at this instant can be used as indicators of performance on a greater time scale. In application to dynamic walking, a Poincare section can be taken immediately preceding the swing leg's collision with the ground, capturing the state variables at that instant. Subsequently, a Poincare section is taken in the same situation for each of the following steps, generating a discrete series of representative states in a sequence of steps. If these states are identical over the series of sections, the walker is considered to be in a limit-cycle condition, indicating each step is dynamically equivalent to the last. In visualization, the walkers gait would appear perfectly steady. More complex linear-algebra-based analysis has been used to characterize the stability of such gaits using this discrete framework (Goswami 1996). This framework will serve here as a basis for a form of robustness analysis contingent upon a discrete system formulation.

In this study, as per the aforementioned example, a Poincare section is defined at the instant immediately prior to the swing leg collision. This situation is defined as the first time-step in which the swing leg has met terrain-crossing conditions. Figure 2.1 depicts the sequence of events which occur between hypothetical Poincare section i and its subsequent counterpart Poincare section $i+1$. The transition from one Poincare section to the following section is defined as the Step-to-Step transfer function. The step-to-step transfer function comprises five stages: terrain cross detection, pre-collision impulse actuation, swing leg collision, swing/stance leg switch, applied hip torque and continuous dynamics.

Terrain-Crossing Conditions

Terrain-crossing conditions, the criteria at which a Poincare section is defined, are only met when the swing leg crosses the current terrain boundary and vertical velocity of the end point of the leg with respect to ground is negative, which precedes any collision computations or applied impulses. This criterion prevents the inevitable "scuffing" that occurs with straight-legged walkers that cannot reduce their leg length mid-step. Both legs being the same length, as the swing leg approaches the stance leg, the swing leg must cross the terrain boundary to which the stance leg is connected. In effect, this would cause the swing leg to "scuff" the ground. A common assumption to avoid scuffing is to simply turn off collision detection until the legs cross each other after some arbitrary small separation distance. To meet this anti-scuffing requirement, terrain crossing

detection is only active once the swing leg is 5 cm in front of the stance leg ground pivot, which approximates the act of retracting the swing leg to avoid premature collisions.

The leg retraction that would be necessary in an actuated device is modeled as have zero dynamical significance outside of collision detection, which is mirrored in the design of prototypes (Iida 2009) which seek to minimize the impact of this retraction in the design. A step is considered a failure if the simulation fails to terminate after five simulation seconds or the main body crosses the terrain boundary, as this indicates that the walker has fallen backward or tripped forward prior to activating the collision detection. If a step failure occurs, a Poincare section is taken but is tagged with a flag indicating the occurrence of a failure.

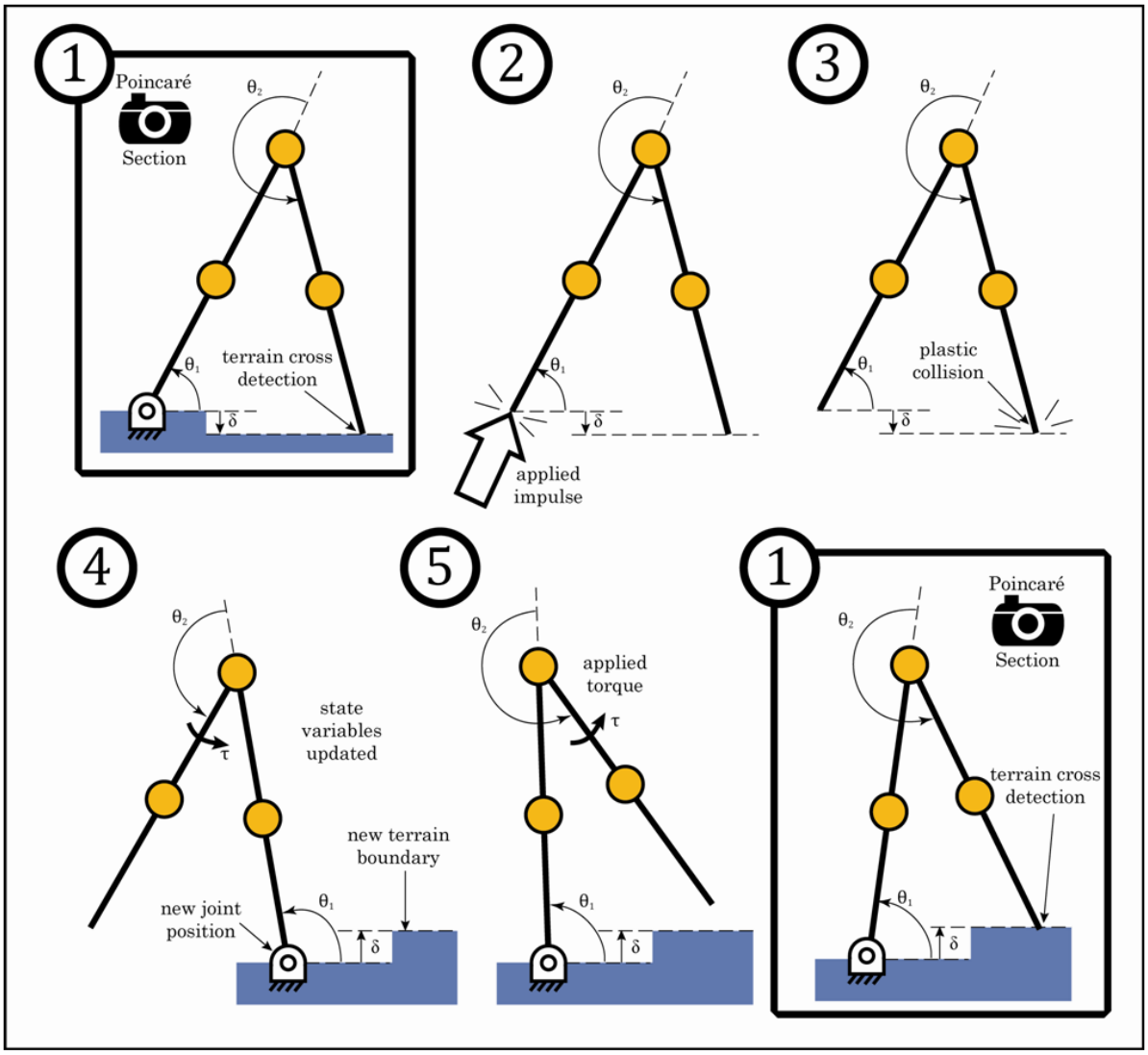

Figure 2.1: Five stages of the single-step transfer function beginning at Poincare section *i* and terminating at section *i+1*: detect terrain crossing of lead leg, apply instantaneous impulse in line with trailing leg, compute plastic collision at leading leg, swap ground revolute joint and state variables, compute continuous dynamics with hip-torque actuation until terrain cross is detected

Compass Gait Continuous Dynamics

Disregarding the discrete impact events, the compass gait model is essentially a double pendulum. The planted (stance) leg is connected to ground via a revolute joint.

In turn, the swing leg is revolute-jointed to the free-swinging end of the stance leg. Each leg is modeled as a massless rod with a lumped point mass at the center. At the connection of these two legs, the hip joint is the main body which is similarly modeled as a point mass. This continuous model includes one mode of actuation, a torque (τ) applied at the hip joint (the second mode of actuation, the pre-collision impulse, is discrete and is not included in the continuous model). The hip actuator exerts an ideal torque at the hip joint between both legs which serves to control the angle between the two legs (the "interleg" angle). The control law for this hip torque is described in the following chapter. The terrain boundary distance (δ), the vertical displacement with respect to the ground pivot, is recalculated for each step in accordance with a stochastic terrain model. A diagram of the utilized compass gait model is shown in Figure 2.2 which illustrates the kinematic layout, relevant masses and dimensions, key variables, coordinate system and the directionality of the actuating torque. These model parameters were chosen to replicate the parameters of similar research (Byl 2009).

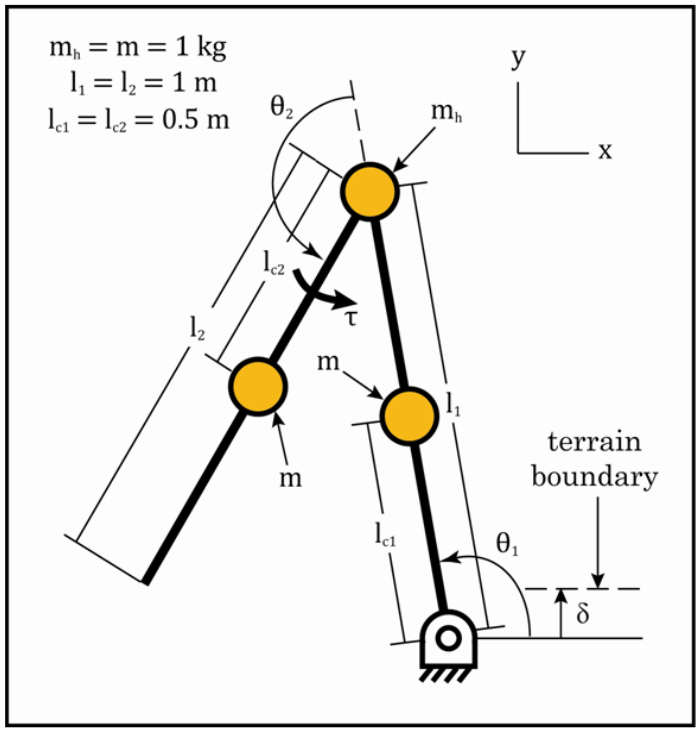

Figure 2.2: A diagram of the utilized compass gait model

The Compass Gait model yields four state variables, corresponding to the angles and angular velocities of each leg: θ_1, θ_2, and their respective time derivatives $\dot{\theta}_1$, and $\dot{\theta}_2$. These state variables, in conjunction with the hip actuator, are governed by the continuous acrobot dynamics (Spong 1994) between discrete impact events. The equations of motion and prerequisite variable assignments are given in Eq. 2.1-2.13. A Newton-Euler numerical solution is computed in MATLAB using a fixed time-step of 0.001 seconds. This time-step allowed the simulation to compute approximately twelve steps per second and resulted in numerical errors of less than 0.01 radians, which was deemed acceptable accuracy given that the controller will be subjected to stochastic terrain which will be a far more dominant effect over many steps.

$$m_1 = m + \frac{m_h}{2}$$ Eq. 2.1

$$m_2 = m_1$$ Eq. 2.2

$$I_1 = m\left((1 + \frac{m_h}{2m})(l_1 - l_{c1}) - l_{c2}\right)^2 + \frac{1}{2} m_h {l_{c2}}^2$$ Eq. 2.3

$$I_2 = I_1$$ Eq. 2.4

$$d_{11} = m_1 {l_{c1}}^2 + m_2\left({l_1}^2 + {l_{c2}}^2 + 2 l_1 l_{c2} \cos\theta_2\right) + I_1 + I_2$$ Eq. 2.5

$$d_{12} = m_2\left({l_{c2}}^2 + l_1 l_{c2} \cos\theta_2\right) + I_2$$ Eq. 2.6

$$d_{22} = m_2 {l_{c2}}^2 + I_2$$ Eq. 2.7

$$h_1 = -m_2 l_1 l_{c2} \sin\theta_2 \, {\dot{\theta}_2}^2 - 2 m_2 l_1 l_{c2} \sin\theta_2 \, \dot{\theta}_2 \dot{\theta}_1$$ Eq. 2.8

$$h_2 = m_2 l_1 l_{c2} \sin\theta_2 \, {\dot{\theta}_1}^2$$ Eq. 2.9

$$\varphi_1 = (m_1 l_{c1} + m_2 l_1) g \cos\theta_1 + m_2 l_{c2} g \cos(\theta_1 + \theta_2)$$ Eq. 2.10

$$\varphi_2 = m_2 l_{c2} g \cos(\theta_1 + \theta_2)$$ Eq. 2.11

$$\ddot{\theta}_2 = \frac{d_{11}(\tau - h_2 - \varphi_2) + d_{12}(h_1 + \varphi_1)}{\left(d_{11} d_{22} - {d_{12}}^2\right)}$$ Eq. 2.12

$$\ddot{\theta}_1 = \frac{d_{12}\ddot{\theta}_2 + h_1 + \varphi_2}{-d_{11}}$$ Eq. 2.13

Collisions

The process of walking, while otherwise modeled using continuous dynamics, is punctuated by a discrete series of impacts. In the utilized model, all collisions are assumed to be perfectly inelastic, which facilitates key features of the compass gait model. For the compass gait model to be valid, the stance leg must remain planted throughout the continuous leg swing. Any elasticity in the collision would inherently result in a momentary separation of the colliding leg and the ground. While the dynamics of an airborne biped can be calculated, the lack of a ground-reaction force to constrain the stance leg motion would likely result in highly aberrant limb behavior. Furthermore, subsequent re-collisions would ensue as a direct result of an airborne stance leg which would complicate a meaningful definition of a successful step. Collisions are one of the primary means of energy loss for the compass gait walker.

Collisions are modeled as occurring instantaneously with perfect plasticity, an event which exchanges the ground and free joints at the legs' distal points from the main body. Originally developed for a model more complicated than the compass gait model, the collision is computed using the visual model in Figure 2.3. Figure 2.3 details the three rigid bodies which are simplified to point masses in the compass gait model. The arrowed distances indicate the separation of the centers of mass of the rigid bodies (in this case, the masses are concentrated in the centers of the legs and at the hip) and the two revolute joints. Using angular momentum conservation equations, post-collision velocities are computed using the formulations in Eq. 2.15-2.18. The components of the distances in the x and y directions are used for the variables r_{2ax}, r_{2ay}, r_{2bx}, r_{2by}, r_{3ax}, r_{3ay},

r_{3bx}, and r_{3by}. The rotational inertias of the links about the center of mass (J_1, J_2 and J_3), because they are point masses, are zero (set to 10^{-6} kg-m^2 to avoid divide-by-zero errors). The leg angles are transformed, post-collision, in a manner which effectively swaps the swing and stance legs, allowing for a self-perpetuating walking sequence.

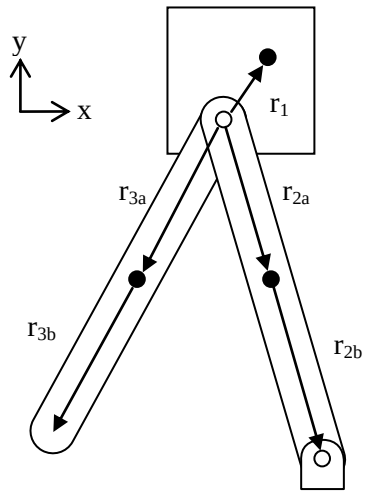

Figure 2.3: Rigid body representation of the compass gait for collision computations

$$A = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{m_1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & & & & & & & & 0 & \frac{1}{m_1} & & & & & & & & \\
0 & & 1 & & & & & & & \frac{r_{1y}}{J_1} & -\frac{r_{1x}}{J_1} & & & & & & & & \\
0 & & & 1 & & & & & & & & \frac{1}{m_2} & 0 & \frac{1}{m_2} & 0 & & & & \\
0 & & & & 1 & & & & & & & 0 & \frac{1}{m_2} & 0 & \frac{1}{m_2} & & & & \\
0 & & & & & 1 & & & & & & \frac{r_{2ay}}{J_2} & -\frac{r_{2ax}}{J_2} & \frac{r_{2by}}{J_2} & -\frac{r_{2bx}}{J_2} & & & & \\
0 & & & & & & 1 & & & & & & & & & \frac{1}{m_3} & 0 & \frac{1}{m_3} & 0 \\
0 & & & & & & & 1 & & & & & & & & 0 & \frac{1}{m_3} & 0 & \frac{1}{m_3} \\
0 & & & & & & & & 1 & & & & & & & \frac{r_{3ay}}{J_3} & -\frac{r_{3ax}}{J_3} & \frac{r_{3by}}{J_3} & -\frac{r_{3bx}}{J_3} \\
0 & & & & & & & & & 1 & & & & 1 & 0 & & & 1 & 0 \\
0 & & & & & & & & & & 1 & & & 0 & 1 & & & 0 & 1 \\
0 & & & & & & & & & & & 1 & & & & & & & \\
0 & & & & & & & & & & & & 1 & & & & & & \\
0 & & & & & & 1 & 0 & r_{3ay} & & & & & & & & & & \\
0 & & & & & & 0 & 1 & -r_{3ay} & & & & & & & & & & \\
-1 & 0 & & 1 & 0 & r_{2by} & & & & & & & & & & & & & \\
0 & -1 & & 0 & 1 & -r_{2bx} & & & & & & & & & & & & & \\
-1 & 0 & & & & & 1 & 0 & r_{3by} & & & & & & & & & & \\
0 & -1 & & & & & 0 & 1 & -r_{3bx} & & & & & & & & & & \\
\end{bmatrix} \qquad \text{Eq. 2.14}$$

$$b = [\dot{x}_1^- \quad \dot{y}_1^- \quad \omega_1^- \quad \dot{x}_2^- \quad \dot{y}_2^- \quad \omega_2^- \quad \dot{x}_3^- \quad \dot{y}_3^- \quad \omega_3^- \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0]^T \qquad \text{Eq. 2.15}$$

$$X = [\dot{x}_1^+ \quad \dot{y}_1^+ \quad \omega_1^+ \quad \dot{x}_2^+ \quad \dot{y}_2^+ \quad \omega_2^+ \quad \dot{x}_3^+ \quad \dot{y}_3^+ \quad \omega_3^+ \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots]^T \qquad \text{Eq. 2.16}$$

$$AX = b \qquad \text{Eq. 2.17}$$

Impulse

One of the primary modes of actuation for the compass gait (and rigid-linked walkers in general) is the pre-collision impulse. Modeled as an instantaneous push-off of the back-foot, the pre-collision impulse has been demonstrated to be an efficient means of imparting energy for forward motion of the biped (Kuo 2002). When tested on rough terrain (Byl 2009), impulse actuation was necessary to successfully traverse terrain with significant roughness. This finding was replicated with this model, showing that a pre-collision impulse was important in rough terrain walking. The effect of the impulse is calculated in a very similar method to the collision computation, and is in effect, an intentional collision. The equations for the impulse calculation are shown in Eq. 2.19-2.22.

$$A = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{m_1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & & & & & & & & 0 & \frac{1}{m_1} & & & & & & & & \\
0 & & 1 & & & & & & & \frac{r_{1y}}{J_1} & -\frac{r_{1x}}{J_1} & & & & & & & & \\
0 & & & 1 & & & & & & & & \frac{1}{m_2} & 0 & \frac{1}{m_2} & 0 & & & & \\
0 & & & & 1 & & & & & & & 0 & \frac{1}{m_2} & 0 & \frac{1}{m_2} & & & & \\
0 & & & & & 1 & & & & & & \frac{r_{2ay}}{J_2} & -\frac{r_{2ax}}{J_2} & \frac{r_{2by}}{J_2} & -\frac{r_{2bx}}{J_2} & & & & \\
0 & & & & & & 1 & & & & & & & & & \frac{1}{m_3} & 0 & \frac{1}{m_3} & 0 \\
0 & & & & & & & 1 & & & & & & & & 0 & \frac{1}{m_3} & 0 & \frac{1}{m_3} \\
0 & & & & & & & & 1 & & & & & & & \frac{r_{3ay}}{J_3} & -\frac{r_{3ax}}{J_3} & \frac{r_{3by}}{J_3} & -\frac{r_{3bx}}{J_3} \\
0 & & & & & & & & & 1 & & & & 1 & 0 & & & 1 & 0 \\
0 & & & & & & & & & & 1 & & & 0 & 1 & & & 0 & 1 \\
0 & & & & & & & & & & & 1 & & & & & & & \\
0 & & & & & & & & & & & & 1 & & & & & & \\
0 & & & & & & 1 & 0 & r_{3ay} & & & & & & & 1 & 0 & & \\
0 & & & & & & 0 & 1 & -r_{3ay} & & & & & & & 0 & 1 & & \\
-1 & 0 & & 1 & 0 & r_{2by} & & & & & & & & & & & & & \\
0 & -1 & & 0 & 1 & -r_{2bx} & & & & & & & & & & & & & \\
-1 & 0 & & & & & 1 & 0 & r_{3by} & & & & & & & & & & \\
0 & -1 & & & & & 0 & 1 & -r_{3bx} & & & & & & & & & & \\
\end{bmatrix} \qquad \text{Eq. 2.18}$$

$$b = [\dot{x}_1^- \quad \dot{y}_1^- \quad \omega_1^- \quad \dot{x}_2^- \quad \dot{y}_2^- \quad \omega_2^- \quad \dot{x}_3^- \quad \dot{y}_3^- \quad \omega_3^- \quad 0 \quad 0 \quad -Imp_x \quad -Imp_y \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0]^T \qquad \text{Eq. 2.19}$$

$$X = [\dot{x}_1^+ \quad \dot{y}_1^+ \quad \omega_1^+ \quad \dot{x}_2^+ \quad \dot{y}_2^+ \quad \omega_2^+ \quad \dot{x}_3^+ \quad \dot{y}_3^+ \quad \omega_3^+ \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots]^T \qquad \text{Eq. 2.20}$$

$$AX = b \qquad \text{Eq. 2.21}$$

Stochastic Terrain Model

At their core, many examples of walker-challenging terrain can be represented as a series of changes in terrain height, and as such, are modeled thusly in the stochastically varying terrain biped model. To provide proper application to later-described control methods, a discretized probability function of changes in ground-height-per-step is used to stochastically model terrain. The current terrain height is regenerated at the beginning

of each new step which means, in effect, the terrain height is constant for the duration of the stride, regardless of step size. The stochastic terrain model is visualized in Figure 2.4, illustrating that the terrain height changes are generated by a characteristic (Gaussian) probability function.

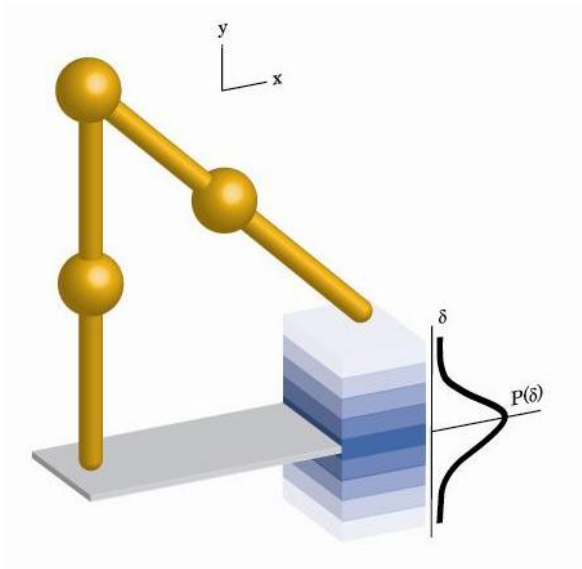

Figure 2.4: Illustration of the core concept of the stochastic terrain model: a ground height which varies in accordance to a given probability distribution

In the numerical experiments presented, a Gaussian distribution is selected to approximate a generically coarse surface with a “roughness” characterized by its standard deviation, as done in prior work by Byl (2009). It should be noted that the proposed methods in no way obligate a Gaussian probability distribution for terrain height as depicted in Figure 2.4. On the contrary, the versatility of this approach allows for discrete distribution functions which can be tailored to accommodate more specialized and exotic features (i.e., stairs, hurdles, or blocks).

This model is not designed to approximate any specific terrain instance in the manner of a predefined obstacle course, but instead, acts as a statistical representation of a given type of terrain. A representative approximation of stochastically generated terrain is pictured in Figure 2.5, which interpolates the terrain linearly between the resulting footholds.

Additionally, the stochastic terrain biped model has no memory of the absolute position of the walker, and hence cannot account for position-dependent terrain features. Efforts have been successful in characterizing terrain attributes in a manner which remains amenable to reinforcement-learning techniques yet are ill-approximated by a single probability distribution function (i.e., pits and chasms). While pits and chasms can be superficially modeled as a sizeable drop in height, attempts to navigate this feature would result in the inevitable failure of the walker. This inherent limitation is the product of the model's inability to represent position-dependent features, which renders the act of spanning the gap impossible. Byl (2009) has demonstrated the usefulness of a deterministic wrapping terrain model in its ability to represent intermittent terrain, which accommodates "no-go" regions that add further constraints to the walking controller. The addition of such a repeating terrain sample can extend terrain models in their applicability to practical scenarios, and consequently, their navigability via reinforcement-learning techniques.

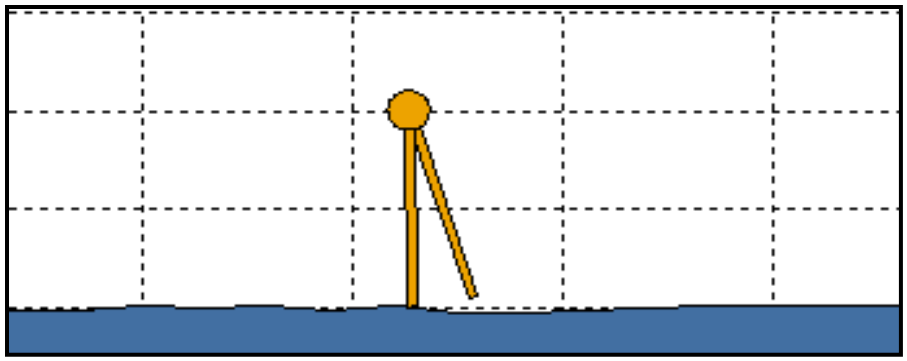

Figure 2.5: Frame of animation of walking sequence picturing a representative terrain roughness (Note: smoothness of terrain in animation is purely aesthetic)

Validation

The continuous dynamics were validated by comparison with similar models constructed in SimMechanics and ADAMS. With a sufficiently small simulation time step (10^{-6} seconds), the output variables for the continuous dynamics were identical to other simulations within 10^{-5} radians. Furthermore, the energy levels were continuously measured to ensure that energy remained conserved during unactuated motions. The full model (with collisions) was tested and shown that the model with no actuation would produce stable passive-dynamic walking on a downward slope. When equilibrium passive-dynamic walking was achieved, it was verified that the work done by gravity was equivalent to the energy lost in each plastic collision.

Actuation and Control

The two modes of actuation in the model are the pre-collision impulse (push-off) and the applied hip torque (forward kick). There are several established means of using these inputs to effectively control the compass gait which vary in complexity. The goal

of this thesis being to synthesize controllers with a wide performance range, it is necessary to utilize these actuation methods for their respective strengths in regard to energy economy, speed, and robustness. The following chapter outlines some traditional approaches for control in dynamic walking as well as a novel method proposed by this investigation.

Chapter 3: Genetically Optimized Gain-Scheduled Control

By nature, walking models have a number of features which render their control difficult for traditional methods. The continuous dynamics of the system are nonlinear, limiting their tractability with linear techniques. Linearization methods, while common tools for solving nonlinear control problems, require approximations (e.g., small angle assumptions) to be useful. Most problematically, the compass gait model is underactuated for the duration of the swing. The lack of direct actuation at the ankle joint surrenders all authority over the stance leg behavior to the momentum transfer of the swing leg controller and Newtonian dynamics. Despite these inherent complexities, relatively simple controllers have been shown to be effective in various experimental prototypes that are well-modeled by simple representations like the compass gait (Karssen 2007, Iida 2009)

Proportional-Derivative (PD) Control

Among the most basic of controllers, the proportional controller, also known as P control, commands a control effort proportional to the control "error". The control effort for mechanical systems is often a torque or force, but is always some form of variable input. The error (e) is defined as the numerical difference between a quantifiable system state, or system output, and the desired system state. The coefficient by which the control effort is proportional to the controller error is dubbed the controller gain (K_P). The commanded control effort, a hip torque (τ) in this application, forces the interleg angle (α)

to converge upon the desired interleg angle (α_{des}) using the control law shown in equation 3.1.

$$\tau = K_P e = K_P(\alpha_{des} - \alpha) \qquad \text{Eq. 3.1}$$

The standard proportional controller is often supplemented by adding further terms to the control law. One such addition is a derivative term which regulates the rate of change of the system output with respect to time. This requires the inclusion of an additional gain (K_D), dubbed the derivative gain. The control law for the Proportional-Derivative (PD) Controller is given in equation 3.2. The derivative term often serves to diminish oscillations and is often necessary to expedite convergence to the desired output. For this application, the desired time derivative of the interleg angle is always set to zero. This creates the functional equivalent of a mechanical damper which retards velocity. Also of note, the derivative controller can serve as a significant energy dissipater in a mechanical system.

$$\tau = K_P(\alpha_{des} - \alpha) + K_D(\dot{\alpha}_{des} - \dot{\alpha}) \qquad \text{Eq. 3.2}$$

The values for the proportional and derivative gains are paramount in tuning the behavior of the system. Generally speaking, heightened proportional gains can be implemented to tighten control of the system and decrease convergence time, but tend to require more energy consumption on the part of the actuators. Conversely, lower proportional gains tend to increase convergence time and alleviate the energy burden.

Tradeoffs emerge in selecting the derivative gain as well. Derivative gains are critical in minimizing "overshoot" and damping the system behavior. Serving as a dissipater, these values also have significant effect on energy consumption.

In application to dynamic walking, either high or low proportional or derivative gains could be advantageous depending upon the scenario. For a simplistic example, high proportional gains for the interleg controller could be desirable for comparatively rough terrain, which would assure each step has converged to the desired interleg angle before landing on an aberrantly tall surface (provided the derivative gain is sufficiently large to prevent grossly overshooting the desired leg angle). Applications demanding greater energy economy could sacrifice such high-fidelity stepping, using lowered proportional gains and reduced dissipative derivative gains to save on power consumption.

Advantageously, this control method has sufficient algorithmic simplicity that equivalent control can be achieved using mechanical springs and dampers (Wisse 2007). However, what this research seeks to find is a tradeoff-conducive controller. The generous computational resources available both on and off-board with current technology allow for a more thorough exploration of potential controllers which feature superior tradeoffs in robustness, energy economy, and speed.

Gain-Scheduled Control

A common approach to controlling nonlinear systems is gain-scheduling. In the aforementioned section, a PD controller was described as having a set of gains, one each

for the proportional and derivative terms respectively, which fully describe the behavior of the controller. In systems where different points in the system state space may have different responses to the control effort, it can become advantageous to apply different sets of gains. This approach is called gain scheduling. For the application at hand, the interleg angle is chosen as the key variable to be discretized for the purposes of gain scheduling, as illustrated in Figure 3.1.

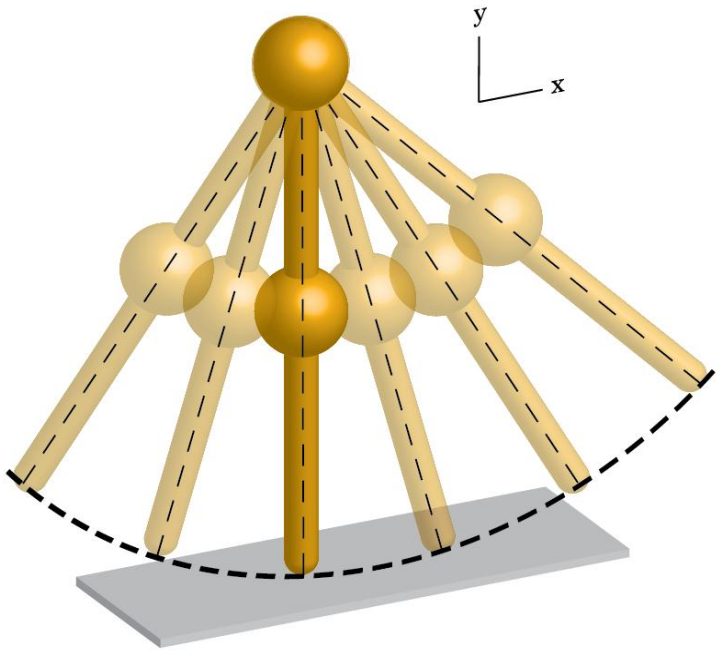

Figure 3.1: Compass gait model discretized by interleg angle for gain-scheduled control

The interleg angle was chosen as a target variable for gain-scheduled control for a number of reasons. The position of the swing leg in relation to the stance leg is an excellent indicator of the net torque on the ground pivot as a result of gravity (i.e., a swing leg held behind the stance leg will tend to cause a backward fall). Given the swing is largely unidirectional (neglecting small oscillations due to P-control near the desired

angle), the interleg angle correlated to the time elapsed during the swing. In addition, a common alternative to a pure PD controller in dynamic walking is the "activate at mid-stance" approach (Byl 2008), where the controller is turned on only after the swing leg passes the vertical. This is a rudimentary example of gain-scheduling by discretizing the leg angle into two regions, which results in greater energy economy and somewhat slower controller convergence time. A gain schedule with a higher resolution discretization has the opportunity to further improve upon this increased energy economy by refining the gain-schedule.

Lastly, the gain schedules' angle discretization is normalized with respect to the desired interleg angle (α_{des}) instead of absolute interleg angle. The normalization is a convenience that helps ensure that if the desired interleg angle is changed, the gain schedule will have this new target angle as a goal. In particular, normalization assures that gains which were tuned to control the leg when close to the target angle continue to apply close to the target even in the event that α_{des} is changed (these near-target gains in essence serve to hold the leg steady). Normalized angles are represented as the ratio of the interleg angle to α_{des}; -1.0 would represent an interleg angle of $-\alpha_{des}$ and 0.0 would indicate leg cross. The normalized angle range is divided into ten sectors in order to provide a relatively fine resolution. Eight of these sectors are evenly split between a relative angle of -1.0 and 1.0, with the two remaining sectors capturing every value outside of that range. This level of discretization was chosen as it was thought that eight intermediate sectors would provide sufficient resolution to examine a general shape of the profile.

Control Parameter Set

It is important to recall from the previous chapter that the hip torque is only one of two methods of actuating the compass gait model. The pre-collision impulse is a critical component of the walker's actuation. The impulse is a significant contributor to the system kinetic energy, and consequently, can be a significant drain on the actuator energy supply. As such, important tradeoffs are likely to be found in the variation of this impulse magnitude. This "push-off" control has only one parameter of variation, the magnitude of the applied impulse (which is always applied along the stance-leg direction, depicted in Figure 2.1). Given the potential prominence in its role for control, this scalar impulse magnitude is appended to the gain schedule as another parameter for adjustment. Including values other than simply controller gains, the gain schedule with the additional applied impulse magnitude is now more aptly dubbed the *control parameter set*.

Genetic Optimization

A genetic algorithm is a stochastically driven global search heuristic which seeks an optimal solution to a defined problem. Inspired by biological evolution, a genetic algorithm utilizes random variation, selection, and reproduction to search for an approximate, optimal solution by maximizing the desirability or solution fitness. A genetic algorithm requires three key components, a genetic representation of the solution domain, a fitness function, and a reproduction algorithm.

“Methinks it is like a weasel”

A classic example for the use of a genetic algorithm is the “weasel” program (Dawkins 1986). The task entails creating a program to generate a target 28 character string, starting from a series of 28 random characters. Using only random variation to edit the string, the program must produce the phrase “METHINKS IT IS LIKE A WEASEL”, a line from Shakespeare’s *Hamlet*. Intuitively, this scenario conjures comparisons to the thought experiment of monkeys randomly pounding on typewriters writing Shakespeare. The probability of such monkeys pounding on keyboards (random character generation) stumbling upon this particular Shakespeare quotation is vanishingly small ($\approx 1{:}10^{40}$). However, by supplementing this random variation with selective and reproductive algorithms, this stochastic approach becomes a powerful means of navigating enormous design spaces to find a workable solution.

The sequence of characters serves as a simple genetic representation, which is required for a genetic algorithm. Each character (analogously, a gene) can be randomly varied (mutated) independently from its neighboring characters. When a mutation occurs, the character is replaced with another randomly selected character from the alphabet. The string closest to the desired string (a metric of fitness) survives and reproduces, creating several offspring which undergo the same process. A sample output of the weasel program which uses a mutation probability (likelihood of any given character being replaced by a randomly selected character) of 5% and yields 25 children per generation is shown below.

```
Generation 000: ZKVQMKSONOLPKRRAHGWUMNQRMXTI
Generation 020: ZXVHMKKS DOWISZCFKK M WIMYEM
Generation 040: ZBTHJTKS DOWIS OFKE M WIZREM
Generation 060: MGTHYUKS NT IS LIKE A WERREZ
Generation 080: MOTHZCKS IT IS LIKE A WE MEG
Generation 100: MOTHITKS IT IS LIKE A WEAKEN
Generation 120: METHINKS IT IS LIKE A WEAKEY
Generation 140: METHINKS IT IS LIKE A WEACEL
Generation 160: METHINKS IT IS LIKE A WEACEL
Generation 168: METHINKS IT IS LIKE A WEASEL
```

The result is a fast convergence to the target phrase, which demonstrates the ability for algorithms with selection and random variation to rapidly traverse a vast set of possible solutions. While the above example is rather trivial, this approach to solving problems can be applied to the control parameter set to optimize the output of the controller.

Mutation

Mutations are the means of random variation in this genetic algorithm. When modifying the control parameter set, mutations are modeled as a random fluctuation of the numerical values following a Gaussian probability distribution. With a Gaussian model of variation, the magnitude of the standard deviation controls the rate of "genetic drift" due to mutations. The proportional gain schedule, derivative gain schedule, and applied impulse each have their own independent mutation rate (standard deviation of Gaussian noise). Mutations are calculated separately for each entry of the control parameter set, allowing each of the individual gains in the schedule (and the impulse magnitude) to drift independently.

Fitness Function

Analogous to an organism attempting to survive in its environment, a control parameter set attempts to "survive" by successfully controlling a robot step. As such, each child parameter set is tested using the compass gait simulation described in Chapter 2. The fitness function is designed to encourage the desired properties of the optimized control parameter set. In this investigation, robustness, energy and speed are of primary concern and form the basis of the fitness function. The fitness function (*F*), being both the conceptual and mathematical negative of undesirable cost (*C*), is formulated in equation 3.3 as a function of consumed energy (*E*), average speed of the step (*S*), a cost associated with the robustness of the controller (C_R), and a weighting factors to generate tradeoffs (w_E and w_S).

$$F = -C = -(C_R + C_E(E, w_E) + C_S(S, w_S)) \qquad \text{Eq. 3.3}$$

To retrieve the necessary energy consumption and speed values, the candidate control parameter set is tested by controlling a single step of the compass gait model. The model is initialized to a specific, narrow range of state space with a single preselected α_{des} value. The initial state variables are randomly generated within the bounds of this defined range of state space, which allows for a small range of disturbance rejection to be developed for the controller. It was found that the use of a larger area of the state space resulted in poor convergence of the algorithm. To elaborate, when the initial state variables were allowed to vary significantly each generation, the solution with the highest fitness varied too much each generation to determine if an optimal solution

was reached. To mitigate this problem, a very narrow range in state-space was used for the optimization.

The cost function for the energy-economy (C_E) is quite trivial and as shown in Eq. 3.4 is simply the product of the energy consumed (E) and its weighting factor (w_E).

$$C_E(E, w_E) = w_E E \qquad \text{Eq. 3.4}$$

The incentive for a speed optimizing controller is to increase speed. As a result, the speed cost function (C_S) is the product of the inverse of speed (S) and its corresponding weighting factor (w_S), as Eq. 3.5 illustrates. The role of weighting factors will be explained in greater detail later in the chapter.

$$C_S(S, w_S) = \frac{w_S}{S} \qquad \text{Eq. 3.5}$$

Robustness Cost Function

To facilitate robustness, a given control parameter set must "successfully complete" a step, or receive a significant penalty to its fitness. Successful step completion is defined, in this case, as the swing leg having reached the set interleg angle and zero interleg angular velocity within an assigned tolerance before the swing leg collides with the ground. This ensures not only that the walker remains upright, but avoids the premature termination of the step before reaching the desired step size.

One could easily envision this robustness cost function reducing to a simple Boolean operation which assigns a penalty if fallen. However, while such a binary view of success may be satisfactory for an evaluation of the end product, it can be important to

the genetic algorithm to be given an indicator of their "proximity" to success or failure. Envision trying to shoot a basketball free throw while blindfolded. When attempting to tune such a challenging shot, it would be useful to be told the direction that the shot is off-target, and preferably the magnitude of the error. By adding two more components to the cost function which indicate the proximity to a successful step, the genetic algorithm can be encouraged to move in the "right direction" when trapped in states of failure with otherwise little chance of escape. These two components are developed from an understanding of the two modes of step failure for the compass gait model: tripping forward and falling backward.

Failure Modes

Tripping forward occurs when the swing leg collides with the ground before taking an adequately large stride. This premature collision sends the walker falling head over heels. The indicator used to dissuade this failure mode is the "convergence height" (h_c), the height at which the leg controller converges on the desired angle (within specified tolerance). This height is calculated even if convergence is reached after colliding with the terrain by continuing the dynamics computations assuming the collision had never occurred. As the convergence height decreases, the walker is closer to (or perhaps deeper in) failure. To greatly discourage negative convergence heights, the "tripping forward" cost function (C_{tf}) is set to an exponential decay described mathematically in Eq. 3.6., If the convergence height is lower than the terrain height (δ), the step is considered a failure. Tripping forward often occurs when the hip torque

controller gains are too low or the applied impulse is too high. The values for the coefficients and exponents in Eq. 3.6 were selected by increasing their magnitudes until they were effective at preventing controllers which "trip" from surviving the algorithm's selection process.

$$C_{tf} = 5e^{-25(h_c-\delta)} \qquad \text{Eq. 3.6}$$

Falling backward, as the name implies, occurs when the stance leg forward velocity slows to the point where gravity pulls the walker backward. Insufficient applied impulse or excessively large hip controller proportional gains (due to the momentum exchange of a quick forward leg swing) will tend to result in falling backward. The indicator utilized for this failure mode is the maximal backward angular velocity of the stance leg ($v_{backward}$). By discouraging backward velocities via the exponential growth relationship between the "backward falling cost" (C_{fb}) and $v_{backward}$ in Eq. 3.7, the genetic algorithm favors controllers which maintain a satisfactory forward velocity. The coefficients and exponents in Eq. 3.7 were increased until they were effective at preventing controllers which fall backward from surviving the algorithm's selection process.

$$C_{fb} = 10e^{0.25\, v_{backward}} \qquad \text{Eq. 3.7}$$

The "tripping forward" and "falling backward" failure terms are finally supplemented by the simplest failure term indicating the presence of a failed step (C_{fall}) by the simple Boolean relationship shown in Eq. 3.8. These three failure terms are

summed via Eq. 3.9 into the final robustness cost C_R. The cost value for falling (500) was chosen to ensure that falling controllers would consistently result in inferior fitness to controllers with even extraordinarily high energy costs and low speed.

$$C_{fall} = \begin{cases} 500, & \text{if fallen} \\ 0, & \text{else} \end{cases} \qquad \text{Eq. 3.8}$$

$$C_R = C_{tf} + C_{fb} + C_{fall} \qquad \text{Eq. 3.9}$$

Energy-Speed Weighting Factor

When attempting to produce a tradeoff, it is essential to define a numerical factor which adds or reduces "weight" to the various metrics being traded off. In this application, the two candidate metrics for trade-off are the energy economy and speed of forward progress. While the goal of this research is to generate meaningful tradeoffs in robustness as well as energy and speed, a tradeoff in robustness for a single tested step is likely not meaningful in a system designed to take over hundreds or thousands of steps. It will be found later that much of long-term failures in walking result from uneven terrain forcing the robot into less viable future states. As such, a constant high penalty is assessed by this algorithm to any control parameter set which results in a failed step.

The energy and speed weighting factors (w_E and w_S respectively) are incorporated into the cost relationship as per Eq. 3.10 and are constrained such that they sum to a constant quantity (a value of 10, which is a magnitude large enough to encourage the desired energy speed tradeoff but small enough not to overwhelm the cost of falling) as expressed in Eq. 3.11.

$$C = C_R + w_E C_E + w_S C_S \qquad \text{Eq. 3.10}$$

$$w_E + w_S = 10 \qquad \text{Eq. 3.11}$$

Constraining the sum of the weighting factors creates an effective sliding scale between a cost function demanding energy economy versus high speed. It is expected that running the genetic algorithm with a variety of weighting factor pairs will produce a tradeoff curve with solutions sweeping from great energy economy to great speed.

Reproduction

The ability of favored solutions to pass down their traits through a form of heredity is foundational to genetic algorithms. For this investigation, only a single control parameter set survives from each generation to reproduce. The reproduction is "asexual" and does not utilize the genetic crossover sometimes used in genetic algorithms, meaning that all offspring of the sole surviving control parameter set are mutated copies of their parent. Each of the many offspring (50, which was chosen for computation speed because it resulted in convergence in fewer than 100 generations) is originally identical to the parent and then are modified using the mutation algorithm. As described in the previous mutation section, each numerical value is modified by adding the results of a scaled Gaussian random number generator. The Gaussian distribution having its peak centered at zero modification allows most of the values do be minimally affected by the mutation, but inevitably results in a few values making a large shift each generation. The mutation rates for the algorithm, as well as value bounds and algorithm

parameters, are given in table 3.1, indicating the standard deviation of the alteration made by the mutation each generation. The mutation parameters were chosen so the changes per generation were both large enough to reach convergence levels within 100 generations (saving computation time) and small enough so noise from the mutations would not obscure convergence.

Mutation Parameters			
	Pre-collision Impulse	Proportional Gains	Derivative Gains
Mutation Rate (standard deviation)	0.125	0.125	0.05
Initial Value	4	5	0.5
Minimum Value	0	0	0
Maximum Value	7	20	5
Reproduction Parameters			
Number of Surviving Parents for each Generation	1	Number of Offspring Produced for each Generation	50

Table 3.1: Parameters used in the genetic algorithm for mutating and reproducing the control parameter sets

Convergence

An optimal control parameter set is only reached once the algorithm is deemed converged, a state which should be specifically defined. The convergence is determined by observing values of the fitness function over several generations and assessing whether the values have become relatively constant. Numerically, the algorithm is deemed converged when the current generation's fitness value does not differ from any of its previous ten generations' fitness values by more than a given threshold (a numerical value of 1.0, a value approximately 1% of the total range of typical fitness values).

Figure 3.2 shows a typical pattern of convergence for the fitness values, including the point of convergence given the aforementioned criteria. The genetic algorithm generally converged in fewer than 80 generations.

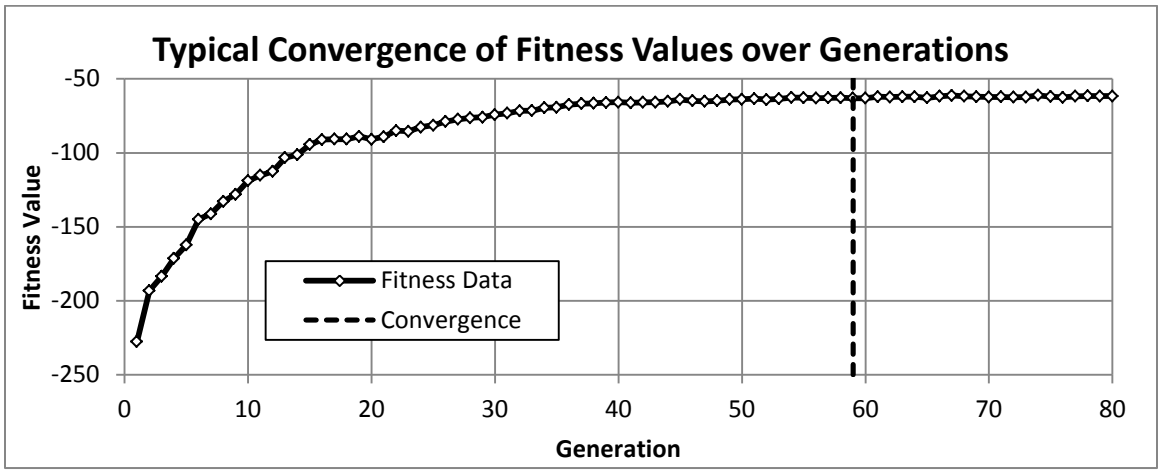

Figure 3.2: The surviving control parameter set's fitness value plotted over 80 generations, indicating that the convergence criterion is met at generation 59

Data Collection

For a given data set (which can be used to plot a single tradeoff curve), the genetic algorithm was run for a narrow region in state space (less than 1% of the total range), with a particular control action, but over a wide range of energy-speed weighting factors. Each run of the algorithm produced an optimized control parameter set which was then tested by simulating a step with 500 randomly generated starting states within the defined narrow state space region. The region in state space from which the initial simulation states are selected is outlined in Table 3.2, which lists the upper and lower bounds for each of the state variables, as well as the desired interleg angle and terrain height. The initial state variable ranges chosen for this data set were selected because

they did not require extraordinarily large impulses or gains in order to avoid falling, meaning it is a reasonably viable range of states.

Genetic Algorithm State Space Range Parameters			
State Variable	Units	Minimum Value	Maximum Value
X1: vertical leg separation	(m)	0.00	0.00
X2: horizontal leg separation	(m)	0.449	0.451
X3: stance leg angular velocity	(°/sec)	-61.0	-59.0
X4: swing interleg angular velocity	(°/sec)	-1.00	1.00
α_{des}: desired interleg angle	(°)	25.0	25.0
δ: terrain height	(m)	0.00	0.00

Table 3.2: The maximum and minimum values denoting the range of state space used to generate the reported data with the genetic algorithm

The resulting 500 simulation runs assure that the generated control parameter set will not fail to take a step within that state space range. In addition, the large number of test runs (perhaps excessively large given the limited breadth of the state range) provides a more solid statistical basis for assessing the energy and speed. The mean values of the speed and energy consumed for the step taken are recorded in addition to the median and standard deviation. The standard deviation was universally found to be two orders of magnitude smaller than the mean, so variation in this figure was considered insignificant.

The energy consumed was further processed into the more generally applicable metric of *specific cost of transport* (SCT) which is the non-dimensional quantity of energy consumed per unit weight per unit distance traveled. This resulting data pair consisting of the single-step speed and specific cost of transport of this control parameter set forms a single point in energy-speed tradeoff space.

A variety of points can be generated in tradeoff space by two means. A wide range is primarily achieved by modifying the energy-speed weighting factors, which is intended to influence, if not completely control, the resulting point's position in tradeoff space. Secondarily, simply running the algorithm repeatedly can yield somewhat differing results due to the inherently stochastic nature of the genetic algorithm. Both methods were used in producing the presented data.

From the outset of data collection, the ratios of weighting factors necessary to produce a wide tradeoff curve were not intuitively clear. The weighting ratio selection was continually assessed throughout the data collection process as more was learned about the relationship between the weighting factors and the resulting position of the points in tradeoff space. No points were omitted in reporting in order to avoid selection bias.

The basic procedure in selecting weighting factors sought to first find the extremes of the tradeoff curve by amplifying the discrepancy between the weighting factors. The weighting of energy economy was increased until the resulting points produced no greater advantage in reduced energy consumption (data which essentially duplicated the results of less extreme weighting ratios). Conversely, attempts to find an upper boundary on step speed were met with the realization that walker speed was only limited by saturation of the actuators. After arbitrarily deciding that 1.25 m/s was a sufficient upper bound on speed for the purposes of this investigation, it was found that intermediate results were easily generated by incrementally adjusting the weighting factors from one extreme to the other. The correspondence between increasing weighting

factors and the change in position on energy-speed axes indicates that the fitness function is appropriate for easily generating tradeoffs.

Genetic Algorithm Results

For this single slice of state space (as defined in Table 3.2), 55 data points were collected using weighting factors ranging from 10:1 to 1:43 ratios of energy economy to speed. All of these data points represent control parameter sets which never failed during 500 random test runs within the narrow scope of their state-space tuning. Figure 3.3 shows each of these points plotted on energy-speed tradeoff space. The plot shows a clear optimal performance frontier which is well fit by a quadratic regression (R^2=0.9924). As can be seen in Figure 3.3, the minimum energy cost found for this commanded step is equal to a value of the specific cost of transport of approximately 0.3, which corresponded to a minimum step speed of approximated 0.33 m/s. The opposite extreme corresponded to a speed of 1.25 m/s and a specific cost of transport of approximately 1.5.

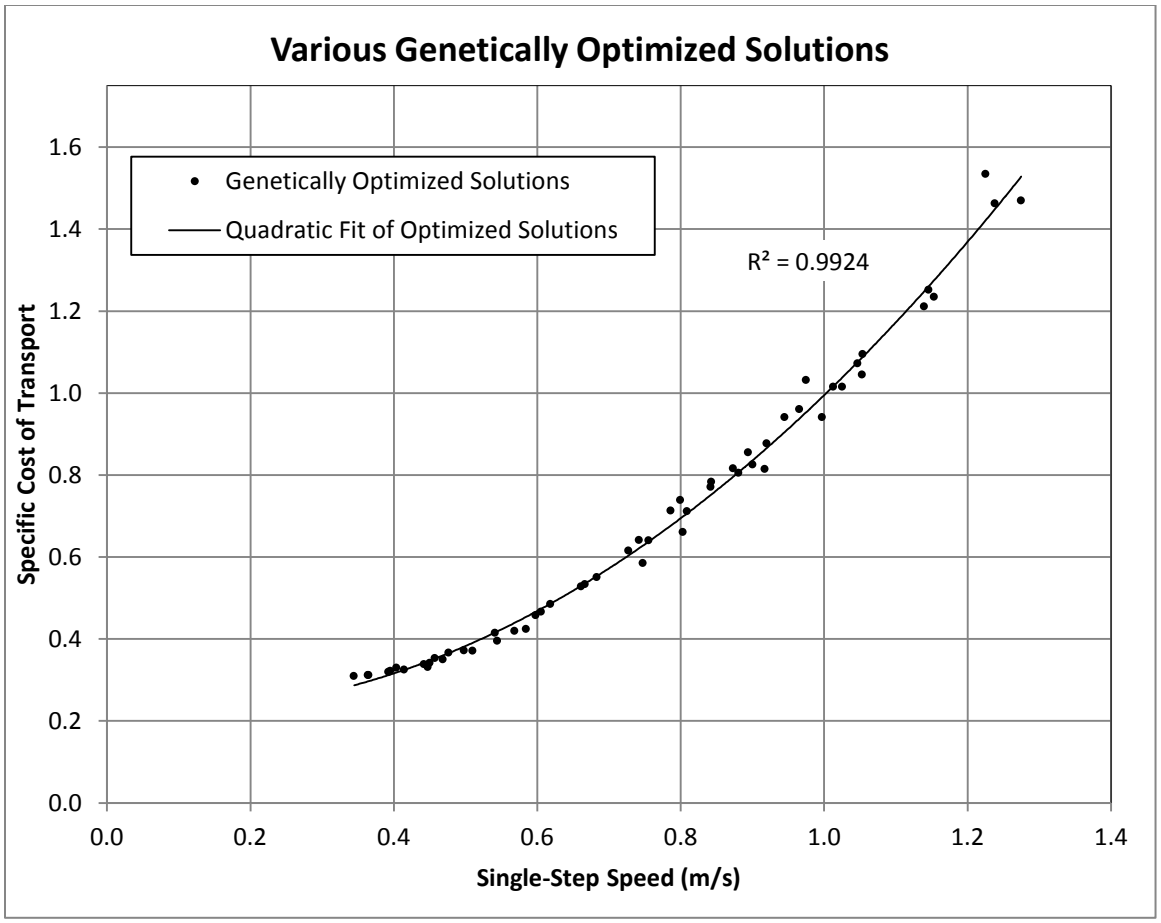

Figure 3.3: 55 solutions generated by the genetic algorithm (one data set) plotted in energy-speed tradeoff space with a quadratic data fit

Conclusions

The devised genetic algorithm, when varied in weighting ratios, produced a clear optimal performance frontier with a strong quadratic nature. This quadratic relationship between energy consumption and speed is in line with well-established principles of mechanics which relate the kinetic energy of a system to the square of its velocity. The actuator work must provide the requisite kinetic energy (which has an inherent quadratic

relationship to speed) to propel the system at the resulting speed, an indication of the tradeoff curve's significant quadratic relationship.

As currently devised, the genetic algorithm is impractical for implementing tradeoff-conducive control for a walking robot. Each of these data points required executing the genetic algorithm to convergence, a process which typically needed 20 minutes of computing time. Furthermore, each collected point represent only a single point on a tradeoff curve within one small slice of the overall robot state space, rendering such an approach so exhaustive that it is computationally intractable. To be sufficiently effective as a tradeoff-conducive controller, a more generalized or efficient means of producing tradeoffs must be developed.

Chapter 4: Heuristic Control

While the term "heuristic" has many definitions depending on the context of its use, in a broad sense, it refers to a process or rule which is generally successful, but has not been demonstrated to be universally effective. Often regarded as guidelines or "rules of thumb", heuristics are typically used when robust, generally applicable solutions are inconvenient or unavailable. In the face of an inconvenience in the form of computational intractability, developing a heuristic is an attractive alternative for the generation of controllers capable of tradeoffs over a range of performance.

Optimization-Inspired Heuristic

Despite a genetic algorithm being an unwieldy tool for generating a controller for every possible action, the results generated by such an algorithm can be analyzed to find patterns or common features in the results. An obvious route involves looking at the control parameter sets produced by the genetic algorithm and plotting trends in the parameter values against the controller outcome measures, in this case energy and speed. If such a clear trend exists, then the control parameters could be approximated and fitted functions could be used to quickly synthesize a controller capable of producing effective tradeoffs. Figure 4.1 plots the impulse magnitude obtained by the genetic algorithm against its corresponding resulting speed. The pre-collision impulse magnitude follows a strongly linear trend ($R^2 = 0.9736$) over the entire range of possible speeds. This stands in contrast to Figures 4.2 and 4.3 which show the proportional and derivative gain schedules respectively plotted against the resulting step speed.

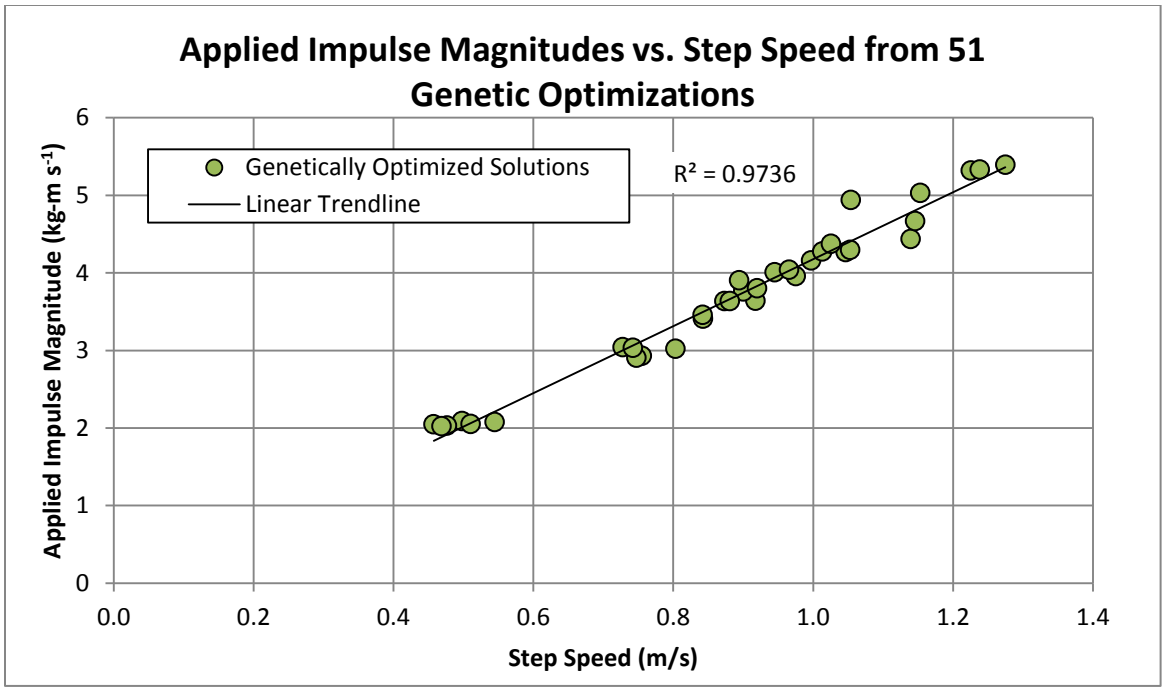

Figure 4.1: All impulse magnitude values for 51 genetic optimizations plotted with a linear trend line, revealing a strong linear correlation

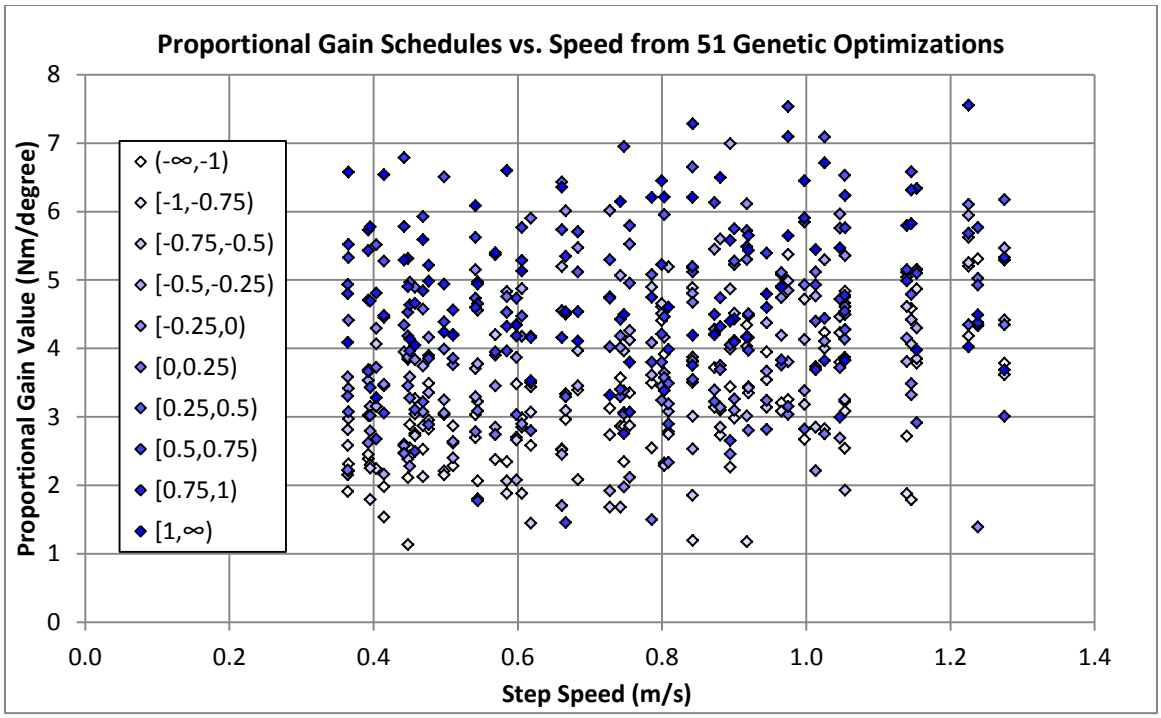

Figure 4.2: All proportional gain schedule values for 51 genetic optimizations, revealing no obvious trend

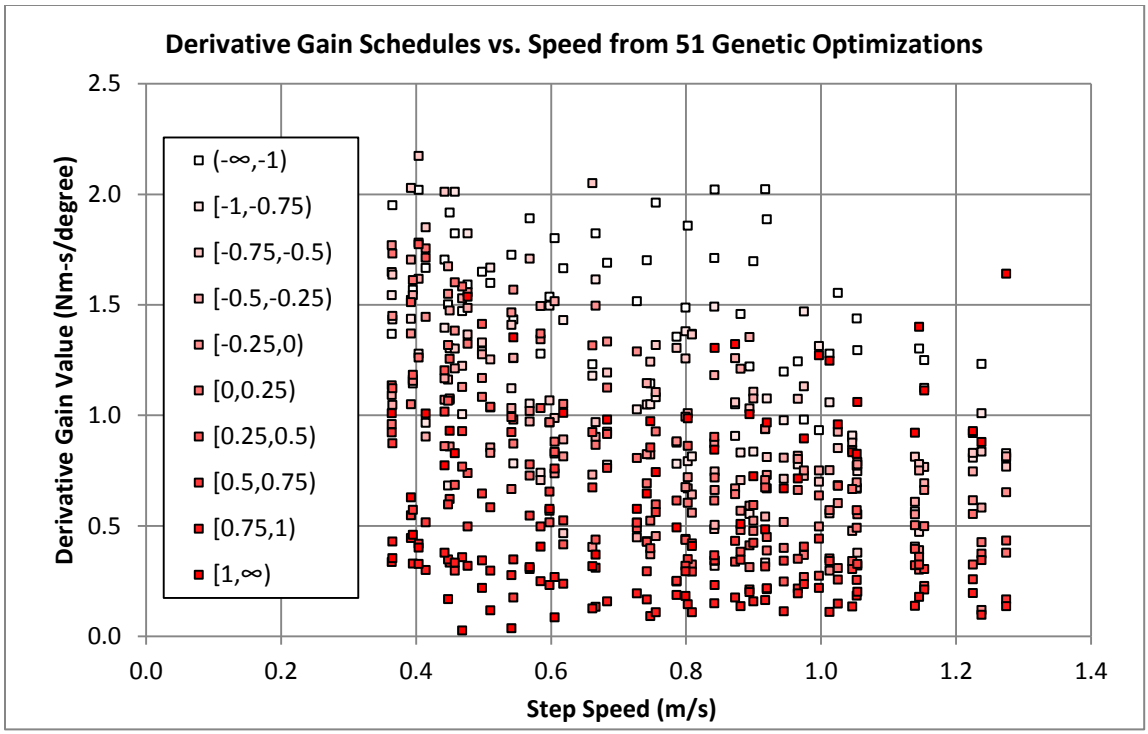

Figure 4.3: All derivative gain schedule values for 51 genetic optimizations, revealing no obvious trend

There are numerous reasons to suspect that the pre-collision impulse has the largest influence on the dynamics of each step. It has been shown to be a highly effective means of imparting kinetic energy to the forward motion of a walker (Kuo 2002) and is likely a significant source of energy expenditure in any genetically optimized walker controller. As such, the pre-collision impulse is subject to significant selection pressures from the genetic algorithm and dissuades random drift in the applied impulse via the selection process. Such obvious trends not being present in the gain schedules, a more quantitative means of detecting the importance of parameters is needed.

Selection Pressures

A selection pressure (sometimes called an evolutionary pressure) is an incentive or disincentive induced by the selection procedure of an evolutionary process which acts on specific traits. For example, an organism which relies heavily on its ability to outrun predators may have a strong selection pressure on its running speed. As a result, the pressure will tend to produce subsequent generations in which high running speed is enhanced or conserved (i.e., protected from degradation). Traits which are largely unrelated to the organism's survival have low selection pressures, and will tend to "drift" due to aggregate variation. These selection pressures play a tangible role in the interpretation of the results of the genetic algorithm. By examining the variation over time (generations) in the control parameters (analogously, the organism traits), the qualitative strength of the selection pressures can be hypothesized by inference. Identifying parameters which are largely conserved after fitness convergence, meaning they experience a lack of drift that would otherwise be associated with random mutations, suggests that such parameters could be critical to the success of the controller.

Random Walk

A series of random changes in a variable as a result of the application of (but not limited to) genetic algorithms is called a random walk. A series of random mutations as described in the genetic algorithm (a normalized random variation) can be similarly considered a random walk phenomenon. When observed over time, these random walks have a distinct statistical behavior, notably an increasing variance over time. A simple

statistical analysis of 5000 runs of a random walk using a normalized random change is shown in Figure 4.4. It depicts the percentile ranking of the values of the random walks over time, demonstrating a "fanning out" of the variation over time. However, if acted upon by an outside force, such as a selection pressure, one would expect the fluctuations in parameters to deviate significantly from this random walk distribution.

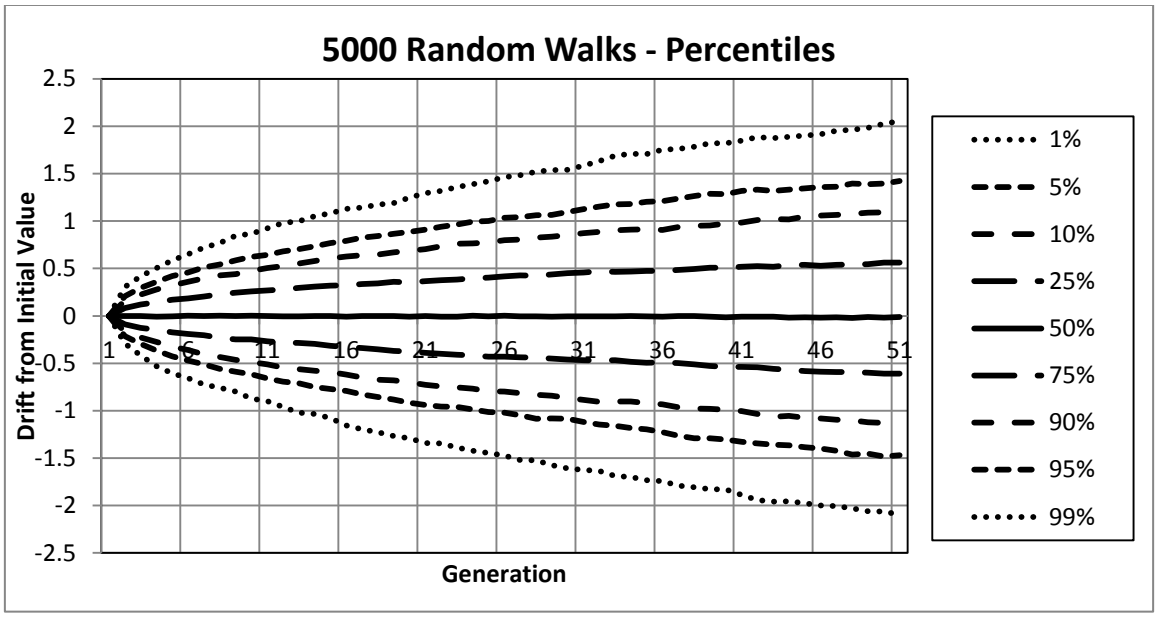

Figure 4.4: The percentile values of 5000 normal ($\sigma = 0.125$) random walks over time (50% indicating the median, 75% denoting the third quartile, etc.)

While a rigorous statistical analysis would be able to detect the probability of a particular variation being explainable by a random walk, a quicker and more simplistic analysis was used to qualitatively assess which parameters have a strong effect on the controller fitness. Two metrics were employed to find deviations from a random walk: rapid changes which were too fast to occur by an unguided random walk and values which were implausibly stagnant if subjected to random variation.

Impulse Selection Pressures

The progression of the pre-collision impulse magnitude while being optimized is displayed for a sample run of the genetic algorithm in Figure 4.5, which shows a fairly typical qualitative convergence behavior. The initial 15 generations appear to plummet followed by a slow drift to convergence (note that the convergence threshold "goal-line" is determined by controller fitness (defined in Chapter 3) and not convergence of the impulse magnitude). While such a qualitative assessment can be useful, Figure 4.6 helps quantify the drastic nature of the drift by overlaying the change in the impulse magnitude with the percentile values predicted by chance.

For the beginning 15 generations, the impulse drifts so fast compared to the result of 5000 random walks of equivalent mutation rate that it surpasses the 99th percentile values. This renders the pre-convergence behavior of the impulse magnitude highly improbable if attributed entirely to a random walk. The suggestion of this result is that lower pre-collision impulse magnitudes were favored by the selection algorithm, which resulted in a rapid reduction of the impulse magnitude.

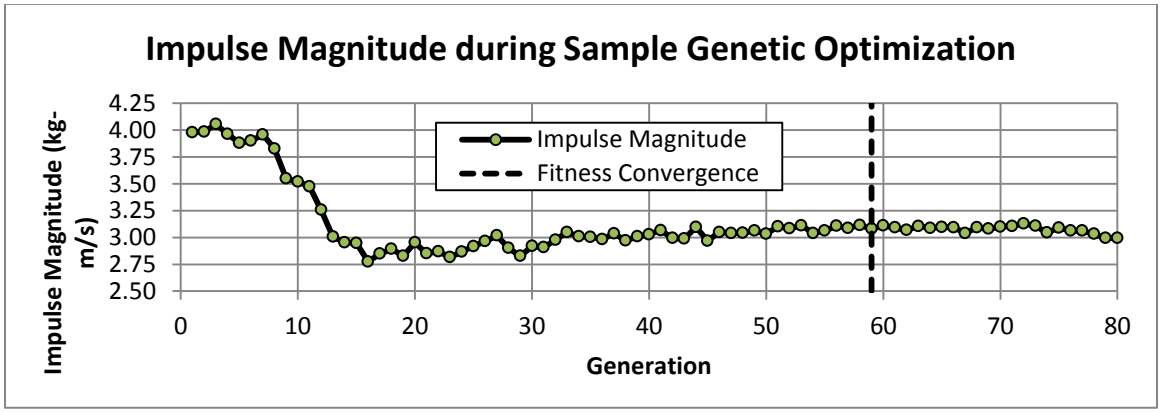

Figure 4.5: A sample genetic optimization following the change in impulse magnitude over 80 generations.

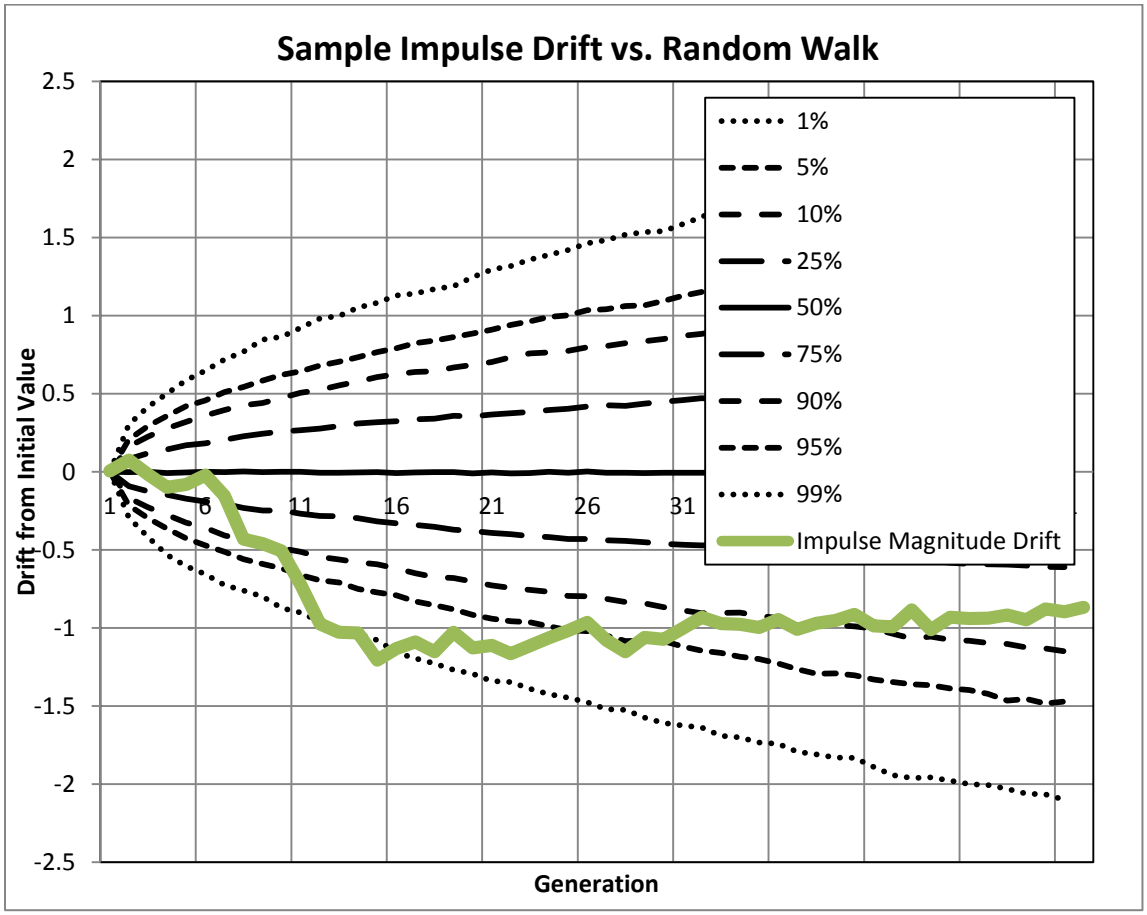

Figure 4.6: Plot of impulse magnitude drift due to the genetic algorithm against the random walk probability profiles (i.e., at generation 15, over 99% of random walks produced drift numbers greater than the impulse magnitude drift at that time, meaning that less than 1% of random walks produced such extreme values)

Furthermore, the impulse magnitude was also observed after fitness convergence was reached. Figure 4.5 shows this post-fitness convergence behavior which is remarkably stagnant. Once fitness convergence is reached, the impulse magnitude never deviated from the value at convergence by more than 0.08 kg-m/s (out of approximately 1.0 kg-m/s) for the 21 generations recorded after convergence. The 5000 random walks

of the same mutation rate were assessed to determine the probability of such a stagnant parameter value emerging by chance. After 12 generations, every single random walk had deviated from its initial value by more than 0.08 at some point, far short of the 21 generations for which the impulse magnitude remained within that window. Figure 4.7 plots the number of random walks which remain within this threshold over a number of generations, showing how quickly this level of preservation becomes an unlikely phenomenon for random walks. This implausible behavior adds further to the body of evidence that the magnitude of the pre-collision impulse was subjected to a strong selective pressure in the genetic algorithm.

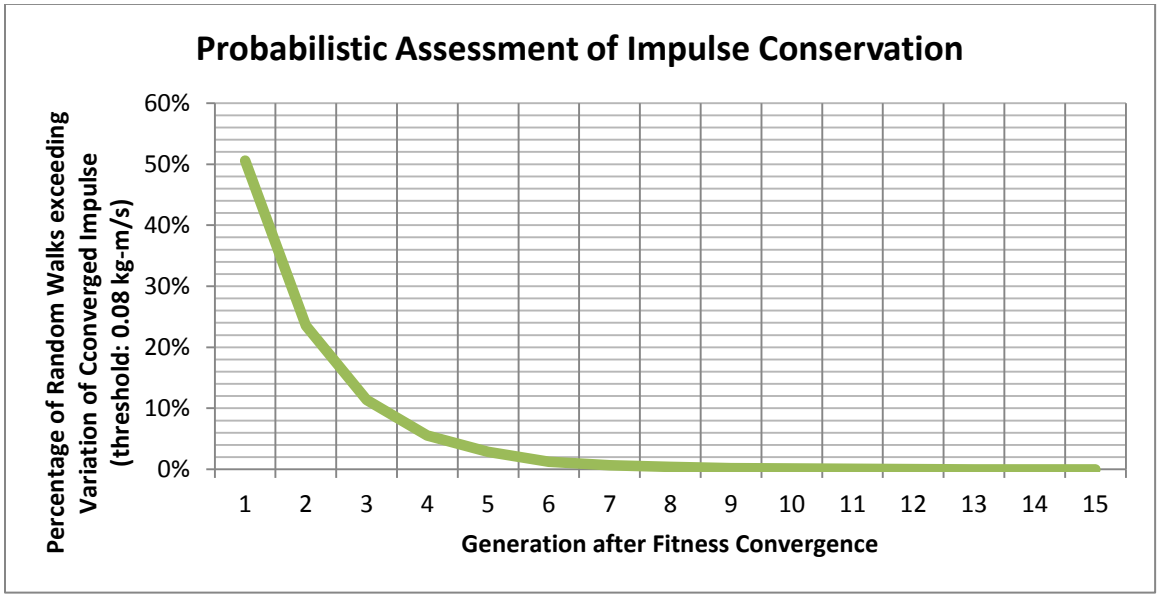

Figure 4.7: A statistical analysis of 5000 random walks with σ = 0.125 (identical to impulse mutation rate), observing the percentage of random walks which remained within 0.08 of their starting value over several generations

Gain Schedule Selection Pressures

In contrast to the pre-collision impulse magnitude, the proportional and derivative gains do not change as rapidly or converge as clearly. Figures 4.8 and 4.9 show the change in the proportional and derivative gain schedules (respectively) over the course of 80 generations. This sample run of the genetic algorithm is the same sample used in the impulse analysis. While it appears that various gains begin to fan out, it is not clear whether the values ever converge after the fitness is achieved convergence (the convergence criterion is outlined in Chapter 3).

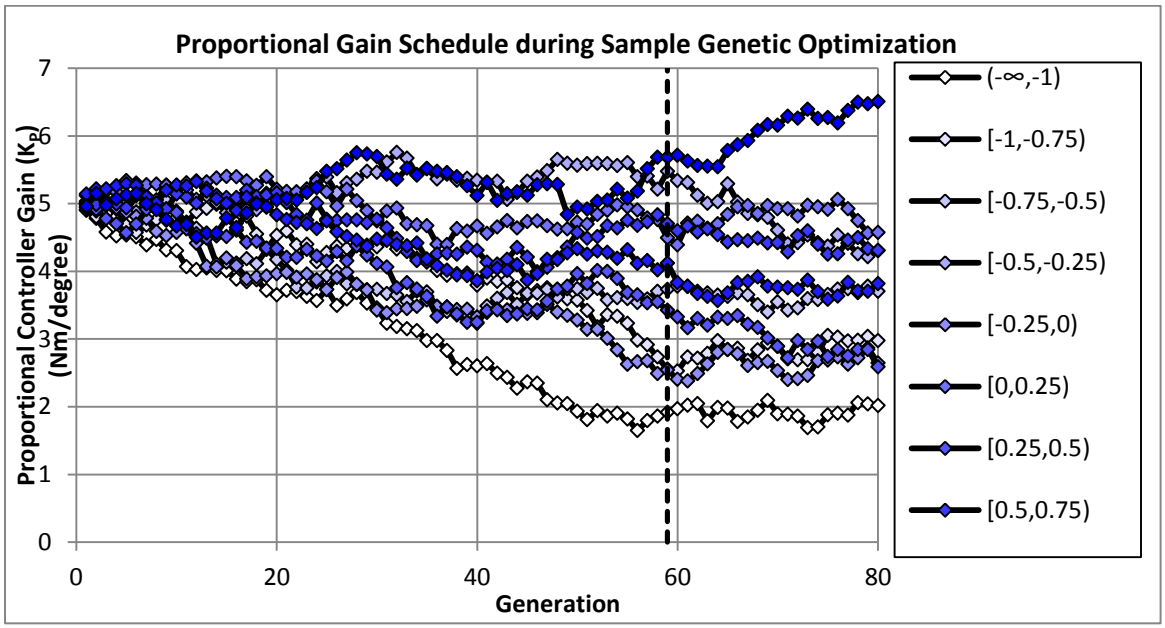

Figure 4.8: A sample genetic optimization following the change in the proportional gains in the gain schedule over 80 generations

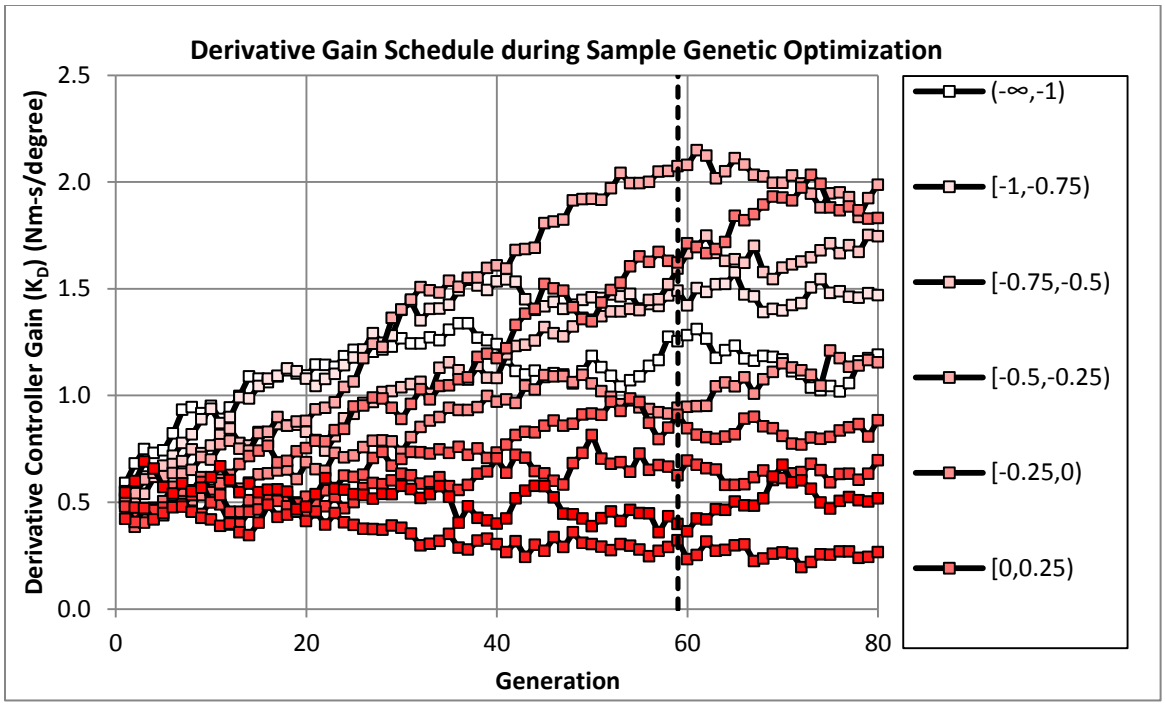

Figure 4.9: A sample genetic optimization following the change in the derivative gains in the gain schedule over 80 generations

The case for strong selection pressures in the proportional and derivative gain schedules is considerably weaker than that for the impulse magnitude. Figure 4.10 plots the drift of the proportional gain schedules after fitness convergence against the percentile ranges of random walks. Two of the gains breach the 1% values, indicating a likely pressure continuing to act after fitness convergence. However, the remaining gains vary significantly enough (qualitatively) that it appears their values are not being preserved by selection, but not to such an extreme that a random walk would be an improbable explanation.

The same is generally true of the derivative gains. Figure 4.11 again shows that only two gains convincingly vary (outside of the 99th percentile). While the null

hypothesis (variation completely explained by random walk) cannot be rejected for many of these gains, the possibility remains that the proportional gains are interdependent or are the results of many redundant solutions being found (redundant in the sense that the performance is similar despite the gain schedule being different). For example, two gains adjacent in their interleg angle discretization may make similar contributions to the forward motion of the swing leg, with the exact order of the gains not being particularly significant.

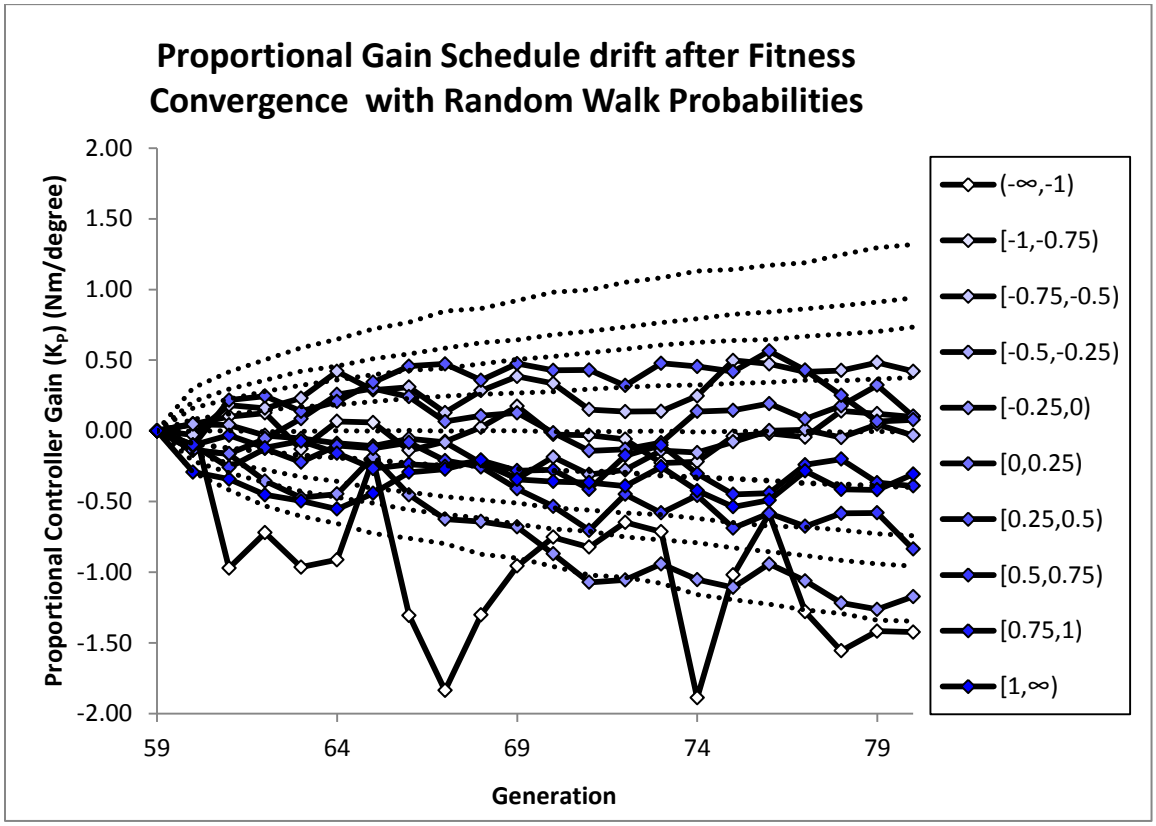

Figure 4.10: Beginning at the generation of convergence, the drift in proportional gain is plotted against the percentile ranges of random walks (i.e., the 50% line indicates the median value, 75% is the third quartile value). The dotted lines from bottom to top are the following percentages: 1%, 5%, 10%, 25%, 50%, 75%, 90%, 95%, and 99%.

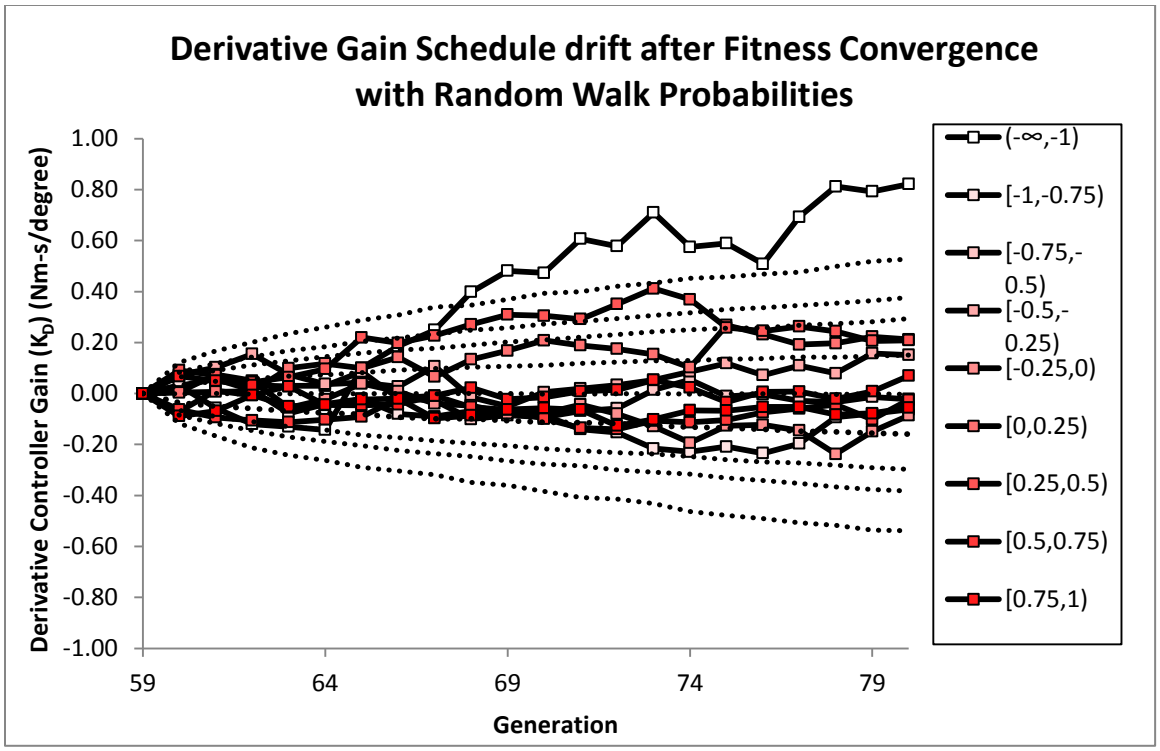

Figure 4.11: Beginning at the generation of convergence, the drift in derivative gain is plotted against the percentile ranges of random walks (i.e., the 50% line indicates the median value, 75% is the third quartile value). The dotted lines from bottom to top are the following percentages: 1%, 5%, 10%, 25%, 50%, 75%, 90%, 95%, and 99%.

Mean Gain Schedule

The behavior of the controller gains when subjected to a genetic algorithm indicates some effect on performance, but it is not as obvious an effect as is present for the pre-collision impulse. Furthermore, the existence of redundant or interdependent solutions has yet to be explored. Without a detailed, multivariate analysis of each of the components of the gain schedule, it is difficult to assess the exact nature of the interactions between the gains. However, operating under the hypothesis that redundant solutions exist for the gain schedule, a representative solution can be used to develop an effective controller. If a representative gain schedule can be used to produce a tradeoff

curve similar to the genetic optimizations, it would be strong evidence of the redundancy of solutions for gain profiles.

Such a representative gain schedule profile (gain profile) was produced by taking the mean values for each of the ten individual gains over all of the 51 optimized gain schedules. This mean "ramped" profile is shown in Figure 4.12, which is named for the inclined shape of the profile with the proportional and derivative gains increasing and decreasing respectively as the interleg angle approaches the target angle (the swing angle ratio approaches one). This ramped profile is used as the basic gain profile, proportional ($\overline{K_{P_{base}}}$) and derivative ($\overline{K_{D_{base}}}$), for a control heuristic.

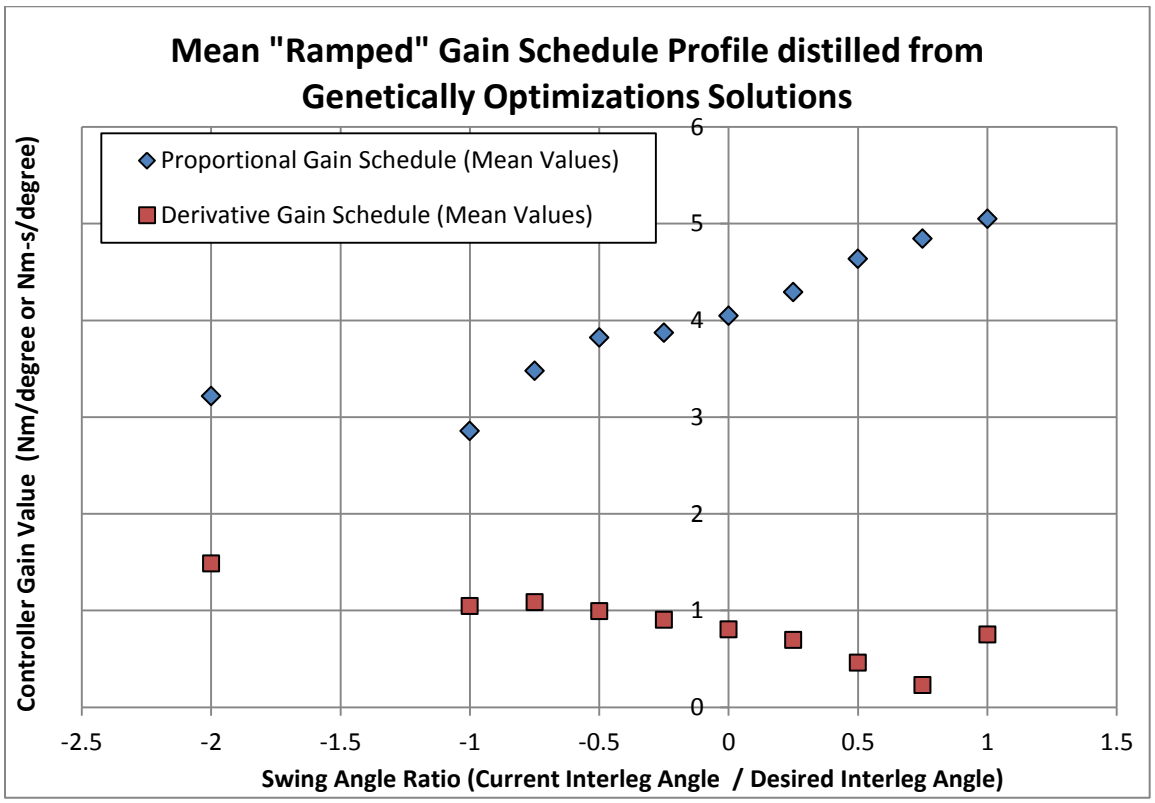

Figure 4.12: Mean values for 51 optimized gain schedules produced by the genetic algorithm for various weighting factors. Each point indicates a gain associated with a lower-bound swing angle ratio range in the gain schedule.

Tradeoff-Conducive Control Heuristic

Using the strong linear relationship between required speed and applied impulse as well the ramped gain profile synthesized from averaging 51 optimization-generated profiles, the components are now in place to produce a heuristic capable of generating efficient tradeoffs for step control, henceforth called a *tradeoff-conducive control heuristic*. The highly linear speed-impulse relationship is used as a starting point for adjusting the controller to accommodate faster versus energy efficient steps. As the demand for step speed increases, the heuristic controller scales the applied impulse linearly to match the increased speed requested.

Unlike the impulse magnitude, it is less obvious how the heuristic should handle any adjustment to the gain profile in response to varying demands for tradeoffs. Some less-definitive insights can be deduced from the genetic optimization data by plotting individual gain values against their resulting speed. Figures 4.13 and 4.14 show some representative proportional and derivative gain schedule values respectively plotted against the controller's resulting speed. The linear trend lines produced are often positive in slope for proportional gains, and negative in slope for derivative gains. However, the R^2 values for some sample proportional and derivative gains are quite low (0.3171 and 0.4075 respectively) when compared to the impulse trends (0.9736). Despite the less convincing nature of these trends, linear scaling was also used to scale the magnitudes of the proportional and derivative gain profiles with respect to speed demand.

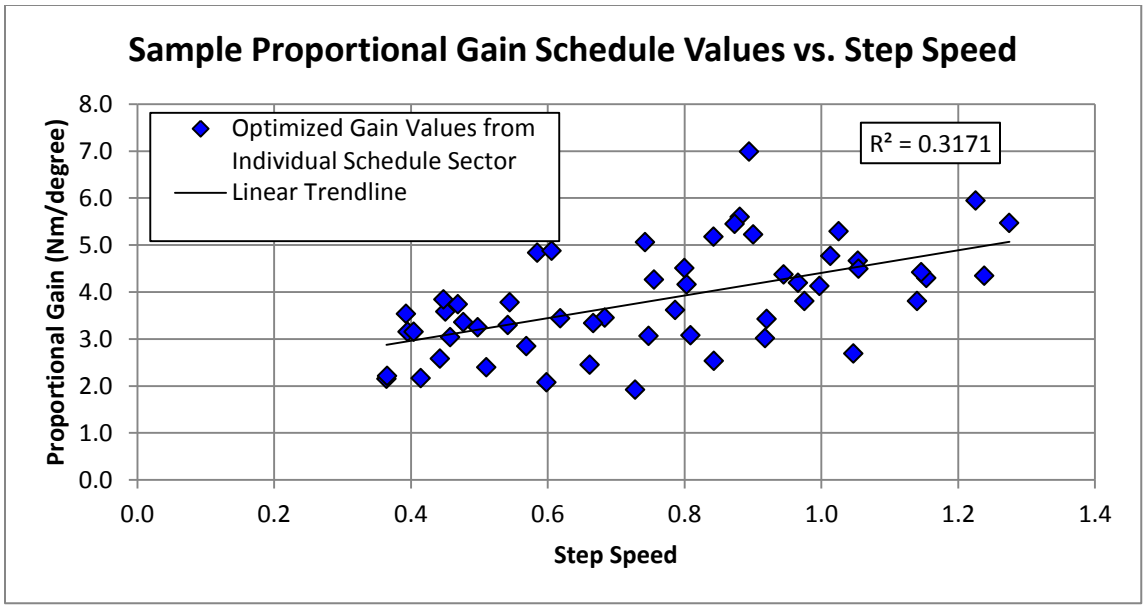

Figure 4.13: Several genetically optimized values (all 51 genetic optimizations) of a single, sample sector (-1.0 to -0.75 normalized interleg angle) of the proportional gain schedule plotted against the resulting controller speed (essentially a single gain schedule entry from Figure 4.2)

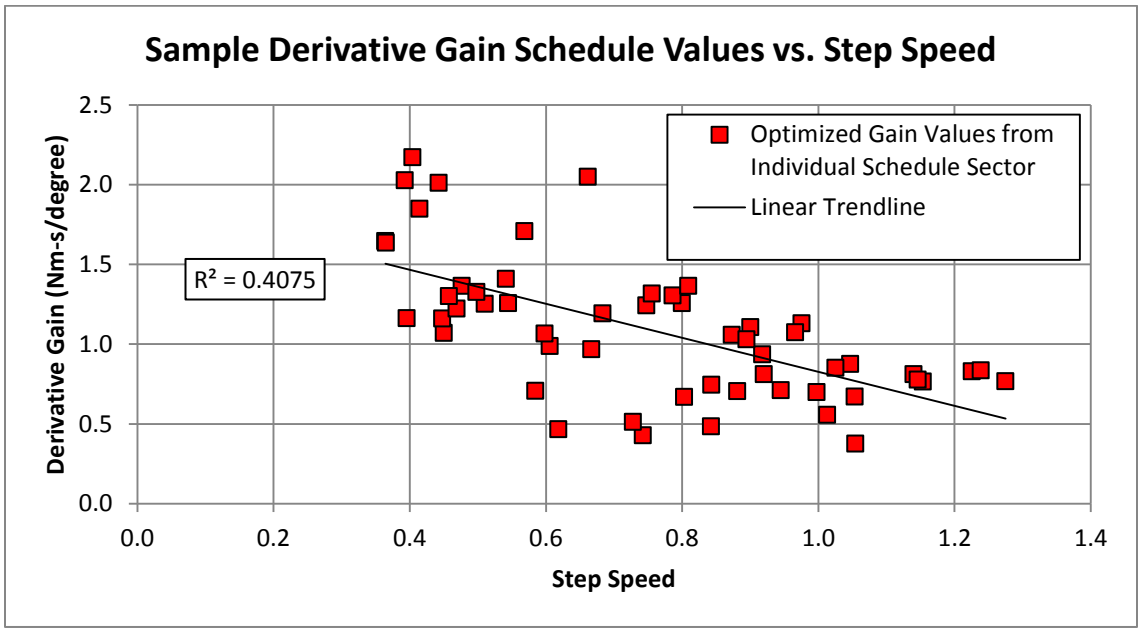

Figure 4.14: Several genetically optimized values (all 51 genetic optimizations) of a single, sample sector (-1.0 to -0.75 normalized interleg angle) of the derivative gain schedule plotted against the resulting controller speed (essentially a single gain schedule entry from Figure 4.3)

Heuristic Bounding Parameters

With the heuristic controller dependent upon a linear relationship between the tradeoff and the parameters, the bounds for the linear scale are imperative in determining the breadth of the tradeoff as well as the slope at which the parameters are scaled. There are two bounds used for scaling each of the applied impulse, the proportional gain schedule, and the derivative gain schedule, which yields six bounding parameters. These six heuristic bounding parameters fully describe the tradeoff controller in that they produce a controller yielding every performance demand between maximal energy-efficiency and highest speed via linear interpolation.

Given the six heuristic bounding parameters, controllers are generated by specifying the energy-speed weighting factor (w_{ES_h}). Similar to the weighting factor used for the genetic algorithm, this value sets the desired operating point on the spectrum between an energy efficient controller and a fast one. The value is set between zero and one, with zero producing the most energy efficient controller and one producing the fastest step. The equations for the interpolation and scaling of the control parameters used in the control parameter set, applied impulse (I_{app}), proportional gain profile ($\overline{K_P}$), and derivative profile ($\overline{K_D}$), are described in Eq. 4.1-4.3 as functions of the two impulse bounds (b_{I_1} and b_{I_2}), proportional profile scaling bounds (b_{P_1} and b_{P_2}), and derivative profile scaling bounds (b_{D_1} and b_{D_2}). Again, the greater of the two values need not be the first, as such an arrangement would indicate a decreasing scaling factor with increasing weight to step speed.

$$I_{app} = w_{ES_h}\left(b_{I_2} - b_{I_1}\right) + b_{I_1} \qquad \text{Eq. 4.1}$$

$$\overline{K_P} = \overline{K_{P_{base}}}\left[w_{ES_h}\left(b_{P_2} - b_{P_1}\right) + b_{P_1}\right] \qquad \text{Eq. 4.2}$$

$$\overline{K_D} = \overline{K_{D_{base}}}\left[w_{ES_h}\left(b_{D_2} - b_{D_1}\right) + b_{D_1}\right] \qquad \text{Eq. 4.3}$$

Lingering questions remain regarding how these six heuristic bounding parameters are selected. Superficially, it appears as though this attempt at creating a less complex heuristic has simply substituted one parameter optimization problem (the genetic algorithm) for another (the tuning of the heuristic bounding parameters). However, this heuristic can result in two significant advantages over the genetic algorithm. First, the results of a single run of the genetic algorithm to tune the control parameter set produce a single point on the tradeoff curve, while a tuned set of heuristic bounding parameters generates the entire energy-speed tradeoff curve and its corresponding set of controllers. Secondly, the heuristic controller is governed by a mere six parameters, as opposed to the 21 which define each control parameter set on the tradeoff curve. This marked decrease in the number of parameters benefits the computational tractability of the problem.

Heuristic Parameter Tuning

To have an efficient and objective means of tuning the heuristic bounding parameters, an automatic “heuristic parameter tuner” was developed to find an optimal set. Unlike the genetic algorithm which “tunes” the 21 variable control parameter set, this heuristic tuning algorithm needs to find a solution in a search space of only six

variables. As such, the smaller computational burden allows for more deterministic algorithms to be used (as opposed to stochastically-driven techniques like the genetic algorithm). This heuristic parameter tuner utilizes a primitive gradient-descent algorithm to navigate toward an optimal set of parameters.

Gradient-Descent Algorithm

Gradient descent algorithms operate by starting with a guessed solution and computing the gradient of the performance function at that point. This gradient, essentially being the "slope" of the performance when plotted against the dependent variables (the heuristic parameters), indicates the direction in which the performance increases to the greatest degree (or decreases undesirable qualities to the greatest degree). After determining the direction of the steepest gradient, the guessed solution is updated by "moving" in that direction. Often, the magnitude of this move is adjusted in proportion to the slope magnitude, but this feature was omitted to facilitate algorithmic simplicity. Figure 4.15 visualizes the gradient-descent process on a contour plot as a navigation from initial guess x_0 to the minimum value at the center, following the path of greatest descent.

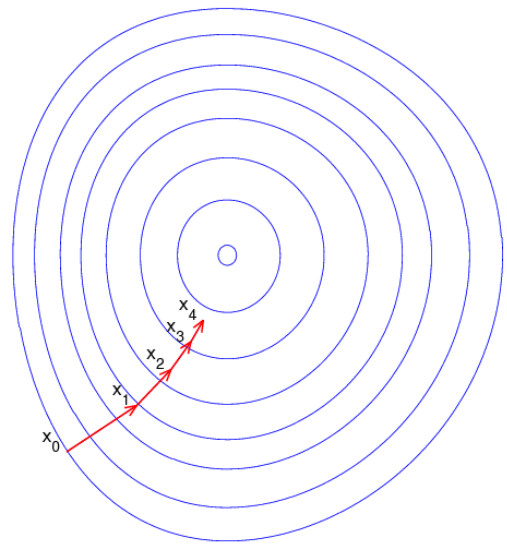

Figure 4.15: A visualization of the gradient descent process on a contour plot, progressing from initial guess ($\boldsymbol{x_0}$) to the most recent approximate minimum ($\boldsymbol{x_4}$) by traversing the maximum gradient.

Tradeoff Curve Metrics

The most valuable qualities in an energy-speed tradeoff curve are the minimization of energy (mean energy) for any given desired speed and the range of speed (speed range) which the tradeoff curves accommodate. Superior energy performance is signified by a curve which is positioned lower on the vertical energy axis, indicating that for a given point on the horizontal axis (step speed), the controller has found a more energy-efficient solution. A wider range on the horizontal axis indicates the controller can produce a great variety of step speeds, meaning a more versatile tradeoff curve. Figure 4.16 uses an illustration to convey visual examples of superior and inferior "mean energy" and "speed range". The formula for performance (P) is given in Eq. 4.4 as a function of the vector of all controller energy values ($\bar{E}$) and upper and lower bounds of

the resulting speeds (S_{max} and S_{min} respectively) with larger resulting negative values indicating superior performance. The coefficient of 2.0 for the speed range term was hand-tuned to produce a speed range that is similar to the tradeoff curve generated by the genetic optimization (as shown in Figure 3.3). During the performance evaluation process, any points in the tradeoff curve which fail at taking a successful step are removed from the curve and do not contribute to the speed range or mean energy calculations.

$$P = mean(\bar{E}) - 2.0(S_{max} - S_{min}) \qquad \text{Eq. 4.4}$$

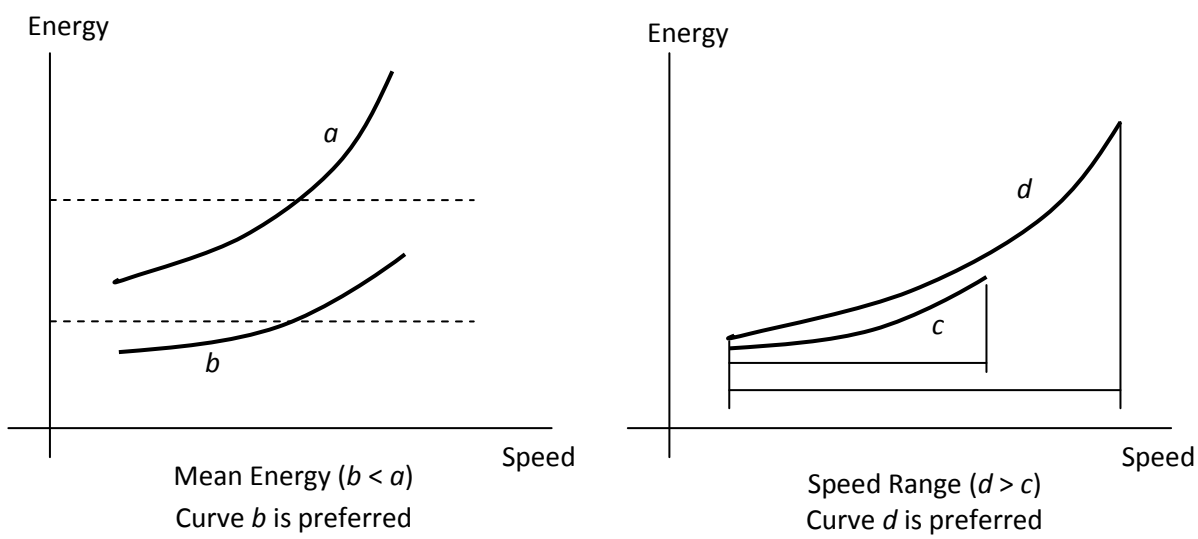

Figure 4.16: Illustrations of energy-speed tradeoff curves highlighting examples of varying performance in mean energy and speed range

To calculate P, a tradeoff curve must first be generated using the candidate heuristic bounding parameters. This process requires two steps: the generation of the control parameter sets and the subsequent testing of the control parameter sets for a single

step. The heuristic bounding parameters indicate an upper and lower bound, but to approximate this tradeoff curve, intermediate points must be calculated. This is achieved by generating controllers using various values of the energy-speed weighting factor (w_{ES_h}). Ten values of this weighting factor were spaced between zero and one, generating ten separate control parameter sets (the number ten was selected to be large enough to discern the quadratic shape of a resulting curve). Each of these ten control parameter sets are tested using the simulated compass gait. The resulting ten energy-speed data points are plotted on the same energy-speed "tradeoff space" for which the genetic algorithm results were reported. The state variables and desired step angle used, shown in Table 4.1, were the mean values of the genetic algorithm's state space range outlined in Table 3.2. This similarity makes the results of the genetic algorithm and gradient heuristic comparable.

Gradient-Descent Algorithm State Variables		
State Variable	Units	Value
X_1: vertical leg separation	(m)	0.00
X_2: horizontal leg separation	(m)	0.45
X_3: stance leg angular velocity	(º/sec)	-60.0
X_4: interleg angular velocity	(º/sec)	0
α_{des}: desired interleg angle	(º)	25.0
δ: terrain height	(m)	0.00

Table 4.1: State variables used for testing the gradient-descent algorithm

Approximated Gradient

Since no closed-form solution exists from which to take partial derivatives and analytically determine the gradient of the tradeoff curve performance, one must be

approximated for the purpose of the gradient-descent algorithm. For a similar problem with a single input variable, the gradient, equivalent to the slope in this simplified case, can be estimated by taking a finite "step" in one direction and comparing the output of the original position. This will give an estimation of the one-dimensional gradient, indicating the best direction for the next iteration of the gradient-descent algorithm. The same general process can be used for multiple variable inputs and exploring multi-dimensional space. For this simple estimation of the direction of maximum gradient, a small change is made in a diagonal direction (as shown in figure 4.17, checking points directly left and right for one dimension and in the four diagonal directions for two dimensions), and the change in output (tradeoff curve performance) is observed.

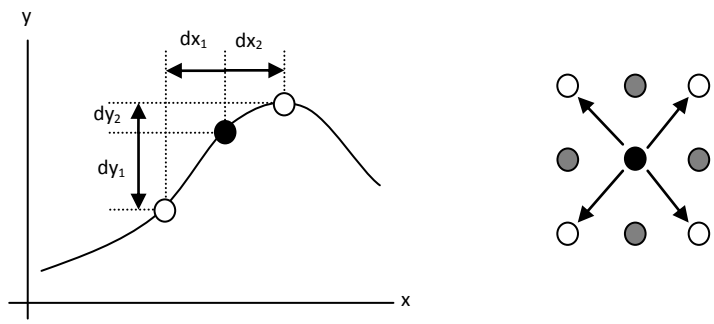

Figure 4.17: An illustration of a hypothetical one-dimensional (left) and two-dimensional performance curves. The one-dimensional case shows how moving in the two possible directions (left and right) yields predictions of the gradient. The two-dimensional case indicates the diagonal motion used to explore and approximate a higher dimensional gradient.

However, when using six variables (as is the case with the heuristic bounding parameters), the problem expands to six dimensions. In six dimensions, there are 64 (2^6) possible diagonal movements to explore and approximate the gradient by this manner.

The gradient descent algorithm explores each of these 64 possible options and finds the path which yields the greatest decrease. The tested point yielding the greatest decrease in the performance curve (performance is a bit of a misnomer as the formula yields larger negative values for good performance) becomes the new starting point for the next iteration of the algorithm. The changes in impulse magnitude, proportional and derivative gains applied to explore the nearby space are listed in Table 4.1. These values are identical to their corresponding mutation rates in the genetic algorithm described in Table 3.1. These relatively small values were chosen in order to keep the changes relatively small, decreasing the likelihood of a downward gradient being "skipped over". The algorithm is run until convergence which occurs when further iterations result in repeating previously encountered heuristic bounding parameters.

Gradient-Descent Exploration Values					
Impulse Magnitude		Proportional Gains		Derivative Gains	
0.125 Nm/s		0.125 Nm/degree		0.05 Nm-s/degree	
Initial Parameter Values					
Impulse Magnitude		Proportional Gains		Derivative Gains	
Minimum	Maximum	Minimum	Maximum	Minimum	Maximum
3.0 Nm/s	3.4 Nm/s	1.0 Nm/degree	1.4 Nm/degree	0.8 Nm-s/degree	1.2 Nm-s/degree

Table 4.2: Gradient-descent exploration values by which heuristic bounding parameters are changed in order to find the path of greatest descent; the initial bounding parameter values for the algorithm are also included in this table

Heuristic Tradeoff Curve Results

After tuning the heuristic bounding parameters using the aforementioned gradient-descent algorithm, the resulting tradeoff curve was plotted against the previous genetic algorithm data. Figure 4.18 shows the results of running the gradient-descent

algorithm to convergence. The "auto-tuned" heuristic curve closely approximates the data generated by the individually genetically optimized controllers. This result is very promising and has a number of potential implications for controlling the compass gait over a range of possible speed-energy demands.

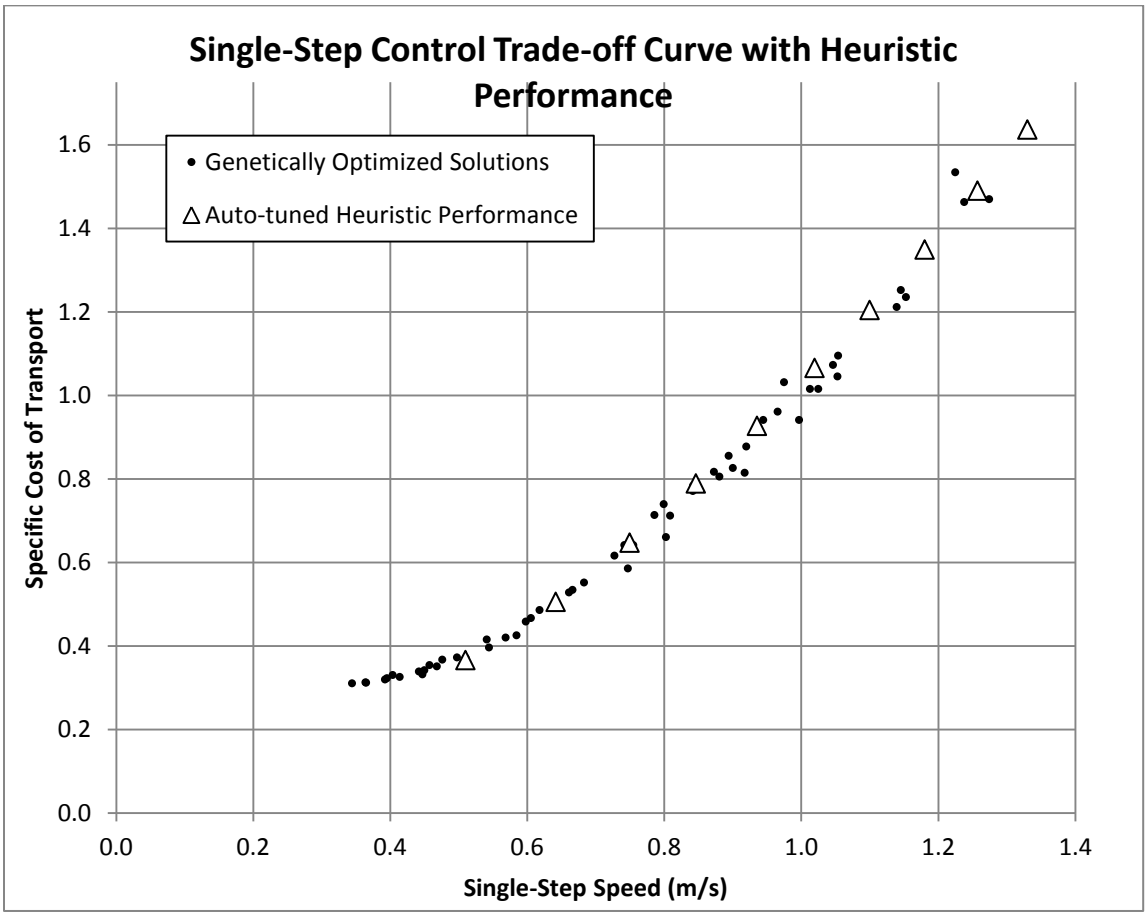

Figure 4.18: Tradeoff curves for genetically optimized solutions and the tuned heuristic tradeoff controller

It was hypothesized earlier in this chapter that redundant solutions may exist for values of the gain schedules. This result supports this hypothesis as many different gain profiles were generated by the genetic algorithm and the single mean gain profile

produced similar results. This is an encouraging finding as a representative mean profile may be useful in other regions of state space and step sizes than this one case. Furthermore, the closeness of the two tradeoff curves is worthy of note as the gradient descent algorithm had no means of knowing where the genetic algorithm tradeoff data was located. This implies that the gradient descent algorithm, which takes significantly less time to run than collecting the genetic algorithm data (at least a factor of ten), can predict an optimal performance curve on par with the genetic algorithm.

To test the notion that a representative gain profile can predict an optimal performance curve similar to that generated by the genetic algorithm, a different system state and step angle were chosen for a second run of the gradient-descent algorithm and the genetic algorithm. Table 4.3 lists the new system states and desired step angle for this new data set. These new states were selected to have a significant difference in most state variables (X_4, however, is almost always near zero since the derivative controller attenuates the interleg angular velocity) in order gauge versatility of the heuristic approach. In this test, the gradient-descent algorithm was run before generating new genetic algorithm data to control for any biases in selecting the gradient-descent initial conditions. All of the parameters, procedures and initial conditions were unchanged from the previous data set, with the exception that only 24 genetic optimizations were run in order to save computation time.

Gradient-Descent Algorithm State Variables		
State Variable	Units	Value
X_1: vertical leg separation	(m)	0.05
X_2: horizontal leg separation	(m)	0.55
X_3: stance leg angular velocity	(º/sec)	-30.0
X_4: interleg angular velocity	(º/sec)	0
α_{des}: desired interleg angle	(º)	20.0
δ: terrain height	(m)	0.00

Table 4.3: State variables used for a second run of the gradient-descent algorithm

The results for the second run of both the genetic algorithm and gradient-descent algorithm are plotted in Figure 4.19. Using the same mean gain schedule and gradient-descent algorithm, the tuned heuristic again closely matches the genetically optimized controller performance. This suggests that the tuned tradeoff heuristic (using the same mean gain profiles $\overline{K_{P_{base}}}$ and $\overline{K_{D_{base}}}$) may be a useful means of quickly (without running additional optimizations) generating controllers which produce a wide tradeoff range. This property suggests that this control heuristic, a representative "ramped" gain schedule coupled with linear scaling, can be called *tradeoff-conducive*.

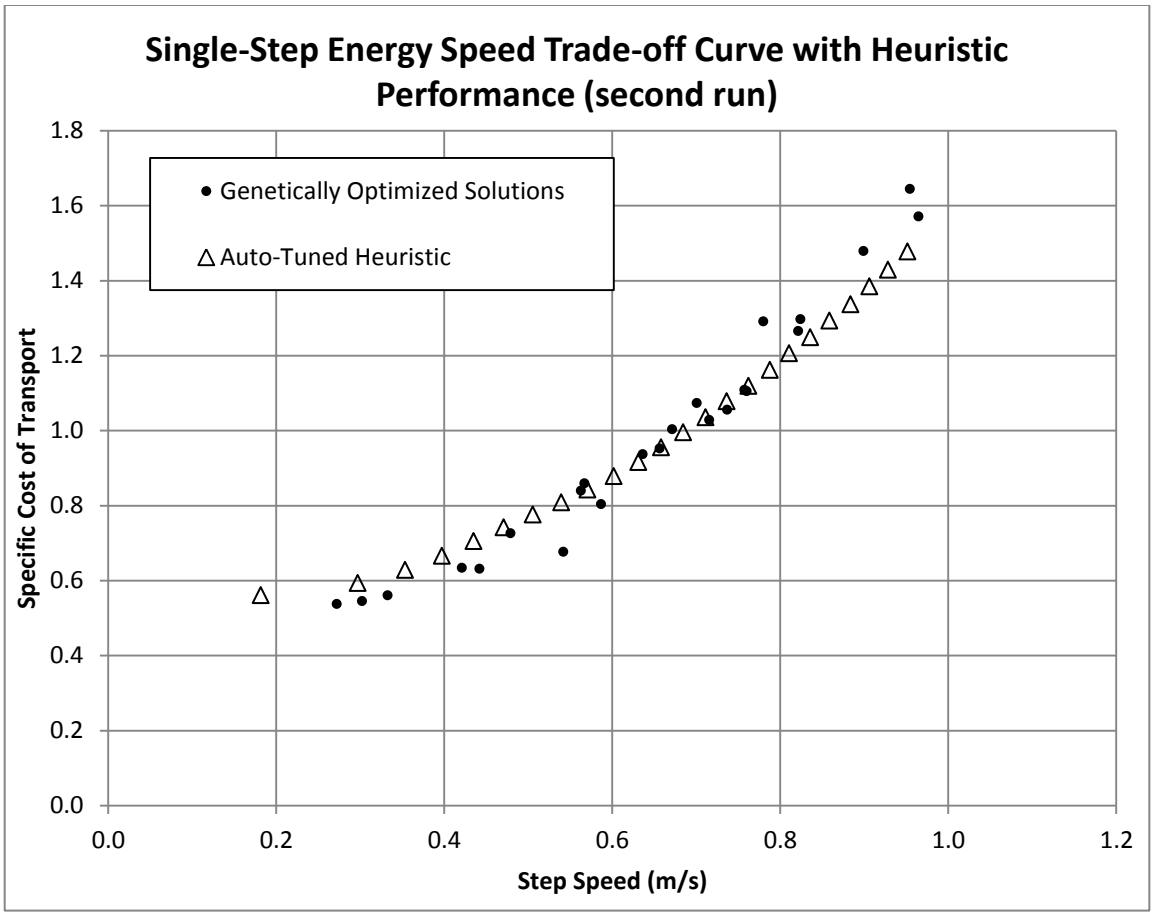

Figure 4.19: Tradeoff curves for genetically optimized solutions and the tuned heuristic tradeoff controller for the second set of state variables and desired step angle

While these results indicate that the mean "ramped" gain profile can be scaled effectively for tradeoffs, it is reasonable to question whether this particular profile is actually an improvement over other profiles. An exhaustive assessment of all other possible gain profiles is unreasonable, but it is worth investigating whether the ramped profile is better suited for tradeoffs than "traditional" profiles. The most traditional profile is a constant proportional and derivative gain, which is the equivalent of a single traditional PD controller. For this investigation, the mean "ramped" profile was further

averaged into a “flat” profile with a constant proportional and derivative gain (K_P = 4.01 Nm/degree and K_D = 0.85 Nm-s/degree) and tuned with the gradient-descent algorithm using the same parameters, initial values, state variables and desired step angle.

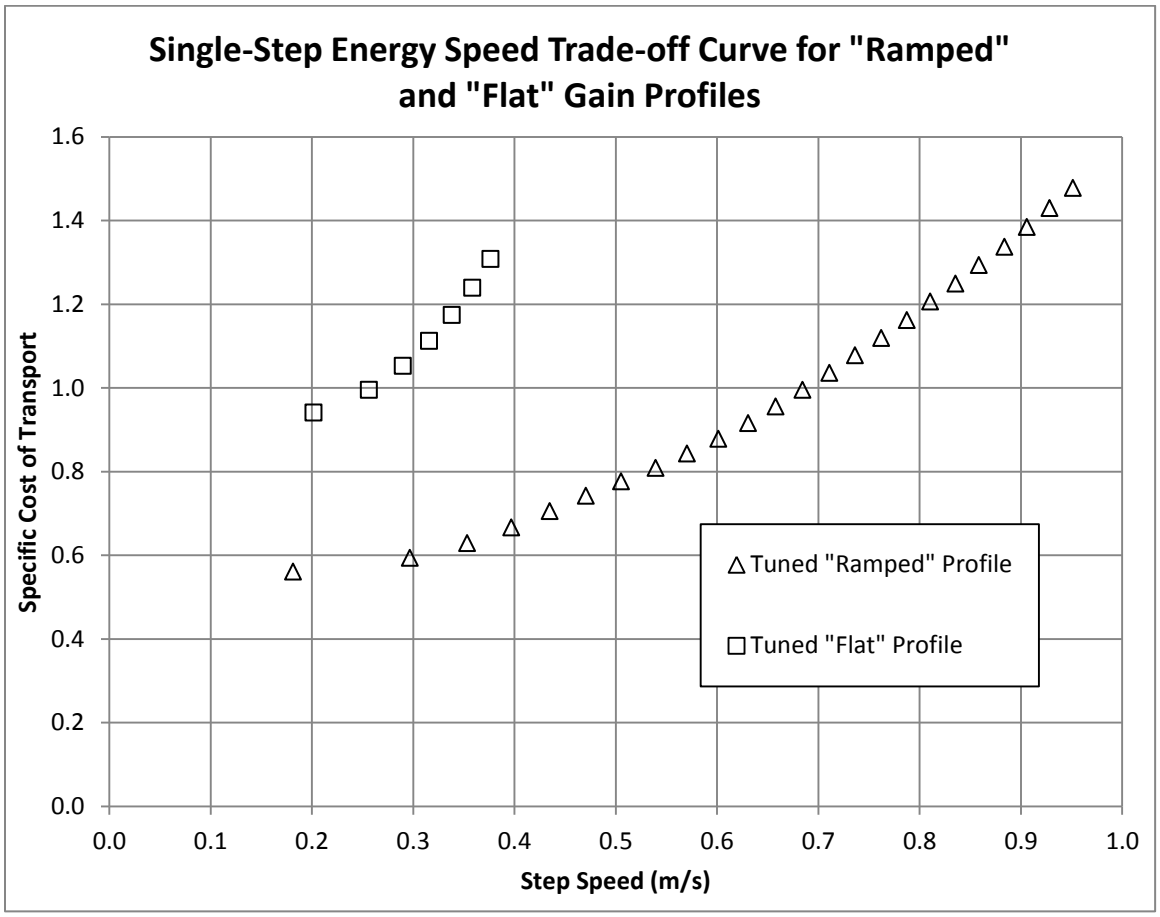

Figure 4.20: Comparison of the mean “ramped” and “flat” gain profiles based on their performance in energy-speed tradeoffs

Figure 4.20 compares the tuned performance of the “ramped” and “flat” profiles plotted on energy-speed coordinates. The ramped profile performance yields a range of speed approximately quadruple that of the flat profile. In addition, the energy cost of the flat profile is significantly greater (0.4 addition specific cost of transport at minimum)

than the flat profile. This suggests that the flat gain profile is not as tradeoff-conducive as the genetic optimization-inspired ramped profile. Put more simply, Figure 4.20 demonstrates that gain scheduling is superior to no gain scheduling for this application.

Conclusions

The endeavor of constructing a representative gain schedule and scaling the controller to achieve demands for energy economy and speed has been demonstrated to be successful for a single step. Furthermore, this success comes with the added computational benefit that these controllers can be rapidly generated by simple numerical scaling and not by optimization techniques. This tradeoff-conducive approach is critical for the final component of the completed walking controller, an overseeing "step chooser" or "agent" in the form of an artificially intelligent reinforcement learning algorithm. Controllers synthesized by this tradeoff-conducive control heuristic are ultimately used as a toolset at the disposal of the reinforcement learning agent.

Chapter 5: Reinforcement Learning

Inspired by the results of genetic optimization, a control heuristic was devised to produce energy-speed tradeoffs for a single step. If controlled using the initial state and terrain height at which the heuristic was tuned, the designed heuristic controller should never fail. However, this investigation seeks to control a simulated walking robot on rough terrain over the course of many steps (henceforth dubbed a *sustained walk*). In such a scenario, the walker is not constrained to a tiny slice of the state space and the terrain is modeled as a stochastically-generated series of varying terrain heights. This is important even if the terrain is flat and the heuristic is tuned over a large swath of the state space, the output states for an individual step may be unsuitable for continued walking. For example, even if a given single step is successful (in that the walker has not fallen), the system state after the step may have values (such as catastrophically slow velocities) which make future steps too difficult to achieve.

While the tradeoff-conducive control heuristic is novel, the controls problem posed by stochastic terrain is not. By intelligently choosing the step size according to the current system state, a compass gait walker has been shown to be able to traverse rough terrain. A reinforcement learning algorithm was implemented to develop a policy for choosing step sizes for each system state (Byl 2008). Similarly, to fulfill the goals of the investigation at hand, a reinforcement learning controller was devised to assess the current system state, selects the step size (α) and energy-speed weighting factor (w_{ES_h}) to produce energy-speed tradeoffs for a sustained walk.

Artificial Intelligence

Artificial intelligence (AI), a term coined in 1956 by computer scientist John McCarthy, refers to the science and engineering of making intelligent machines (McCarthy 2007). The applications of this broad concept in the decades since its inception have included planning (Wilkins 1988), pattern recognition (Bishop 1995), machine learning (Michalski 1986), and knowledge representation (Brachman 1985). Many of these applications of AI have found a home in the field of robotics. While planning (Latombe 1991) and obstacle recognition (Regensburger 1994) are important fields which employ artificial intelligence in robotics, it is machine learning which is most relevant to implementing this tradeoff-conducive control heuristic over a long sequence of steps or sustained walk.

Machine Learning

As has been loosely defined (Nilsson 1998), a machine learns whenever it changes its structure, program, or data (based on its inputs or in response to external information) in such a manner that its expected future performance improves. A variety of methods have been employed to facilitate this ability for a machine to change its structure, program, or data, which span two major categories: supervised and unsupervised learning. Supervised learning methods such as gradient-descent-learning neural networks function by observing and reacting to examples provided by a knowledgeable, external supervisor (Sutton 1998). Unsupervised methods lack such an

overseer or instructor. One such unsupervised approach, reinforcement learning, is the method of primary interest for this investigation.

Reinforcement Learning

Reinforcement learning does not require comparison to known solutions as a means of learning, but instead only requires interaction with its environment in order to change its program for improved performance. Central to reinforcement learning is the concept of *reward*, a numerical value awarded to the algorithm as a result of good performance. A reinforcement learning algorithm seeks to maximize a metric of long-term reward, termed *value* (Sutton 1998). By seeking maximum value instead of maximum reward, the algorithm is less likely to make short-term "greedy" mistakes which hamper long-term performance. Reinforcement learning algorithms come in many flavors such as policy iteration, value iteration, and asynchronous dynamic programming. Due to its prior use with the compass gait model (Byl 2008) and its relative computational simplicity compared to its counterparts, the value-iteration algorithm was employed for learning how to walk economically.

State and Action Value Functions

At its most basic level, the value iteration algorithm learns which actions are most "valuable" at particular system states. By identifying states which are valuable, actions can be selected which are likely to result in valuable states, a process which tends to converge to optimal performance. Assigning value to states and actions requires the

definition of a functional relationship between states, actions and their respective values. This need is met in the form of the *state value function* and the *action value function*.

While it is possible to define these as functions of continuous system states, this implementation of value iteration deals entirely with states and actions that have been discretized. A visualization of these discrete functions is shown in Figure 5.1, detailing their relationship to discrete states (*s*) and discrete actions (*a*). This manner of discretizing states was chosen to closely mirror prior implementation for using a value-iteration algorithm to control the compass gait (Byl 2009) and is detailed in Table 5.1. This provides a base for comparison with previously published data.

State Variable, Action Variable, and Stochastic variable Discretization			
Discretized Variables	Units	Elements	Discretization (MATLAB Vector Format)
X_1: vertical leg separation	m	19	[-0.1, -0.05, -0.04, -0.03:0.005:0.03, 0.04, 0.05, 0.1]
X_2: horizontal leg separation	m	10	[0.16:0.06:0.7]
X_3: stance leg angular velocity	deg/s	15	[-140:10:0]
X_4: swing interleg angular velocity	deg/s	9	[-20:5:20]
α_{des}: desired interleg angle	deg	9	[15:2.78:40]
δ: terrain height	m	17	[0.05, 0.04, 0.03:-0.005:-0.03, -0.04, -0.05]

Table 5.1: Discretization of variables for approximating the system states, actions, and terrain heights

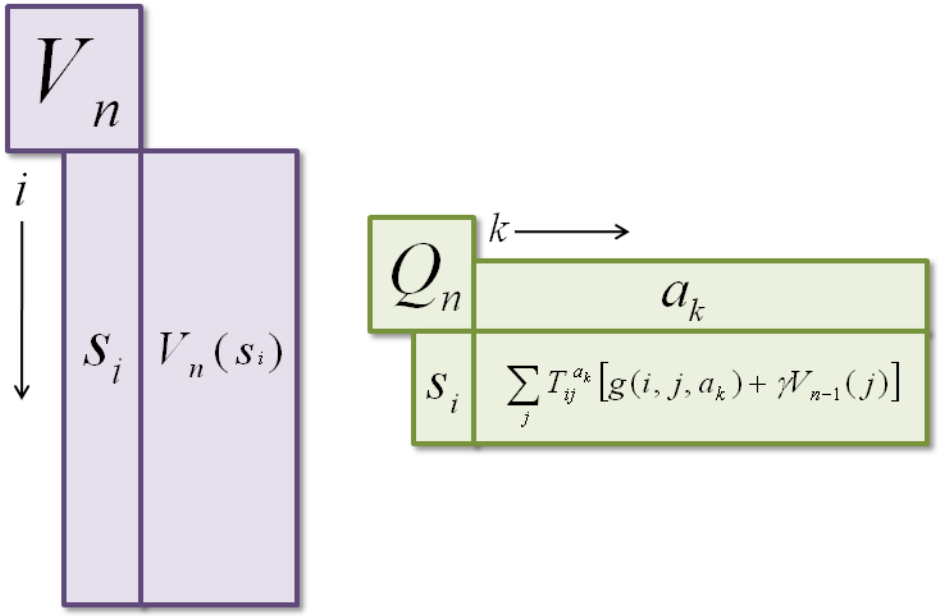

Figure 5.1: Visualization of state-value functions (V) and action-value functions (Q) as vectors indexed by i (enumerating distinct states) and k (enumerating distinct actions)

Value Iteration

The value-iteration algorithm requires a reward function (g), a state-value function (V), an action-value function (Q), a Markov Decision Process ($T_{ij}^{a_k}$), and a discount factor (γ). It should be noted that the term "vector" in this chapter refers to a one-dimensional programming structure (akin to a MATLAB vector). The *Markov Decision Process* (abbreviated *MDP*, notated $T_{ij}^{a_k}$) is a square matrix containing "transition probabilities", meaning each matrix entry contains the probability that a particular state (s_i) will result in another particular state (s_j) after performing a particular control action (a_k). A separate MDP is generated for each possible control action before any learning takes place, so the MDP does not update as a result of the reinforcement

learning method. Every single discrete state is simulated with each possible control action and every possible terrain height. The resulting state from each of these tests is binned to the nearest state in the discretized state space (described in detail in Table 5.1 with a single symmetric bin for each discrete state and bin boundaries placed at the average value of two adjacent states) and the probability of that terrain instance occurring is assigned to the MDP. Figure 5.2 illustrates the MDP, showing the states s_i and s_j on the axes and the probability of transition mapped inside.

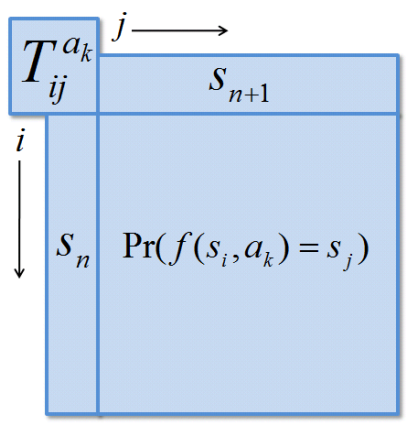

Figure 5.2: Markov Decision Process (MDP) Matrix

The state-value function (SVF) is a vector representing the "value" of being in a particular state (s_i) before an action is taken. Using a discrete function associating a value with each discrete state, the state-value function indicates whether a given state is likely to yield greater long-term reward, i.e., value. The SVF is initialized to all zero values, which are later updated through the value-iteration algorithm.

The action-value function (AVF) is a vector assessing the "value" of taking different control actions (a_k) at a given state, s_i . The AVF uses the MDP and state-value

function to probabilistically assess the value of each action. The cost function, $g(i, j, a_k)$, computes the costs of each possible result of this immediate, upcoming step, which are then multiplied by their respective probabilities. This process is detailed mathematically in Eq. 5.1.

$$Q_n = \sum_j T_{ij}^{a_k} \left[g(i, j, a_k) + \gamma V_{n-1}(j) \right] \qquad \text{Eq. 5.1}$$

Multiplying the probability of stepping into each possible state (via the MDP) with the corresponding value of that post-step state (via the SVF) yields an expected value of the future state. The expected future state value is multiplied by the adjustable "discount factor", ($0 \leq \gamma \leq 1$), which weights the importance of planning ahead in the value computation (higher discount factors favor long-term thinking). In this investigation, a discount factor of 0.9 was chosen to replicate prior published data. The action which yields the most "valuable" result, $min(Q_n)$, is selected for use by the controller. The AVF is completely recalculated before each step because the SVF, which is needed to calculate the AVF, is updated after every step. Figure 5.3 illustrates the aforementioned process by showing how the MDP and current SVF are incorporated into Eq. 5.1.

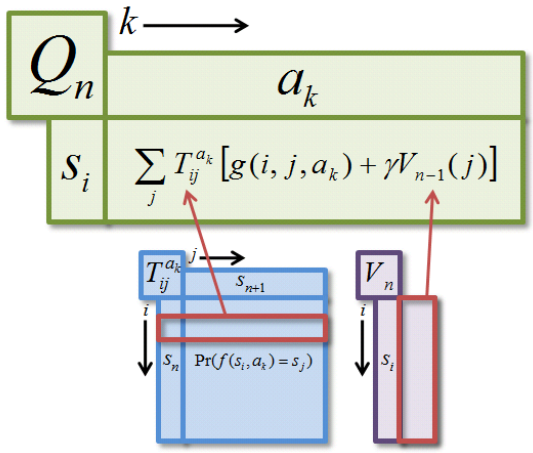

Figure 5.3: A visualization of the relationship between the action-value function (Q), Markov Decision Process matrix ($T_{ij}^{a_k}$), and state-value function (V) where n represents the current step number, i is the current state number, j is the state number potentially occupied for the next step, and k is an index enumerating all of the available state actions

The state-value function updates after every step, a process which is visualized in block form in Figure 5.4. The action-value function is calculated using the MDP and current state-value function. The best action is determined by selecting the discrete action with the optimal value. The state-value function is updated by replacing the entry for the current state with the optimal value in the action-value function, which is cartooned in Figure 5.5. Due to the AVF's consideration of future state values, the updated SVF now contains a better assessment of future performance when starting from a given state (s_i). The optimal step is then taken which interacts with a randomly generated terrain height, and results in a new state.

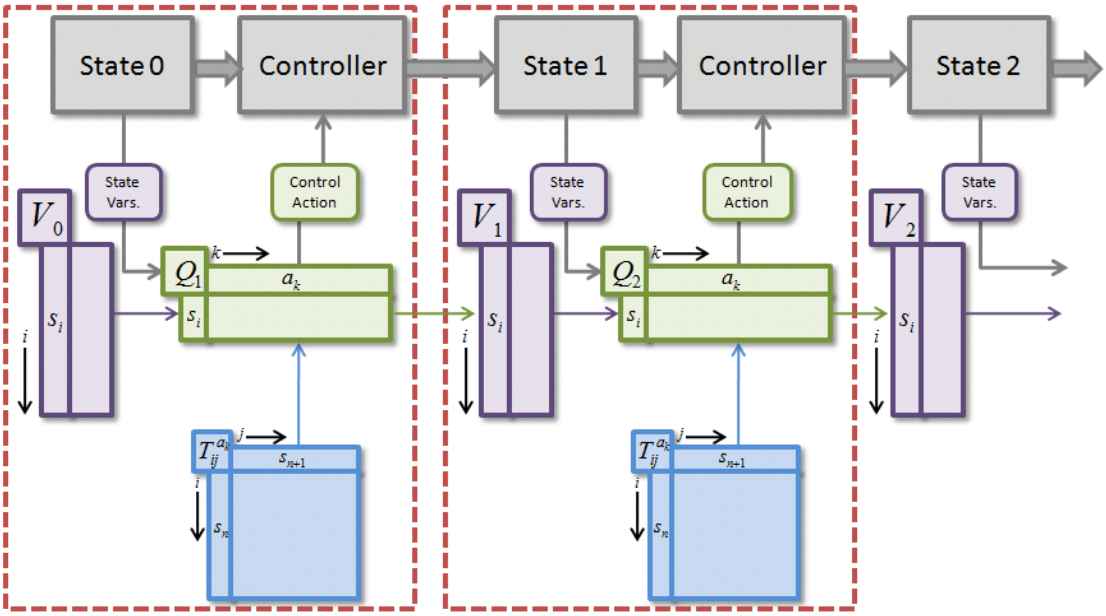

Figure 5.4: Block diagram of the value-iteration reinforcement learning process

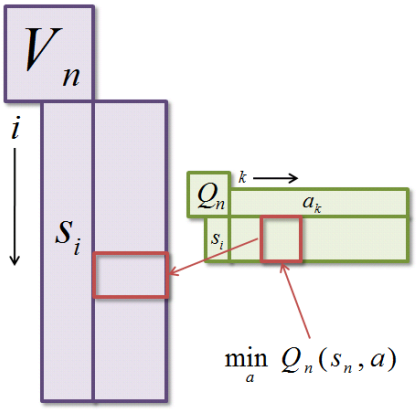

Figure 5.5: Visualization of the updating process for the state-value function using the value of the best action ($min(Q_n)$)

Mean First-Passage Time

For an application such as a robot walking on significantly rough terrain, classic definitions of stability regions are not necessarily the ideal standard for measuring the

reliability of a walker. When subjected to large stochastic disturbances, walking robots are likely to fail eventually due to some series of drastic events. As such, it is more apt to describe the robustness in terms of the expected duration between failures. This metric is dubbed the *mean first-passage time* (Tedrake 2006) which is the expected amount of time (in this case, the number of steps taken) before the robot first falls.

The calculation of the mean first passage time (MFPT) requires a Markov Decision Process matrix generated after the learning process is complete. This MDP is computed using the best actions possible (as determined by calculating the action-value function for every possible state and selecting the highest valued action, the result of which is called a *policy*) and determining the probability of transitioning to any of the given states. Given this MDP, which is a large square matrix, the eigenvalues are calculated and ranked. The second largest of these eigenvalues (λ_2) is used in Eq. 5.2 to calculate the MFPT. The details of the derivation of the MFPT formula (Byl 2009) are of cursory interest to this investigation, especially as it is used exclusively to compare preliminary results with other research.

$$MFPT = \frac{1}{1 - \lambda_2} \qquad \text{Eq. 5.2}$$

Approximate Optimal Robustness

A previously published approach (Byl 2009) used this algorithm to maximize the number of steps to failure while walking on rough terrain. This is accomplished by setting the reward function to encourage future steps and punishing failed steps as shown in Eq. 5.3.

$$g(i,j,a_k) = \begin{cases} 0, & if\ fallen \\ -1, & else \end{cases} \qquad \text{Eq. 5.3}$$

The stochastic terrain is defined by a Gaussian distribution with a terrain roughness defined by the standard deviation (and a mean value of zero). Once a roughness is selected, probabilities are binned into the nearest discrete terrain height listed in Table 5.1, creating a discrete probability function.

Value-Iteration Robustness Results

With each of the components in place for the value-iteration algorithm, it is important to compare the results of this algorithm with other data. Using nearly identical parameters, models, and methods (differing only in Poincare section definition and state discretization) to those in a paper by Byl and Tedrake (Byl 2009), the results should be comparable. For the single step controller, a standard proportional-derivative controller was used (K_P = 10 Nm/degree and K_D = 1 Nm/degree) and a constant pre-collision impulse magnitude (2 kg-m-s^{-1}) was used for in both this investigation and the referenced paper.

Many terrain roughness values were selected between 0.00375 and 0.0125 m, a range which is encompassed by previously published data (Byl 2009). For each of these values for terrain roughness, the value-iteration algorithm was run and the MFPT calculated every 10,000 steps. If three successive MFPT computations were found to have varied by less than 5%, the algorithm was considered converged and it was assumed that more learning would not make an appreciable difference in performance.

The resulting MFPT for many magnitudes of terrain roughness by both this investigation and the cited study (Byl 2009) are plotted in Figure 5.6 on semi-log axes. The trends of both data sets are largely the same, exhibiting an expected drop in MFPT as the terrain gets "rougher". The Hubicki-Buffinton data generally yielded an order of magnitude larger MFPT, but this may be due to differences in how the Poincare sections are defined (Chapter 2) or other differences in the state-space discretization (small changes to which the MFPT metric may be relatively sensitive). Nonetheless, this data confirms that the value-iteration algorithm programmed for this thesis produces robustness at least on par with published data.

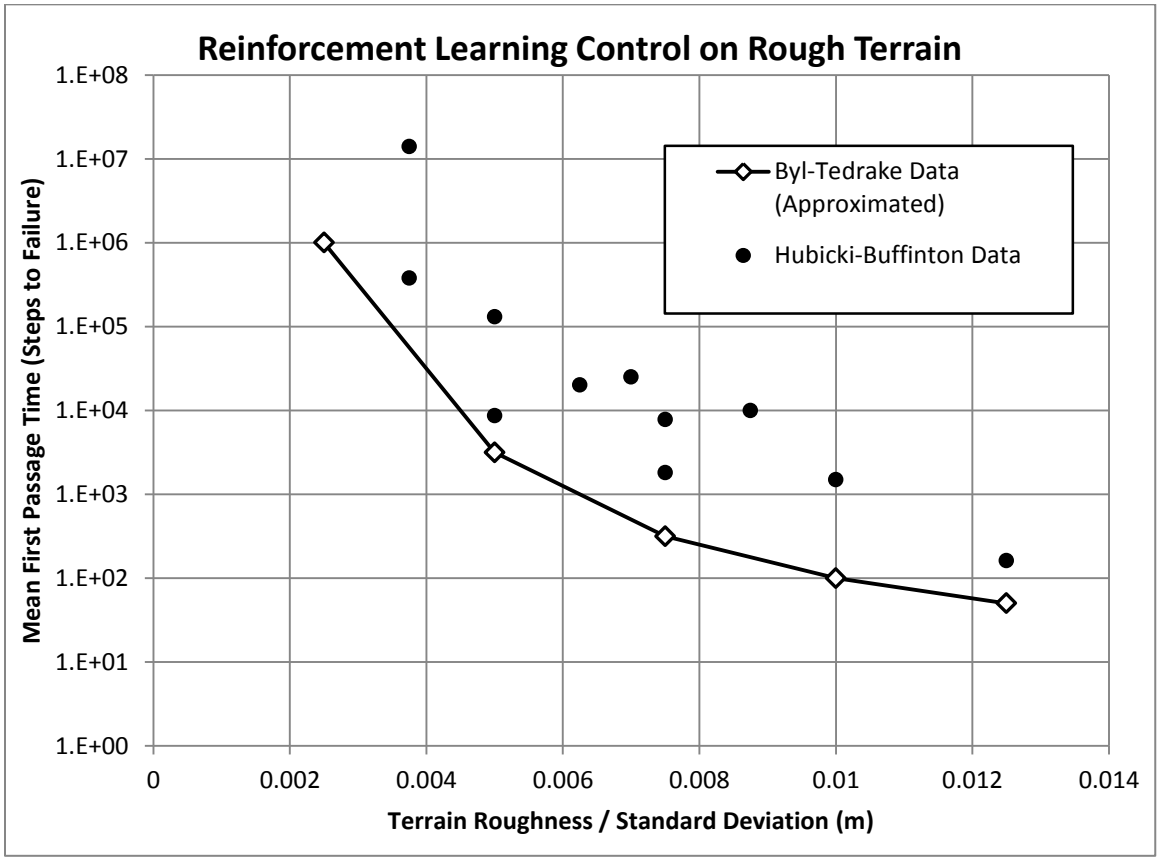

Figure 5.6: Comparison of resulting mean-first passage times of the value-iteration algorithm with published data (Byl 2009)

With the value-iteration algorithm in operation, it can now be coupled with the toolset provided by the tradeoff-conducive control heuristic. While the above value-iteration algorithm only had the ability to choose the desired step size, this next addition will allow the algorithm to choose both the step size and the control parameter set which propels the walker that single-step distance. Essentially, the algorithm will have to decide not only how far ahead to put the robot's foot, but also how quickly and economically it gets there. It is surmised that this added freedom will allow for long sustained walks exhibiting similar tradeoffs to those demonstrated in Chapter 4.

Chapter 6: Simulated Walking Experiment

With heuristically synthesized single step controllers and an overseeing algorithm to intelligently select step actions, the elements are in place for testing the complete hierarchical walking controller on rough terrain. Lingering questions about the proposed hierarchical controller largely concern the transition from single-step controller to "many-step" controller. The tradeoff-conducive controller heuristic provides a set of actions capable of a wide range of energy economy and speed performance to the value-iteration algorithm. It remains to be seen how effectively the value-iteration algorithm can make use of this toolset to produce a sustained walk.

Value-Iteration Cost Function

In the previous chapter, the cost function for the value-iteration algorithm rewarded each additional step taken, seeking to maximize the number of steps taken before falling. A cost function for the final hierarchical controller must incentivize robustness, energy economy, and speed. The proposed cost function, shown in Eq. 6.1, includes the distance taken by the step (D), the energy cost of the step (E), the time taken to complete the step (t), and their weighting factors (W_R, W_E, and W_S respectively).

$$C(D, E, S) = W_R D + W_E \frac{D}{E} + W_S \frac{D}{t} \qquad \text{Eq. 6.1}$$

The robustness term ($W_R D$) rewards longer travel distances, as opposed to the previous function which rewards an increased number of steps. The energy economy term ($W_E \frac{D}{E}$) is based on the inverse of the specific cost of transport metric, the energy

cost per unit distance traveled per unit weight. Simply incorporating the energy cost per step could result in many small energy-conservative steps instead of good energy economy for the sustained walk. The speed term ($W_S \frac{D}{t}$) is intuitive as it rewards larger distances traversed in less time. Like so many other cost functions discussed in this investigation, the weighting factors are chosen as needed by the user with larger values associated with greater incentive to improve robustness, energy economy, or speed. While a hypothetical user would select a single set of weighting factors to suit their application, this investigation selects a wide range to demonstrate a breadth of performance capability. Aside from a changed cost function, the value-iteration algorithm remains unchanged from its description in the previous chapter.

Action Space

A well-defined action space capable of producing near-optimal steps is only possible due to the tradeoff-conducive control heuristic. Using the ramped gain schedule plotted in Figure 4.12 and the six heuristic bounding parameters, the value-iteration algorithm has access to a library of controllers that have been shown to approximate optimal performance. A remaining weakness pertains to the fact that the heuristic bounding parameters are tuned to a single point in state space using the gradient-descent algorithm in Chapter 4. The state variables outlined in Table 6.1 were chosen because they represent the mean values of the defined discrete state space in Table 5.1. As such, they perhaps have the best chance of being the best point in state space to represent the entire state space. The one variable not chosen by taking the mean value is the terrain

height. The terrain height was set at 1 cm above the starting height, as that will be the terrain roughness (standard deviation) used for the final simulated walking experiment.

Gradient-Descent Algorithm State Variables		
State Variable	Units	Value
X_1: vertical leg separation	(m)	0.00
X_2: horizontal leg separation	(m)	0.46
X_3: stance leg angular velocity	(º/sec)	-70
X_4: interleg angular velocity	(º/sec)	0
α_{des}: desired interleg angle	(º)	27.5
δ: terrain height	(m)	0.01

Table 6.1: State variables used for gradient-descent algorithm to obtain the six heuristic bounding parameters for the simulated walking experiment

The resulting heuristic bounding parameters are listed in Table 6.2, values which fully detail how to scale the pre-collision impulse and gain schedule for varying needs of energy economy and speed. To review, a highly energy economical controller would be generated using values closer to the minimum values. Conversely, a high-speed controller would be generated by using values closer to the listed maximum values. For the final walking experiment, six control parameter sets (see Chapter 3 for definition) were generated using Eq. 4.1-4.3 in Chapter 4. These six evenly spaced values for energy-speed weighting factor (w_{ES_h}) spanned between zero and one (0.0, 0.2, 0.4, 0.6, 0.8, 1.0). Six control sets were presumed to provide sufficient resolution on the tradeoff curve for the value-iteration algorithm to have sufficient variety from which to select appropriate actions.

Heuristic Bounding Parameters for Walking Simulation					
Impulse Magnitude		Proportional Gains Scaling		Derivative Gain Scaling	
Minimum (b_{I_1})	Maximum (b_{I_2})	Minimum (b_{D_1})	Maximum (b_{P_2})	Minimum (b_{D_1})	Maximum (b_{D_2})
1.75 Nm/s	6.27 Nm/s	0.75	2.29	0.90	1.35

Table 6.2: Heuristic bounding parameters resulting from gradient-descent algorithm for the simulated walking experiment

Discrete Dynamics

The tradeoff-conducive controller heuristic has been used to create a set of six control parameter sets for a range of energy and speed demands. The discrete dynamics are computed using these six discrete control parameter sets, a discrete set of step sizes (α_{des}), a discrete state space, and a discretized terrain probability function. The discretizations of step sizes, state space, and terrain are listed in Table 6.3, which was kept identical to the values used in Chapter 5 for simplicity. Every combination of control parameter set, step size, state variable, and terrain height are simulated and the output state variables, distance traversed, time taken, and energy consumed for each are stored in a database. Energy consumed is computed as the sum of the kinetic energy imparted by the pre-collision impulse, the positive work done by the hip actuator, and the negative work done by the hip actuator (the hip actuator is not regenerative and consumes energy to dissipate the system energy). This database is a discrete representation of the dynamics and outcomes for all possible actions.

State Variable, Action Variable, and Stochastic variable Discretization			
Discretized Variables	Units	Elements	Discretization (MATLAB Vector Format)
X_1: vertical leg separation	m	19	[-0.1, -0.05, -0.04, -0.03:0.005:0.03, 0.04, 0.05, 0.1]
X_2: horizontal leg separation	m	10	[0.16:0.06:0.7]
X_3: stance leg angular velocity	deg/s	15	[-140:10:0]
X_4: swing interleg angular velocity	deg/s	9	[-20:5:20]
α_{des}: desired interleg angle	deg	9	[15:2.78:40]
δ: terrain height	m	17	[0.05, 0.04, 0.03:-0.005:-0.03, -0.04, -0.05]

Table 6.3: Discretization of variables for approximating the system states, actions, and terrain heights for simulated walking experiment

Walking Experiment Results

With the dynamics approximated by discretization, the value-iteration algorithm is ready to learn how to walk. A wide variety of combinations of weighting factors (W_R, W_E, and W_S) were used (all with values inclusively bounded by zero and one). The terrain roughness (standard deviation) is set to 1 cm, as this roughness yielded distances on the order of a kilometer in Chapter 5, which is a reasonable distance to simulate and save computation time

The simulation is initialized with a random state within the discretized state space and iterates the value-iteration algorithm with each step. If a fall occurs, the state is re-initialized to a random state and the iteration continues. The learning is halted when the simulated robot was able to successfully walk one kilometer many consecutive times (ten times was deemed to be sufficient given the computational rigor of the simulation) and

the average energy economy and speed compared to subsequent one-kilometer walks did not deviate by more than 5%. A change of less than 5% in performance over ten kilometers of learning indicates that there is likely little further learning that would greatly improve performance.

Each set of weighting factors resulted in an average walking speed and specific cost of transport (energy consumed per unit distance per unit weight) which was plotted on an energy-speed curve. The results of using a variety of weighting factors were surprisingly consistent. Dozens of unique sets of weighting factors ultimately lumped their resulting performance very near (within 0.05 m/s and 0.05 transport cost) one of three points which are the average values of many closely lumped solutions. These three points are plotted in Figure 6.1 against the corresponding performance of the tuned single-step controller (heuristic bounding parameters for which are listed in Table 6.2). In addition, the single-step tradeoff curve generated using no gain schedule (a “flat” profile optimized using the gradient-descent technique) is provided for comparison to traditional PD techniques.

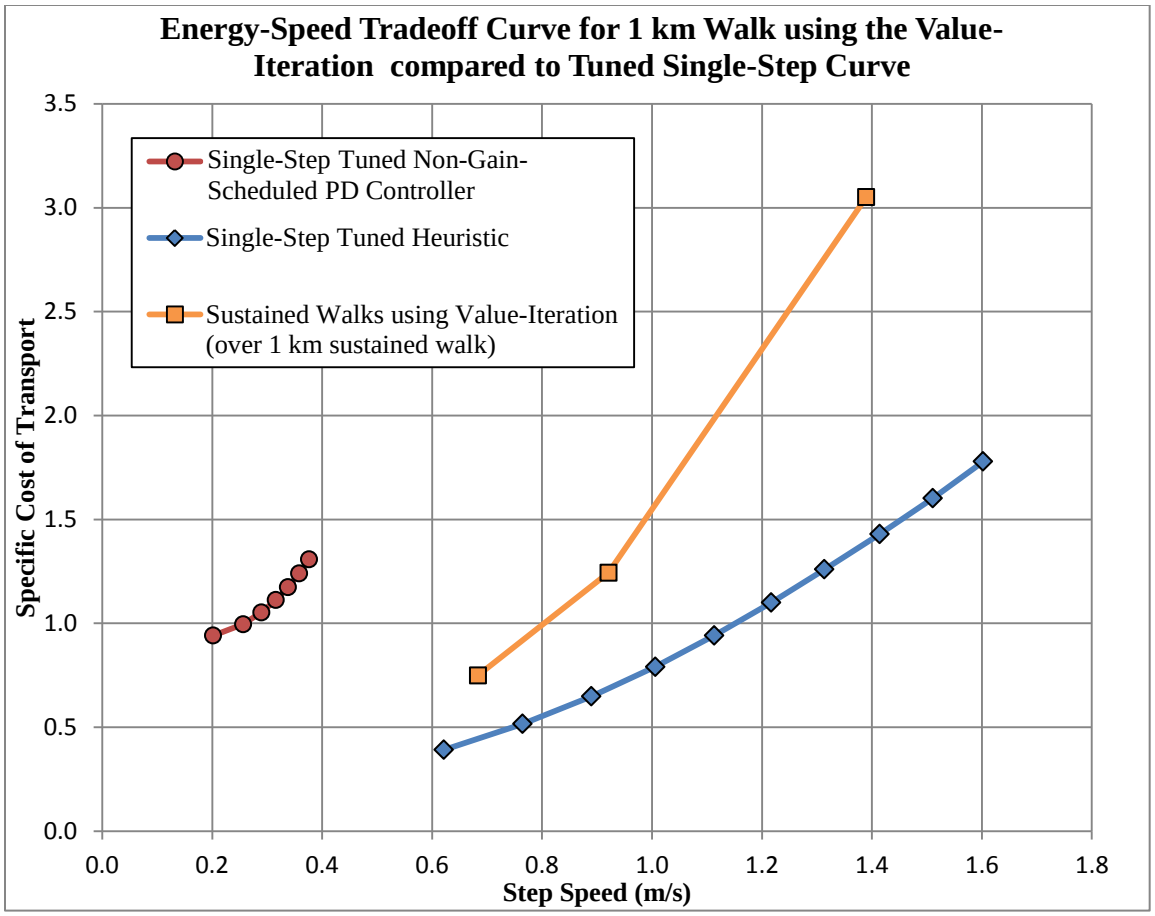

Figure 6.1: Energy-Speed Tradeoff Curve for 1 km Walk using Value-Iteration compared to the Tuned Single-Step Curve

The performance of the value-iteration algorithm is somewhat inferior to the tuned single-step controller, which is to be expected. The single-step controller is tuned to a single point in state space while the one-kilometer walker likely encounters a much larger swath of state space and must contend with changing terrain. What is important to note is that the speed range of the 1 kilometer walk is quite similar to that of the single-step curve. Also, the results are staggering when compared with the traditional (no gain schedule) single-step controller which was optimized using gradient descent (Chapter 4). The one-kilometer-walk tradeoff curve is much wider than the traditional controller,

which provides the increased versatility which is the primary motivation for developing a tradeoff-conducive controller heuristic. Additionally, the one-kilometer-walk delivers far more speed for a given energy cost even without the traditional controller suffering the effects of rough terrain over many steps.

Walking Experiment Conclusions

Despite the many possible sources of performance degradation in coupling the value-iteration algorithm with the single-step controller heuristic for a sustained walk, the resulting controller still resulted in a far greater range of performance (speed and energy economy) than an optimized version of the traditional non-gain-scheduled proportional derivative controller. The resulting hierarchical controller also made far more economical use of its energy budget for the achieved speeds. This makes a strong argument for the use of a “ramped” gain schedule for controlling the leg swing of walking robots.

One surprising result was the lack of resolution exhibited by the one-kilometer sustained walk curve, meaning that only three distinct points were found on the performance curve. It is likely caused by relatively few control parameter sets (six) being generated for this experiment. A small number of control parameter sets were generated in response to computational limitations. While it is mathematically and computationally simple to generate a control parameter set using a tradeoff-conducive control heuristic, calculating the discrete dynamics for expedient execution of the value-iteration algorithm is significantly slowed by each additional control option.

It is also not clear from these results how significant different system states are to the performance of the tuned heuristic bounding parameters. In this experiment, it can only be inferred that their impact is less than catastrophic for the complete hierarchical controller. The smaller the effect of the system state on performance, the more powerful this heuristic approach may be for controlling walking robots.

Chapter 7: Conclusions and Future Work

Summary

The aim of this investigation has been to develop solutions to some of the major problems currently hampering the field of dynamic walking and walking machines in general. In particular, the ability to robustly control a dynamic walking robot on rough terrain while optimizing energy consumption and speed has not been adequately addressed by research to date. The hierarchical controller developed in this thesis has been demonstrated to outperform more traditional approaches in simulation on a simple walking model known as the Compass Gait. The result is an artificially-intelligent algorithm which selects control actions generated by a novel control heuristic.

The development of this heuristic likely proved to be the most intriguing insight in the investigation. By taking a statistical look at the results of many computation-intensive genetic optimizations, it was discovered that a wide range of optimized gain schedules could be represented by a simple, optimization-inspired "ramped" gain profile. Furthermore, this ramped profile and pre-collision impulse (leg push-off) magnitude could simply be scaled to meet the energy and speed demands of the control designer, yielding a simple control heuristic. Upon further testing, this heuristic also proved capable of closely approximating the results of independent walking optimizations over a wide range of performance tradeoffs, and was hence termed a *tradeoff-conducive control heuristic*.

Conclusions

The tradeoff curves generated by the tradeoff-conducive control heuristic vastly outperformed controllers without gain scheduling in regard to the breadth of available tradeoffs and energy economy. The control heuristic also excelled at synthesizing these controllers using trivially simple calculations as opposed to the generally lengthy computations required by optimization techniques. When coupled with a value-iteration reinforcement learning algorithm, the control heuristic still greatly outperformed the tradition non-gain-scheduled controller. As gain schedules are not standard practice in controlling walking robots, these results make a compelling case for their use in producing a wide performance range.

The value-iteration algorithm has proved to be a useful technique, which is not surprising as reinforcement learning has been in use for decades. However, the algorithm becomes less useful as the number of state variables increases. Each state variable adds a new dimension to the problem which exponentially expands the computational demands. This property of reinforcement learning algorithms limits its utility for systems more complicated than the compass gait.

This novel approach of developing a tradeoff-conducive control heuristic is not without its flaws. The six heuristic bounding parameters must be deduced by some means, for which a gradient-descent algorithm was used. It is has not been determined how sensitive the heuristic parameters are to changes in the initial state variables. Furthermore, it has not been shown how changing the mass parameters of the robot model affects the validity of the ramped profile. The mass parameters in this study use

very heavy legs (each equivalent to the mass of the main body) as it accentuates the difficulty of the underactuated controls problem. Far lighter legs might affect the slopes of the ramped profile, but the lightened legs would also diminish their effect on the dynamics.

Future Work

There are some experiments which could bolster the findings presented here with additional evidence. Given the success of the control heuristic for the particular mass properties in this investigation, it would be prudent to replicate these results for robots with different mass parameters. In particular, the effect of different mass ratios between the legs and main body may change some aspect of the representative "ramped" gain profile. Quantification of the heuristic parameters' sensitivity to initial states would also be important in assessing the ease with which the heuristic can be generalized across the state space.

In regard to the value-iteration algorithm, the problem of dimensionality remains. It would be useful to run lengthier simulations to reproduce the sustained-walk tradeoff curve with more than six control parameter sets. In the long term, solutions to high-dimensional problems are needed which would allow for the control of robots with more degrees of freedom. A potential approach may rely on simplifying more articulated walkers into a model similar to the compass gait by lumping some links together as an approximately rigid leg. Such an approach may provide a straight-forward extension to walking with revolute knees akin to humans. It is likely that for a significant subset of

kneed walking motions, the compass gait model can serve as a close approximation to the more complex dynamics of walking with jointed knees. In such situations of approximate equivalence, the tradeoff-conducive control heuristic documented in this investigation could retain much of its performance capability with little modification.

More possibilities for advancements lie in the terrain modeling which is quite flexible in accommodating interesting features. By modifying the discrete probability distribution, the robot can encounter the equivalent of steps, hurdles, or other obstacles. The policy generated by the learning algorithm is capable of being analyzed and mined for useful stepping strategies, perhaps resulting in control heuristics for common obstacles. Wrapping terrain has also been explored and used to model intermittent terrain (pits) or very particularly shaped obstacles (Byl 2008).

Ultimately, the techniques developed for generating tradeoffs in control of walking robots can be applied to control in other applications. In the field of mobile robotics alone, there are likely tradeoffs in running, climbing, and jumping which may be synthesized in a similar manner to this investigation. In fact, some running models are even simpler than the compass gait, such as the spring-loaded inverted pendulum (SLIP) model (Schwind 1998). As robots inevitably become more dynamic, knowing the energetic costs and how to execute these inherently dynamic and energetically intensive actions will further enable these machines to make informed decisions about how to do more with less.

Bibliography

1. Bishop, C. (1995). *Neural Networks for Pattern Recognition*. Oxford: Clarendon.
2. Brachman, R., Levesque, H. (1985). *Readings in Knowledge Representation*. Los Altos, CA: Morgan Kaufmann.
3. Byl, K., Tedrake, R. (2008). Approximate optimal control of the compass gait on rough terrain. *International Conference on Robotics and Automation*, 1258–1263.
4. Byl, K., Tedrake, R. (2009). Metastable Walking Machines. *International Journal of Robotics Research*, 28(8), 1040-1064.
5. Collins, S., Ruina, A. (2005). A Bipedal Walking Robot with Efficient and Human-Like Gait. *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, 1983-1988.
6. Dawkins, R. (1986). *The Blind Watchmaker*. New York: Norton.
7. Espiau, B., Goswami, A., Compass gait revisited. (1994). *Proceedings of the International Federation on Automatic Controls, Symposium on Robot Control*, 839–846.
8. Goswami, A., Espiau, B., Keramane, A. (1996). Limit cycles and their stability in a passive bipedal gait. *Proceedings of IEEE International Conference on Robotics and Automation*, 246–251.
9. Hobbelen, D., Wisse, M. (2007). Limit cycle walking. *Humanoid Robots, Human-Like Machines*, 14.

10. Hurst, J., Chestnutt, J., Rizzi, A. (2007). Design and philosophy of the BiMASC, a highly dynamic biped. *Proceedings of the IEEE International Conference on Robotics and Automation*, 1863–1868.

11. Iida, F., Tedrake, R. (2009). Minimalistic control of a compass gait robot in rough terrain. *Proceedings of the IEE/RAS International Conference on Robotics and Automation*.

12. *International Journal of Humanoid Robotics*, 1(1), 157-173.

13. Karssen, J. (2007). *Design and construction of the Cornell Ranger, a world record distance walking robot*. Internship Report, Cornell University.

14. Kuo, A., Donelan, J., Ruina, A. (2002). Energetic Consequences of Walking Like an Inverted Pendulum: Step-to-Step Transitions. *Exercise & Sport Sciences Reviews*, 33(2), 88-97.

15. Kurfess, T. (2005). *Robotics and Automation Handbook*. Boca Raton: CRC.

16. Latombe, J. (1991). *Robot Motion Planning*. Boston: Kluwer Academic.

17. McCarthy, J. (2007). *Formal Reasoning Group*. Retrieved August 10, 2010, from http://www-formal.stanford.edu/jmc/whatisai/node1.html

18. McGeer, T. (1990). Passive dynamic walking. *International Journal of Robotics Research*, 9(2),62–82.

19. Mitchell, T., Carbonell, J., Michalski, R. (1986). *Machine Learning: a Guide to Current Research*. Boston: Kluwer Academic.

20. Nilsson, N. (1998). *Artificial Intelligence: a New Synthesis*. San Francisco, CA: Morgan Kaufmann.

21. Pratt, J., Carff, J., Drakunov, S., Goswami, A. (2006). Capture Point: A Step toward Humanoid Push Recovery. *International Conference on Humanoid Robots*, 200-207

22. Pratt, J., Krupp, B. (2008). Design of a bipedal walking robot. *Proceedings of the SPIE*, 6962.

23. Regensburger, U., Graefe, V. (1994) "Visual Recognition of Obstacles on Roads," *IEEE International Conference on Intelligent Robots and Systems*, 982-987.

24. Schwind, W., Koditschek, D. (1997). Characterization of monoped equilibrium gaits. *Proceedings of the IEEE International Conference on Robotics and Automation*, 1986-1992.

25. Spong, M. (1998). Underactuated mechanical systems. *Control Problems in Robotics and Automation*, 230.

26. Sutton, R., Barto, A. (1998) *Reinforcement Learning: an Introduction*. Cambridge, MA: MIT.

27. Tedrake R., Byl K., Pratt J. (2006). Probabilistic stability in legged systems: Metastability and the mean first passage time (FPT) stability margin. *In progress*.

28. Vukobratovic, M., Borovac, B. (2004). Zero-moment point – thirty five years of its life. *International Journal of Humanoid Robotics*, 1(1), 157-173.

29. Wiklendt, L., Chalup, S., Middleton, R. (2008). A Small Spiking Neural Network with LQR Control Applied to the Acrobot. *Neural Computing and Applications*, 17.

30. Wilkins, D. (1988). *Practical Planning: Extending the Classical AI Planning Paradigm*. San Mateo, CA: Morgan Kaufmann.

31. Wisse, M., Hobbelen, D., Schwab, A. (2007). Adding an Upper Body to Passive Dynamic Walking Robots by Means of a Bisecting Hip Mechanism. *IEEE Transactions on Robotics*, 23(1), 112-123.

32. Yin, K., Loken, K., Van de Panne, M. (2007). Simbicon: Simple biped locomotion control. *ACM Transactions on Graphics, Proceedings of SIGGRAPH*, 105.

Appendix

Note:

For privacy reasons, the body of the code for "EmailSimulationUpdate.m" and "RunUpdateRequestSystem.m" was not included in the documentation. Every instance of these functions may be commented out without adversely affecting the code.

Simulated Walking Experiment Code

Step 1: Run "GenerateMasterDynamicsTable.m"
Step 2: Run "RunStochasticHeuristicSetup.m"

BipedOneStepEOM.m

```
%% Inputs:
% IC_StLeg_position
% IC_Base_angle
% IC_StLeg_angvel
% IC_SwLeg_angle
% IC_SwLeg_angvel
% terrain_height_vector
% ACTIVATE_AT_LEG_CROSS
%
% angle_des
% angle_ratio_vec
% gain_schedule
% ratio_schedule
%
% StLeg_mass
% StLeg_inertia
% StLeg_length
% StLCG_ratio
% SwLeg_mass
% SwLeg_inertia
% SwLeg_length
% SwLCG_ratio
% MBody_mass

%% Outputs:
% Base_angle
% Base_angvel
% StLCG_position
% StLCG_velocity
% StLCG_angvel
% StLeg_angle
% StLeg_angvel
% StLeg_angaccel
% MBCG_position
```

```
% MBCG_velocity
% MBCG_angvel
% MBCG_accel
% SwLeg_angle
% SwLeg_angvel_joint
% SwLeg_angaccel2
% SwLCG_position
% SwLCG_velocity
% SwLCG_angvel
% SwLeg_angle2
% SwLCG_accel
% SwLeg_angaccel
% interleg_angle
% interleg_velocity
% hip_torque
% SwLeg_position
% SwLeg_velocity
% SwLeg_accel

% HitCheck
% TotalHits
% FallCheck

%%

% close all

SLOMO = 1;
FRAMES_PER_SECOND = 30*SLOMO;
NUM_SAMPLES = 1;

ANIMATION_ON = 0;

theta1_init = 1*(IC_Base_angle*pi/180) + pi/2;
theta2_init = pi - IC_SwLeg_angle*pi/180 - IC_Base_angle*pi/180;
theta_dot1_init = IC_StLeg_angvel*pi/180;
theta_dot2_init = IC_SwLeg_angvel*pi/180;

%t_max = 2; % assigned
dt = 1e-3;
if(t_max == 0)
    num_max = 1;
else
    num_max = floor(t_max/dt);
end

theta1 = theta1_init;
theta2 = theta2_init;
theta_dot1 = theta_dot1_init;
theta_dot2 = theta_dot2_init;

tau = 0;
m = StLeg_mass;
mh = MBody_mass;
L = StLeg_length;
% g = 9.81; % Assigned elsewhere
```

```
a = StLCG_ratio*L;
b = SwLCG_ratio*L;

m1 = m + mh/2;
m2 = m1;
l1 = a + b;
l2 = l1;
lc1 = L - b*m/m1;
lc2 = L - lc1;
I1 = m*(b-lc2)^2 + 0.5*mh*lc2^2;
I2 = I1;

theta1_vec = zeros(num_max,1);
theta2_vec = zeros(num_max,1);
theta_dot1_vec = zeros(num_max,1);
theta_dot2_vec = zeros(num_max,1);
hip_torque = zeros(num_max,1);

interleg_angle = zeros(num_max,1);
interleg_velocity = zeros(num_max,1);

SwLeg_position = zeros(num_max,2);
SwLeg_velocity = zeros(num_max,2);

MBody_inertia(3,3) = 0.0001;
StLeg_inertia(3,3) = 0.0001;
SwLeg_inertia(3,3) = 0.0001;

for index = 2:num_max
    theta1_vec(index-1) = theta1;
    theta2_vec(index-1) = theta2;

    theta_dot1_vec(index-1) = theta_dot1;
    theta_dot2_vec(index-1) = theta_dot2;

    interleg_angle(index-1) = (pi - theta2)*180/pi;
    interleg_velocity(index-1) = theta_dot2*180/pi;

    SwLeg_position(index-1,1) = L*cos(theta1) + L*cos(theta1+theta2);
    SwLeg_position(index-1,2) = L*sin(theta1) + L*sin(theta1+theta2);

    SwLeg_velocity(index-1,1)          =          L*cos(theta_dot1)          +
L*cos(theta_dot1+theta_dot2);
    SwLeg_velocity(index-1,2)          =          L*sin(theta_dot1)          +
L*sin(theta_dot1+theta_dot2);

    hip_torque(index-1) = tau;

    d11 = m1*lc1^2 + m2*(l1^2+lc2^2+2*l1*lc2*cos(theta2)) + I1 + I2;
    d12 = m2*(lc2^2 + l1*lc2*cos(theta2)) + I2;
    d22 = m2*lc2^2 + I2;

    h1            =            -m2*l1*lc2*sin(theta2)*theta_dot2^2            -
2*m2*l1*lc2*sin(theta2)*theta_dot2*theta_dot1;
    h2 = m2*l1*lc2*sin(theta2)*theta_dot1^2;

    p1 = (m1*lc1 + m2*l1)*g*cos(theta1) + m2*lc2*g*cos(theta1+theta2);
```

```
    p2 = m2*lc2*g*cos(theta1+theta2);

    tau  =  GetControlTorque(interleg_angle(index-1),  interleg_velocity(index-
1), -angle_des, angle_ratio_vec, gain_schedule, ratio_schedule);

    theta_dot_dot2 = (d11*(tau - h2 - p2) + d12*(h1 + p1))/(d11*d22 - d12^2);
    theta_dot_dot1 = (d12*theta_dot_dot2 + h1 + p1)/(-d11);

    theta_dot1 = theta_dot_dot1*dt + theta_dot1;
    theta_dot2 = theta_dot_dot2*dt + theta_dot2;

    theta1 = theta_dot1*dt + theta1;
    theta2 = theta_dot2*dt + theta2;

    %%
    %     interleg_angle(index-1)

end

theta1_vec(num_max) = theta1;
theta2_vec(num_max) = theta2;

if(num_max > 1)
    theta_dot1_vec(index-1) = theta_dot1;
    theta_dot2_vec(index-1) = theta_dot2;

    hip_torque(index-1) = tau;
else
    theta_dot1_vec(1) = theta_dot1;
    theta_dot2_vec(1) = theta_dot2;

    hip_torque(1) = tau;

    HitCheck = 1;
end

interleg_angle(num_max) = (pi + theta2)*180/pi;
interleg_velocity(num_max) = theta_dot2*180/pi;

clear MBCG_position
clear MBCG_velocity
clear StLCG_position
clear StLCG_velocity
clear SwLeg_position
clear SwLCG_position
clear SwLCG_velocity
clear Base_angvel
clear StLeg_angvel
clear StLCG_angvel
clear SwLeg_angvel

MBCG_position(:,1) = L*cos(theta1_vec)';
MBCG_position(:,2) = L*sin(theta1_vec)';

MBCG_velocity(:,1) = (-L*theta_dot1_vec.*sin(theta1_vec))';
MBCG_velocity(:,2) = (L*theta_dot1_vec.*cos(theta1_vec))';
```

```
StLCG_position(:,1) = StLCG_ratio*L*cos(theta1_vec)';
StLCG_position(:,2) = StLCG_ratio*L*sin(theta1_vec)';

StLCG_velocity(:,1) = (-L*StLCG_ratio.*sin(theta1_vec).*theta_dot1_vec)';
StLCG_velocity(:,2) = (L*StLCG_ratio.*cos(theta1_vec).*theta_dot1_vec)';

SwLeg_position(:,1) = L*cos(theta1_vec)' + L*cos(theta1_vec+theta2_vec)';
SwLeg_position(:,2) = L*sin(theta1_vec)' + L*sin(theta1_vec+theta2_vec)';

SwLCG_position(:,1)             =             L*cos(theta1_vec)'             +
SwLCG_ratio*L*cos(theta1_vec+theta2_vec)';
SwLCG_position(:,2)             =             L*sin(theta1_vec)'             +
SwLCG_ratio*L*sin(theta1_vec+theta2_vec)';

SwLCG_velocity(:,1)    =    (-L.*sin(theta1_vec).*theta_dot1_vec)'    +    (-
SwLCG_ratio*L*sin(theta1_vec+theta2_vec).*(theta_dot1_vec+theta_dot2_vec))';
SwLCG_velocity(:,2)       =       (L.*cos(theta1_vec).*theta_dot1_vec)'       +
(SwLCG_ratio*L*cos(theta1_vec+theta2_vec).*(theta_dot1_vec+theta_dot2_vec))';

Base_angle = (theta1_vec-pi/2)*180/pi;
SwLeg_angle = -theta2_vec*180/pi + 180 - Base_angle;

Base_angvel(:,1) = theta_dot1_vec*180/pi;
StLeg_angvel(:,1) = Base_angvel.*0;
StLCG_angvel(:,3) = theta_dot1_vec;
SwLeg_angvel(:,3) = (theta_dot1_vec+theta_dot2_vec);
MBCG_angvel = zeros(num_max,3);
SwLeg_angvel_joint = theta_dot2_vec*180/pi;

if(num_max > 1)
    left_height_vec            =            meshgrid([SwLeg_position(:,2);L],
terrain_height_vector);
    right_height_vec           =           meshgrid([-L;SwLeg_position(:,2)],
terrain_height_vector);

    terrain_mat = meshgrid(terrain_height_vector, ones(1,num_max+1))';
    position_mat             =             meshgrid([SwLeg_position(:,1)',-L],
zeros(1,length(terrain_height_vector)));

    %     HitCheck_raw = (left_height_vec <= terrain_mat).*(right_height_vec >
    %     terrain_mat).*([SwLeg_position(:,1)',-L] > 0.05);

    HitCheck_raw  =  (left_height_vec  <=  terrain_mat).*(right_height_vec  >
terrain_mat).*(position_mat > 0.05);

    HitCheck_raw(:,length([SwLeg_position(:,1)',-L]))          =          (1-
sum(HitCheck_raw,2));

    %Assumes only one terrain height
    %     HitCheck_raw(length(HitCheck_raw)) = 1;

    final_index = find(HitCheck_raw);
    if(isempty(final_index))
        final_index = length(HitCheck_raw)-1;
    end

    HitCheck = zeros(1,final_index(1));
```

```
    HitCheck(final_index) = 1;
else
    HitCheck = 1;
    final_index = 1;
end

% length(HitCheck)

if(ANIMATION_ON)
    %     hold on

    if(t_max == 0)
        time_interp = t_max;
        theta1_interp = theta1_vec;
        theta2_interp = theta2_vec;
    else
        time_interp = [0:1/FRAMES_PER_SECOND:t_max];
        theta1_interp = interp1(dt*[1:num_max], theta1_vec, time_interp);
        theta2_interp = interp1(dt*[1:num_max], theta2_vec, time_interp);
    end

    for index = [1:length(time_interp)]
        x1 = L*cos(theta1_interp(index));
        y1 = L*sin(theta1_interp(index));
        x2 = x1 + L*cos(theta1_interp(index)+theta2_interp(index));
        y2 = y1 + L*sin(theta1_interp(index)+theta2_interp(index));
        CMx1 = a*cos(theta1_interp(index));
        CMy1 = a*sin(theta1_interp(index));
        CMx2 = x1 + b*cos(theta1_interp(index)+theta2_interp(index));
        CMy2 = y1 + b*sin(theta1_interp(index)+theta2_interp(index));
        plot([0,x1], [0,y1], 'bo-', [x1,x2], [y1,y2], 'ro-', CMx1, CMy1, 'bx',
CMx2, CMy2, 'rx')
        axis equal
        axis([-2,2,-2,2])
        pause(1/FRAMES_PER_SECOND*SLOMO)
    end
end

% figure(4)
% plot(interleg_velocity)

% debug_BA = Base_angle(1)
% debug_SwA = SwLeg_angle(1)
%
% SwLeg_position
```

ComputeBestActionTransitions.m

```
%% Inputs:
% IC_StLeg_position
% IC_Base_angle
% IC_StLeg_angvel
% IC_SwLeg_angle
% IC_SwLeg_angvel
% terrain_height_vector
% ACTIVATE_AT_LEG_CROSS
%
% angle_des
% angle_ratio_vec
% gain_schedule
% ratio_schedule
%
% StLeg_mass
% StLeg_inertia
% StLeg_length
% StLCG_ratio
% SwLeg_mass
% SwLeg_inertia
% SwLeg_length
% SwLCG_ratio
% MBody_mass

%% Outputs:
% Base_angle
% Base_angvel
% StLCG_position
% StLCG_velocity
% StLCG_angvel
% StLeg_angle
% StLeg_angvel
% StLeg_angaccel
% MBCG_position
% MBCG_velocity
% MBCG_angvel
% MBCG_accel
% SwLeg_angle
% SwLeg_angvel_joint
% SwLeg_angaccel2
% SwLCG_position
% SwLCG_velocity
% SwLCG_angvel
% SwLeg_angle2
% SwLCG_accel
% SwLeg_angaccel
% interleg_angle
% interleg_velocity
% hip_torque
% SwLeg_position
% SwLeg_velocity
% SwLeg_accel

% HitCheck
% TotalHits
```

```
% FallCheck

%%

% close all

SLOMO = 1;
FRAMES_PER_SECOND = 30*SLOMO;
NUM_SAMPLES = 1;

ANIMATION_ON = 0;

theta1_init = 1*(IC_Base_angle*pi/180) + pi/2;
theta2_init = pi - IC_SwLeg_angle*pi/180 - IC_Base_angle*pi/180;
theta_dot1_init = IC_StLeg_angvel*pi/180;
theta_dot2_init = IC_SwLeg_angvel*pi/180;

%t_max = 2; % assigned
dt = 1e-3;
if(t_max == 0)
    num_max = 1;
else
    num_max = floor(t_max/dt);
end

theta1 = theta1_init;
theta2 = theta2_init;
theta_dot1 = theta_dot1_init;
theta_dot2 = theta_dot2_init;

tau = 0;
m = StLeg_mass;
mh = MBody_mass;
L = StLeg_length;
% g = 9.81; % Assigned elsewhere

a = StLCG_ratio*L;
b = SwLCG_ratio*L;

m1 = m + mh/2;
m2 = m1;
l1 = a + b;
l2 = l1;
lc1 = L - b*m/m1;
lc2 = L - lc1;
I1 = m*(b-lc2)^2 + 0.5*mh*lc2^2;
I2 = I1;

theta1_vec = zeros(num_max,1);
theta2_vec = zeros(num_max,1);
theta_dot1_vec = zeros(num_max,1);
theta_dot2_vec = zeros(num_max,1);
hip_torque = zeros(num_max,1);

interleg_angle = zeros(num_max,1);
interleg_velocity = zeros(num_max,1);
```

```
SwLeg_position = zeros(num_max,2);
SwLeg_velocity = zeros(num_max,2);

MBody_inertia(3,3) = 0.0001;
StLeg_inertia(3,3) = 0.0001;
SwLeg_inertia(3,3) = 0.0001;

for index = 2:num_max
    theta1_vec(index-1) = theta1;
    theta2_vec(index-1) = theta2;

    theta_dot1_vec(index-1) = theta_dot1;
    theta_dot2_vec(index-1) = theta_dot2;

    interleg_angle(index-1) = (pi - theta2)*180/pi;
    interleg_velocity(index-1) = theta_dot2*180/pi;

    SwLeg_position(index-1,1) = L*cos(theta1) + L*cos(theta1+theta2);
    SwLeg_position(index-1,2) = L*sin(theta1) + L*sin(theta1+theta2);

    SwLeg_velocity(index-1,1)          =          L*cos(theta_dot1)          +
L*cos(theta_dot1+theta_dot2);
    SwLeg_velocity(index-1,2)          =          L*sin(theta_dot1)          +
L*sin(theta_dot1+theta_dot2);

    hip_torque(index-1) = tau;

    d11 = m1*lc1^2 + m2*(l1^2+lc2^2+2*l1*lc2*cos(theta2)) + I1 + I2;
    d12 = m2*(lc2^2 + l1*lc2*cos(theta2)) + I2;
    d22 = m2*lc2^2 + I2;

    h1            =            -m2*l1*lc2*sin(theta2)*theta_dot2^2            -
2*m2*l1*lc2*sin(theta2)*theta_dot2*theta_dot1;
    h2 = m2*l1*lc2*sin(theta2)*theta_dot1^2;

    p1 = (m1*lc1 + m2*l1)*g*cos(theta1) + m2*lc2*g*cos(theta1+theta2);
    p2 = m2*lc2*g*cos(theta1+theta2);

    tau  =  GetControlTorque(interleg_angle(index-1),  interleg_velocity(index-
1), -angle_des, angle_ratio_vec, gain_schedule, ratio_schedule);

    theta_dot_dot2 = (d11*(tau - h2 - p2) + d12*(h1 + p1))/(d11*d22 - d12^2);
    theta_dot_dot1 = (d12*theta_dot_dot2 + h1 + p1)/(-d11);

    theta_dot1 = theta_dot_dot1*dt + theta_dot1;
    theta_dot2 = theta_dot_dot2*dt + theta_dot2;

    theta1 = theta_dot1*dt + theta1;
    theta2 = theta_dot2*dt + theta2;

    %%
    %     interleg_angle(index-1)

end

theta1_vec(num_max) = theta1;
theta2_vec(num_max) = theta2;
```

```
if(num_max > 1)
    theta_dot1_vec(index-1) = theta_dot1;
    theta_dot2_vec(index-1) = theta_dot2;

    hip_torque(index-1) = tau;
else
    theta_dot1_vec(1) = theta_dot1;
    theta_dot2_vec(1) = theta_dot2;

    hip_torque(1) = tau;

    HitCheck = 1;
end

interleg_angle(num_max) = (pi + theta2)*180/pi;
interleg_velocity(num_max) = theta_dot2*180/pi;

clear MBCG_position
clear MBCG_velocity
clear StLCG_position
clear StLCG_velocity
clear SwLeg_position
clear SwLCG_position
clear SwLCG_velocity
clear Base_angvel
clear StLeg_angvel
clear StLCG_angvel
clear SwLeg_angvel

MBCG_position(:,1) = L*cos(theta1_vec)';
MBCG_position(:,2) = L*sin(theta1_vec)';

MBCG_velocity(:,1) = (-L*theta_dot1_vec.*sin(theta1_vec))';
MBCG_velocity(:,2) = (L*theta_dot1_vec.*cos(theta1_vec))';

StLCG_position(:,1) = StLCG_ratio*L*cos(theta1_vec)';
StLCG_position(:,2) = StLCG_ratio*L*sin(theta1_vec)';

StLCG_velocity(:,1) = (-L*StLCG_ratio.*sin(theta1_vec).*theta_dot1_vec)';
StLCG_velocity(:,2) = (L*StLCG_ratio.*cos(theta1_vec).*theta_dot1_vec)';

SwLeg_position(:,1) = L*cos(theta1_vec)' + L*cos(theta1_vec+theta2_vec)';
SwLeg_position(:,2) = L*sin(theta1_vec)' + L*sin(theta1_vec+theta2_vec)';

SwLCG_position(:,1)             =             L*cos(theta1_vec)'             +
SwLCG_ratio*L*cos(theta1_vec+theta2_vec)';
SwLCG_position(:,2)             =             L*sin(theta1_vec)'             +
SwLCG_ratio*L*sin(theta1_vec+theta2_vec)';

SwLCG_velocity(:,1)    =    (-L.*sin(theta1_vec).*theta_dot1_vec)'    +    (-
SwLCG_ratio*L*sin(theta1_vec+theta2_vec).*(theta_dot1_vec+theta_dot2_vec))';
SwLCG_velocity(:,2)       =       (L.*cos(theta1_vec).*theta_dot1_vec)'       +
(SwLCG_ratio*L*cos(theta1_vec+theta2_vec).*(theta_dot1_vec+theta_dot2_vec))';

Base_angle = (theta1_vec-pi/2)*180/pi;
SwLeg_angle = -theta2_vec*180/pi + 180 - Base_angle;
```

```
Base_angvel(:,1) = theta_dot1_vec*180/pi;
StLeg_angvel(:,1) = Base_angvel.*0;
StLCG_angvel(:,3) = theta_dot1_vec;
SwLeg_angvel(:,3) = (theta_dot1_vec+theta_dot2_vec);
MBCG_angvel = zeros(num_max,3);
SwLeg_angvel_joint = theta_dot2_vec*180/pi;

if(num_max > 1)
    left_height_vec            =            meshgrid([SwLeg_position(:,2);L],
terrain_height_vector);
    right_height_vec           =           meshgrid([-L;SwLeg_position(:,2)],
terrain_height_vector);

    terrain_mat = meshgrid(terrain_height_vector, ones(1,num_max+1))';
    position_mat             =             meshgrid([SwLeg_position(:,1)',-L],
zeros(1,length(terrain_height_vector)));

    %     HitCheck_raw = (left_height_vec <= terrain_mat).*(right_height_vec >
    %     terrain_mat).*([SwLeg_position(:,1)',-L] > 0.05);

    HitCheck_raw  =  (left_height_vec  <=  terrain_mat).*(right_height_vec  >
terrain_mat).*(position_mat > 0.05);

    HitCheck_raw(:,length([SwLeg_position(:,1)',-L]))          =          (1-
sum(HitCheck_raw,2));

    %Assumes only one terrain height
    %     HitCheck_raw(length(HitCheck_raw)) = 1;

    final_index = find(HitCheck_raw);
    if(isempty(final_index))
        final_index = length(HitCheck_raw)-1;
    end

    HitCheck = zeros(1,final_index(1));
    HitCheck(final_index) = 1;
else
    HitCheck = 1;
    final_index = 1;
end

% length(HitCheck)

if(ANIMATION_ON)
    %     hold on

    if(t_max == 0)
        time_interp = t_max;
        theta1_interp = theta1_vec;
        theta2_interp = theta2_vec;
    else
        time_interp = [0:1/FRAMES_PER_SECOND:t_max];
        theta1_interp = interp1(dt*[1:num_max], theta1_vec, time_interp);
        theta2_interp = interp1(dt*[1:num_max], theta2_vec, time_interp);
    end
```

```
    for index = [1:length(time_interp)]
        x1 = L*cos(theta1_interp(index));
        y1 = L*sin(theta1_interp(index));
        x2 = x1 + L*cos(theta1_interp(index)+theta2_interp(index));
        y2 = y1 + L*sin(theta1_interp(index)+theta2_interp(index));
        CMx1 = a*cos(theta1_interp(index));
        CMy1 = a*sin(theta1_interp(index));
        CMx2 = x1 + b*cos(theta1_interp(index)+theta2_interp(index));
        CMy2 = y1 + b*sin(theta1_interp(index)+theta2_interp(index));
        plot([0,x1], [0,y1], 'bo-', [x1,x2], [y1,y2], 'ro-', CMx1, CMy1, 'bx',
CMx2, CMy2, 'rx')
        axis equal
        axis([-2,2,-2,2])
        pause(1/FRAMES_PER_SECOND*SLOMO)
    end
end

% figure(4)
% plot(interleg_velocity)

% debug_BA = Base_angle(1)
% debug_SwA = SwLeg_angle(1)
%
% SwLeg_position
```

ComputeMDP.m

```
% COMPUTE_MDP

% state_value_vector

MDP = sparse(max_state_num, max_state_num);
policy = zeros(max_state_num,1);

clock
for s = 1:max_state_num
    [action_index, action_transitions] =
ComputeBestActionTransitions(s, state_value_vector,
stochastic_transition_database, num_actions);

    policy(s) = action_index;
    MDP(s,:) = action_transitions;

%     if(sum(MDP(s,:) ~= stochastic_transition_database{9}(s,:)) > 0)
%         s
%         MDP(s,:)
%         pause
%     end

end
clock

% eigs(MDP)

save AI_allvars
```

ComputeProbDistribution.m

```
function prob_distribution = ComputeProbDistribution(terrain_sigma,
mean_value, bin_vector)

%NOTE: bin vector must be NON-INCREASING!!!

%ComputeProbDistribution
log_probability_cutoff = 4; % if 4, probabilities lower than 1:10^4
are ignored

NUM_SAMPLES = 1e6;

rand_samples = randn(1,NUM_SAMPLES)*terrain_sigma;

bin_bound = 0.5*diff(bin_vector)+bin_vector(1:(length(bin_vector)-1));

bin_sum = zeros(1,length(bin_vector));

for m = 1:NUM_SAMPLES
    bin_num = length(find(rand_samples(m) <= bin_bound))+1;

    bin_sum(bin_num) = bin_sum(bin_num) + 1;
end

prob_distribution = bin_sum/sum(bin_sum);

% Cuts off small probabilities (more remote than
10^log_probability_cutoff)
prob_distribution =
round(prob_distribution*10^log_probability_cutoff)/sum(round(prob_dist
ribution*10^log_probability_cutoff));

% plot(bin_vector, prob_distribution, 'kx-')
```

EnergyComputationOneStep.m

```
% EnergyComputationOneStep

StLCG_position = evalin('base','StLCG_position');
MBCG_position = evalin('base','MBCG_position');
SwLeg_position = evalin('base','SwLeg_position');
SwLCG_position = evalin('base','SwLCG_position');

MBCG_velocity = evalin('base','MBCG_velocity');
StLCG_velocity = evalin('base','StLCG_velocity');
Base_angvel = evalin('base','Base_angvel');
SwLCG_velocity = evalin('base','SwLCG_velocity');
SwLeg_angvel = evalin('base','SwLeg_angvel');
StLeg_angvel = evalin('base','StLeg_angvel');

s1 = size(MBCG_position);

if(final_index(1) > s1(1))
    final_index(1) = s1(1);
end

PE = g*(MBody_mass.*MBCG_position(1:final_index(1),2) +
StLeg_mass.*StLCG_position(1:final_index(1),2) +
SwLeg_mass.*SwLCG_position(1:final_index(1),2));

KE_MBody = 0.5*MBody_mass*(MBCG_velocity(1:final_index(1),1).^2 +
MBCG_velocity(1:final_index(1),2).^2) +
0.5*MBody_inertia(3,3).*(MBCG_angvel(1:final_index(1),3)*pi/180).^2;
KE_StLeg = 0.5*StLeg_mass*(StLCG_velocity(1:final_index(1),1).^2 +
StLCG_velocity(1:final_index(1),2).^2) +
0.5*StLeg_inertia(3,3).*(StLCG_angvel(1:final_index(1),3)*pi/180).^2;
KE_SwLeg = 0.5*SwLeg_mass*(SwLCG_velocity(1:final_index(1),1).^2 +
SwLCG_velocity(1:final_index(1),2).^2) +
0.5*SwLeg_inertia(3,3).*(SwLeg_angvel(1:final_index(1),3)*pi/180).^2;

% final_index
KE = KE_MBody + KE_StLeg + KE_SwLeg;
% PE

total_energy = PE-PE(1)+KE;
energy_delta = -1*total_energy +
[total_energy(2:length(total_energy));0];
energy_delta = energy_delta(1:(length(energy_delta)-1));

% max_energy = max(total_energy)
% min_energy = min(total_energy)

% figure(5)
% s_ed = size(energy_delta)
% plot(energy_delta)
% plot(total_energy)
```

```
% pause
% figure(1)

energy_added = sum((energy_delta>=0).*energy_delta);
energy_dissipated = sum((energy_delta<=0).*energy_delta);
energy_net = energy_added + energy_dissipated;

PE_delta = PE(length(PE)) - PE(1);
```

GenerateMasterDynamicsTable.m

```
%GenerateMasterDynamicsTable

%SAVE:
% [numX1, [X1vec]]
% [numX2, [X2vec]]
% [numX3, [X3vec]]
% [numX4, [X4vec]]
% [numDelta, [delta_vec]]
% [numAlpha, [alpha_vec]]
% [masterDynamicsTable (alpha slice 1)]
% [masterDynamicsTable (alpha slice 2)]
% ...
% [masterDynamicsTable (alpha slice numAlpha)]

% clc
clear
close all

try


    EMAIL_ALERT = 1;
    %     [last_update_time, last_update_text] = CheckUpdateRequests;

    addpath P:\UrbanRobots\private\Hubicki\Simulation\2009-12\Tools

    initial_time = clock;
    if(initial_time(5) < 10)
        initial_minutes = ['0' num2str(initial_time(5))];
    else
        initial_minutes = [num2str(initial_time(5))];
    end
    initial_hours = num2str(initial_time(4));
    initial_time_readout = [initial_hours ':' initial_minutes];

    text_body = ['Greetings,' 10 'Your simulation has commenced,
beginning at ' initial_time_readout ' local machine time.' 10
'Regards,' 10 '- CodeBot'];
    if(EMAIL_ALERT)
        EmailSimulationUpdate(['Simulation Commenced at ',
initial_time_readout], text_body)
    end



    % Hubicki state space discretization
    X1_vec = [-0.1, -0.05, -0.04, -0.03:0.005:0.03, 0.04, 0.05, 0.1];
    X2_vec = [0.16:0.06:0.7];
    X3_vec = [-140:10:0];
    X4_vec = [-20:5:20];
```

```
    % delta_terrain_vec = [0.029 0.02 0.01 0.0 -0.01 -0.02 -0.029];
    delta_terrain_vec = [0.05 0.04 0.03:-0.005:-0.03 -0.04 -0.05];
    %     alpha_vec = linspace(15, 40, 5);
    alpha_vec = linspace(27.5, 40, 3);
    impulse_value = 2;

    tradeoff_weighting_vec = linspace(0, 1, 6);

    % X3_vec = [-2.1:0.1:-1.4, -1.25, -1.1];
    % X4_vec = [-1, -0.7, -0.5:0.25:0.75, 1.1, 1.5];

    heuristic_parameters = [1.75 6.27 0.75 2.29 0.9 1.35];
%     heuristic_parameters = [1.75 6.27 0.75 2.29 0.9 0.135];

    numX1 = length(X1_vec);
    numX2 = length(X2_vec);
    numX3 = length(X3_vec);
    numX4 = length(X4_vec);

    state_dimensions = [numX1, numX2, numX3, numX4];

    [X1_bound_vec, X2_bound_vec, X3_bound_vec, X4_bound_vec,
max_state_num] = GetStateBoundaryVectors(X1_vec, X2_vec, X3_vec,
X4_vec);

    % state_in = [-0.02, 0.3, -50, -0];

    root_angle_ratio_vec = [-1   -0.75 -0.5  -0.25 0     0.25  0.5   0.75
      1];
    root_gain_schedule = 1.0.*[3.217524076  2.854678338
      3.476241818      3.823100867      3.871491193      4.046413024
      4.290005978      4.634429007      4.844104976      5.049151904];
    root_ratio_schedule = [1.486095457 1.045793855      1.085561499
      0.993975082      0.903970409      0.804858655      0.695194153
      0.460624279      0.228208339      0.752267483];

    %     angle_ratio_vec = [-1,1];
    %     gain_schedule = [10, 10, 10];
    %     ratio_schedule = [1 1 1];
    action = [25 2 1 1 1];

    angle_ratio_vec = root_angle_ratio_vec;
    gain_schedule = root_gain_schedule;
    ratio_schedule = root_ratio_schedule;

    % time1 = clock;
    % [states_out, is_fallen, distance_traversed, time_elapsed,
energy_consumed] = StepToStepTFarchive(state_in, action,
delta_terrain_vec);
    time2 = clock;
```

```
    state_num = zeros(1,length(delta_terrain_vec));

    % for m = 1:length(delta_terrain_vec)
    %     state_num(m) = GetStateNumber(states_out(m,:), is_fallen(m),
X1_bound_vec, X2_bound_vec, X3_bound_vec, X4_bound_vec,
state_dimensions);
    % end

    time1 = clock;

    blank_trans_matrix = zeros(max_state_num,
length(delta_terrain_vec));
    blank_trans_matrix(1,:) = ones(1, length(delta_terrain_vec));

    master_dynamics_database =
cell(length(tradeoff_weighting_vec),length(alpha_vec));
    master_distance_database =
cell(length(tradeoff_weighting_vec),length(alpha_vec));
    master_time_database =
cell(length(tradeoff_weighting_vec),length(alpha_vec));
    master_energy_database =
cell(length(tradeoff_weighting_vec),length(alpha_vec));

    clock

    for tradeoff_index = 1:length(tradeoff_weighting_vec)

        tradeoff_weight = tradeoff_weighting_vec(tradeoff_index);

        impulse_value = tradeoff_weight*(heuristic_parameters(2) -
heuristic_parameters(1)) + heuristic_parameters(1);
        gain_schedule =
root_gain_schedule.*(tradeoff_weight*(heuristic_parameters(4) -
heuristic_parameters(3)) + heuristic_parameters(3));
        ratio_schedule =
root_ratio_schedule.*(tradeoff_weight*(heuristic_parameters(6) -
heuristic_parameters(5)) + heuristic_parameters(5));

        for p = 1:length(alpha_vec)



            action = [alpha_vec(p) impulse_value 1 1 1];
            new_trans_matrix = blank_trans_matrix;

            new_distance_matrix = zeros(max_state_num,1);
            new_time_matrix = blank_trans_matrix;
            new_energy_matrix = zeros(max_state_num,1);


            for q = 2:max_state_num
```

```
                index_vector = GetStateIndices(q, state_dimensions);
                state_in = [X1_vec(index_vector(1))
X2_vec(index_vector(2)) X3_vec(index_vector(3))
X4_vec(index_vector(4))];

                [states_out, is_fallen, distance_traversed,
time_elapsed, energy_consumed] = StepToStepTFarchive(state_in, action,
delta_terrain_vec);

%                 energy_consumed1 = energy_consumed

                [dummy1, dummy1, dummy1, dummy1, energy_consumed,
dummy1, dummy1] = StepToStepGetEnergy(state_in, action,
delta_terrain_vec, [1,5]);

%                 energy_consumed

                %                 distance_traversed
                %                 time_elapsed
                %                 energy_consumed
                %                 pause

                state_num = zeros(1,length(delta_terrain_vec));

                for n = 1:length(delta_terrain_vec)
                    state_num(n) = GetStateNumber(states_out(n,:),
is_fallen(n), X1_bound_vec, X2_bound_vec, X3_bound_vec, X4_bound_vec,
state_dimensions);
                end

                if(sum(state_num ~= 1) > 0)
%                     state_in
%
%                     state_num
                    distance_traversed = max(states_out(:,2));
%                     time_elapsed
%                     energy_consumed
%                     pause
                end

                new_trans_matrix(q,:) = state_num;

                new_distance_matrix(q,1) = distance_traversed;
                new_time_matrix(q,:) = time_elapsed;
                new_energy_matrix(q,1) = energy_consumed;

            end

            master_dynamics_database{tradeoff_index, p} =
new_trans_matrix;
```

```
            master_distance_database{tradeoff_index, p} =
new_distance_matrix;
            master_time_database{tradeoff_index, p} = new_time_matrix;
            master_energy_database{tradeoff_index, p} =
new_energy_matrix;

            %             disp('cycle done')
            %             pause

        end
    end

    time2 = clock;

    time1
    time2

    %     cellplot(master_dynamics_database)


    final_time = clock;
    if(final_time(5) < 10)
        final_minutes = ['0' num2str(final_time(5))];
    else
        final_minutes = [num2str(final_time(5))];
    end
    final_hours = num2str(final_time(4));
    final_time_readout = [final_hours ':' final_minutes];

    elapsed_time = final_time - initial_time;
    elapsed_hours = num2str(elapsed_time*[0 0 24 1 0 0]');
    elapsed_minutes = num2str(elapsed_time(5));

    text_body = ['Greetings,' 10 'Your simulation beginning at '
initial_time_readout ' has executed without error.' 10 'Total run time:
' elapsed_hours ' hours and ' elapsed_minutes ' minutes.' ...
        10 'Regards,' 10 '- CodeBot'];
    if(EMAIL_ALERT)
        EmailSimulationUpdate(['Simulation Complete at '
final_time_readout '!'], text_body)
    end


    save 'dynamics_database.mat' master_dynamics_database
master_distance_database master_time_database master_energy_database
X1_vec X2_vec X3_vec X4_vec delta_terrain_vec alpha_vec
tradeoff_weighting_vec root_angle_ratio_vec root_gain_schedule
root_ratio_schedule heuristic_parameters
    save generated_dynamics_data_all

catch ME
    rep = getReport(ME)
```

```
    rep_email = getReport(ME, 'extended', 'hyperlinks', 'off');
    text_body = ['The error report was recorded as follows:' 10 ' ' 10
rep_email 10 ' ' 10 'Regards,' 10 '- CodeBot'];
    if(EMAIL_ALERT)
        EmailSimulationUpdate('Simulation Update: Untimely
Termination', text_body)
    end
end
```

GenerateStochasticHeuristicTable.m

```
function [stochastic_transition_database] =
GenerateStochasticTransitionHeuristicTable(master_dynamics_database,
prob_distribution, max_state_num, num_actions, num_weights)

blank_transition_table = sparse(max_state_num, max_state_num);

num_delta = length(prob_distribution);

stochastic_transition_database = cell(num_weights,num_actions);

for tradeoff_index = 1:num_weights
    for p = 1:num_actions

        %     clock

        current_transition_table = blank_transition_table;

        for q = 1:max_state_num
            for r = 1:num_delta

current_transition_table(q,master_dynamics_database{tradeoff_index,p}(
q,r)) =
current_transition_table(q,master_dynamics_database{tradeoff_index,p}(
q,r)) + prob_distribution(r);

                %
current_transition_table(q,master_dynamics_database{p}(q,r))
                %             pause(0.1)

            end
            %         if(sum(current_transition_table(q,:) > 0.999) ==
0)
            %             q
            %             current_transition_table(q,:)
            %             pause(0.1)
            %         end


        end

        stochastic_transition_database{tradeoff_index,p} =
current_transition_table;
    end
end
```

GetControlTorque.m

```
function tau = GetControlTorque(interleg_angle, interleg_velocity,
angle_des, angle_ratio_vec, gain_schedule, ratio_schedule)

% angle_ratio_vec = -angle_ratio_vec;

% angle_des
% angle_ratio_vec.*abs(angle_des)
% interleg_angle
% gain_schedule
% ratio_schedule
index = find(-interleg_angle > angle_ratio_vec.*abs(angle_des));

% if(-interleg_angle >=
angle_ratio_vec(length(angle_ratio_vec))*abs(angle_des))
%     KP = gain_schedule(length(angle_ratio_vec));
%     KD = KP*ratio_schedule(length(angle_ratio_vec));
%     disp('highest')
% elseif(-interleg_angle <= angle_ratio_vec(1)*angle_des)
%     KP = gain_schedule(1);
%     KD = KP*ratio_schedule(1);
%     disp('lowest')
% else
%     index = find(interleg_angle > angle_ratio_vec.*abs(angle_des));
%     KP = gain_schedule(index(length(index)));
%     KD = ratio_schedule(index(length(index)))*KP;
%     index(length(index))
% end

if(isempty(index))
    used_index = 1;
    KP = gain_schedule(used_index);
%     KD = KP*ratio_schedule(used_index);
    KD = ratio_schedule(used_index);
else
    used_index = max(index)+1;
    KP = gain_schedule(used_index);
%     KD = KP*ratio_schedule(used_index);
    KD = ratio_schedule(used_index);
end

% disp(['index: ',num2str(used_index)])

tau = -KP*(angle_des-interleg_angle) - KD*interleg_velocity;
```

GetStateBoundaryVectors.m

```
function [X1_bound_vec, X2_bound_vec, X3_bound_vec, X4_bound_vec,
max_state_num] = GetStateBoundaryVectors(X1_vec, X2_vec, X3_vec,
X4_vec)

X1_bound_vec = [X1_vec(1) 0.5*diff(X1_vec)+X1_vec(1:(length(X1_vec)-1))
X1_vec(length(X1_vec))];
X2_bound_vec = [X2_vec(1) 0.5*diff(X2_vec)+X2_vec(1:(length(X2_vec)-1))
X2_vec(length(X2_vec))];
X3_bound_vec = [X3_vec(1) 0.5*diff(X3_vec)+X3_vec(1:(length(X3_vec)-1))
X3_vec(length(X3_vec))];
X4_bound_vec = [X4_vec(1) 0.5*diff(X4_vec)+X4_vec(1:(length(X4_vec)-1))
X4_vec(length(X4_vec))];

max_state_num = 1 +
length(X1_vec)*length(X2_vec)*length(X3_vec)*length(X4_vec);
```

GetStateIndices.m

```
function index_vector = GetStateIndices(state_num, state_dimensions)

state_num = state_num - 1;

state1_index = floor((state_num-1)/prod(state_dimensions(2:4)))+1;
state_num = state_num - (state1_index-1)*prod(state_dimensions(2:4));

state2_index = floor((state_num-1)/prod(state_dimensions(3:4)))+1;
state_num = state_num - (state2_index-1)*prod(state_dimensions(3:4));

state3_index = floor((state_num-1)/prod(state_dimensions(4:4)))+1;
state_num = state_num - (state3_index-1)*prod(state_dimensions(4:4));

state4_index = state_num;

index_vector = [state1_index state2_index state3_index state4_index];
```

GetStateNumber.m

```
function state_num = GetStateNumber(state_in, is_fallen, X1_bound_vec,
X2_bound_vec, X3_bound_vec, X4_bound_vec, state_dimensions)



X1_bin = length(find(state_in(1) >= X1_bound_vec));
X2_bin = length(find(state_in(2) >= X2_bound_vec));
X3_bin = length(find(state_in(3) >= X3_bound_vec));
X4_bin = length(find(state_in(4) >= X4_bound_vec));

if(is_fallen || X1_bin == 0 || X1_bin > state_dimensions(1) || X2_bin
== 0 || X2_bin > state_dimensions(2) || X3_bin == 0 || X3_bin >
state_dimensions(3) || X4_bin == 0 || X4_bin > state_dimensions(4))
    state_num = 1; % Outside of discrete states so assigned to
absorbing state (state 1)
else
    state_num = 1 + (X1_bin-
1)*state_dimensions(2)*state_dimensions(3)*state_dimensions(4) +
(X2_bin-1)*state_dimensions(3)*state_dimensions(4) + (X3_bin-
1)*state_dimensions(4) + X4_bin;
end
```

ImpulseComputationEOM.m

```
function [omega2, omega3, KE_vec, PE_vec] =
ImpulseComputationEOM(impulse_mag)

%% Get vector information

% disp(['Impulse: ',num2str(impulse_mag)])

final_index = evalin('base','final_index(1)');

assignin('caller','final_index',final_index);
assignin('base','final_index',final_index);

%% Get Inertial Values

m1 = evalin('base','MBody_mass');
m2 = evalin('base','StLeg_mass');
m3 = evalin('base','SwLeg_mass');

J1 = evalin('base','MBody_inertia(3,3)');
J2 = evalin('base','StLeg_inertia(3,3)');
J3 = evalin('base','SwLeg_inertia(3,3)');

% Get incline measurement and gravity (for potential energy
calculation only)
incline = evalin('base','incline');
g = evalin('base','g');

%% Get CG Geometries

r1x = 0;
r1y = 0;

r2ax = evalin('base','StLCG_position(final_index, 1) -
IC_StLeg_position(1)');
r2ay = evalin('base','StLCG_position(final_index, 2) -
IC_StLeg_position(2)');
r2bx = evalin('base','StLCG_position(final_index, 1) -
MBCG_position(final_index, 1)');
r2by = evalin('base','StLCG_position(final_index, 2) -
MBCG_position(final_index, 2)');

r3ax = evalin('base','SwLCG_position(final_index, 1) -
SwLeg_position(final_index, 1)');
r3ay = evalin('base','SwLCG_position(final_index, 2) -
SwLeg_position(final_index, 2)');
r3bx = evalin('base','SwLCG_position(final_index, 1) -
MBCG_position(final_index, 1)');
r3by = evalin('base','SwLCG_position(final_index, 2) -
MBCG_position(final_index, 2)');
```

```
%% Get Position and Velocity Values

% Set Position Variables

MB_CGpos_x = evalin('base','MBCG_position(final_index, 1)');
MB_CGpos_y = evalin('base','MBCG_position(final_index, 2)');
StLeg_CGpos_x = evalin('base','StLCG_position(final_index, 1)');
StLeg_CGpos_y = evalin('base','StLCG_position(final_index, 2)');
SwLeg_CGpos_x = evalin('base','SwLCG_position(final_index, 1)');
SwLeg_CGpos_y = evalin('base','SwLCG_position(final_index, 2)');

% Set velocity variables
x1dot_pre = evalin('base','MBCG_velocity(final_index, 1)');
y1dot_pre = evalin('base','MBCG_velocity(final_index, 2)');
omega1_pre = 0;
x2dot_pre = evalin('base','StLCG_velocity(final_index, 1)');
y2dot_pre = evalin('base','StLCG_velocity(final_index, 2)');
omega2_pre = evalin('base','Base_angvel(final_index)');
x3dot_pre = evalin('base','SwLCG_velocity(final_index, 1)');
y3dot_pre = evalin('base','SwLCG_velocity(final_index, 2)');
omega3_pre = evalin('base','SwLeg_angvel(final_index, 3)');

%% Pre-Impulse Energy Computation

Body_Vars = [MB_CGpos_x, MB_CGpos_y, x1dot_pre, y1dot_pre,
omega1_pre*pi/180, m1, J1];
StLeg_Vars = [StLeg_CGpos_x, StLeg_CGpos_y, x2dot_pre, y2dot_pre,
omega2_pre*pi/180, m2, J2];
SwLeg_Vars = [SwLeg_CGpos_x, SwLeg_CGpos_y, x3dot_pre, y3dot_pre,
omega3_pre*pi/180, m3, J3];

[KE1, PE1] = EnergyComputation(Body_Vars, StLeg_Vars, SwLeg_Vars,
incline, g);

% disp('Pre-Impulse Energy')
% disp(['KE: ',num2str(KE1,10)])
% disp(['PE: ',num2str(PE1,10)])
% disp(['Total: ',num2str(KE1+PE1,10)])

%% Impulse Transformation Matrix
A = zeros(19);

A(01,01) = 1; A(01,10) = 1/m1;
A(02,02) = 1; A(02,11) = 1/m1;
A(03,03) = 1; A(03,10) = r1y/J1; A(03,11) = -r1x/J1;
A(04,04) = 1; A(04,12) = 1/m2; A(04,14) = 1/m2;
A(05,05) = 1; A(05,13) = 1/m2; A(05,15) = 1/m2;
A(06,06) = 1; A(06,12) = r2ay/J2; A(06,13) = -r2ax/J2; A(06,14) =
r2by/J2; A(06,15) = -r2bx/J2;
A(07,07) = 1; A(07,16) = 1/m3; A(07,18) = 1/m3;
A(08,08) = 1; A(08,17) = 1/m3; A(08,19) = 1/m3;
```

```
A(09,09) = 1; A(09,16) = r3ay/J3; A(09,17) = -r3ax/J3; A(09,18) =
r3by/J3; A(09,19) = -r3bx/J3;

A(10,10) = 1; A(10,14) = 1; A(10,18) = 1;
A(11,11) = 1; A(11,15) = 1; A(11,19) = 1;

A(12,12) = 1;
A(13,13) = 1;


% Changed for introducing pre-collision impulse
A(14,16) = 1;
A(15,17) = 1;

% A(14,07) = 1; A(14,09) = r3ay;
% A(15,08) = 1; A(15,09) = -r3ax;
% END change for impulse

A(16,01) = -1; A(16,04) = 1; A(16,06) = r2by;
A(17,02) = -1; A(17,05) = 1; A(17,06) = -r2bx;
A(18,01) = -1; A(18,07) = 1; A(18,09) = r3by;
A(19,02) = -1; A(19,08) = 1; A(19,09) = -r3bx;

%%
b = zeros(19,1);
b(01) = x1dot_pre;
b(02) = y1dot_pre;
b(03) = omega1_pre*pi/180;
b(04) = x2dot_pre;
b(05) = y2dot_pre;
b(06) = omega2_pre*pi/180;
b(07) = x3dot_pre;
b(08) = y3dot_pre;
b(09) = omega3_pre*pi/180;

%%
% Add Applied Impulse (as per Kuo 2005)

r_stance_x = r2ax - r2bx;
r_stance_y = r2ay - r2by;
r_stance_mag = sqrt(r_stance_x^2 + r_stance_y^2);

imp_comp_x = impulse_mag*r_stance_x/r_stance_mag;
imp_comp_y = impulse_mag*r_stance_y/r_stance_mag;

b(12) = -imp_comp_x;
b(13) = -imp_comp_y;

% b


%%
```

```
x = A\b;

% omega2 = x(6)*180/pi;
% omega3 = x(9)*180/pi;
%

% disp('Impulse Applied...')


%% Pre-Collision Impulse Applied
%*********************************************************************
*%
% Now calculating collision with ground
%*********************************************************************
*%

%% Ground Collision Computation
% Setting pre-collision variables equal to post-impulse variables

x1dot_pre = x(1);
y1dot_pre = x(2);
omega1_pre = x(3);
x2dot_pre = x(4);
y2dot_pre = x(5);
omega2_pre = x(6);
x3dot_pre = x(7);
y3dot_pre = x(8);
omega3_pre = x(9);

%% Post-Impulse/Pre-Collision Energy Computation

Body_Vars = [MB_CGpos_x, MB_CGpos_y, x1dot_pre, y1dot_pre, omega1_pre,
m1, J1];
StLeg_Vars = [StLeg_CGpos_x, StLeg_CGpos_y, x2dot_pre, y2dot_pre,
omega2_pre, m2, J2];
SwLeg_Vars = [SwLeg_CGpos_x, SwLeg_CGpos_y, x3dot_pre, y3dot_pre,
omega3_pre, m3, J3];

[KE2, PE2] = EnergyComputation(Body_Vars, StLeg_Vars, SwLeg_Vars,
incline, g);

% disp('Post-Impulse Energy')
% disp(['KE: ',num2str(KE2)])
% disp(['PE: ',num2str(PE2)])
% disp(['Total: ',num2str(KE2+PE2)])

% x

%% Collision Transformation Matrix

A = zeros(19);
```

```
A(01,01) = 1; A(01,10) = 1/m1;
A(02,02) = 1; A(02,11) = 1/m1;
A(03,03) = 1; A(03,10) = r1y/J1; A(03,11) = -r1x/J1;
A(04,04) = 1; A(04,12) = 1/m2; A(04,14) = 1/m2;
A(05,05) = 1; A(05,13) = 1/m2; A(05,15) = 1/m2;
A(06,06) = 1; A(06,12) = r2ay/J2; A(06,13) = -r2ax/J2; A(06,14) =
r2by/J2; A(06,15) = -r2bx/J2;
A(07,07) = 1; A(07,16) = 1/m3; A(07,18) = 1/m3;
A(08,08) = 1; A(08,17) = 1/m3; A(08,19) = 1/m3;
A(09,09) = 1; A(09,16) = r3ay/J3; A(09,17) = -r3ax/J3; A(09,18) =
r3by/J3; A(09,19) = -r3bx/J3;

A(10,10) = 1; A(10,14) = 1; A(10,18) = 1;
A(11,11) = 1; A(11,15) = 1; A(11,19) = 1;

A(12,12) = 1;
A(13,13) = 1;
A(14,07) = 1; A(14,09) = r3ay;
A(15,08) = 1; A(15,09) = -r3ax;
A(16,01) = -1; A(16,04) = 1; A(16,06) = r2by;
A(17,02) = -1; A(17,05) = 1; A(17,06) = -r2bx;
A(18,01) = -1; A(18,07) = 1; A(18,09) = r3by;
A(19,02) = -1; A(19,08) = 1; A(19,09) = -r3bx;

%%
b = zeros(19,1);
b(01) = x1dot_pre;
b(02) = y1dot_pre;
b(03) = omega1_pre*pi/180;
b(04) = x2dot_pre;
b(05) = y2dot_pre;
b(06) = omega2_pre*pi/180;
b(07) = x3dot_pre;
b(08) = y3dot_pre;
b(09) = omega3_pre*pi/180;

%%
x = A\b;

% Get post-collision states
x1dot_pre = x(1);
y1dot_pre = x(2);
omega1_pre = x(3);
x2dot_pre = x(4);
y2dot_pre = x(5);
omega2_pre = x(6);
x3dot_pre = x(7);
y3dot_pre = x(8);
omega3_pre = x(9);

%% Post-Collision Energy Computation
```

```
Body_Vars = [MB_CGpos_x, MB_CGpos_y, x1dot_pre, y1dot_pre, omega1_pre,
m1, J1];
StLeg_Vars = [StLeg_CGpos_x, StLeg_CGpos_y, x2dot_pre, y2dot_pre,
omega2_pre, m2, J2];
SwLeg_Vars = [SwLeg_CGpos_x, SwLeg_CGpos_y, x3dot_pre, y3dot_pre,
omega3_pre, m3, J3];

[KE3, PE3] = EnergyComputation(Body_Vars, StLeg_Vars, SwLeg_Vars,
incline, g);

% disp('Post-Collision Energy')
% disp(['KE: ',num2str(KE3,10)])
% disp(['PE: ',num2str(PE3,10)])
% disp(['Total: ',num2str(KE3+PE3,10)])
%
% % x
%
% disp([' '])

% pause

omega2 = x(6)*180/pi;
omega3 = x(9)*180/pi;

KE_vec = [KE1, KE2, KE3];
PE_vec = [PE1, PE2, PE3];
```

InitialConditionTransformation.m

```
function [theta_stance, theta_swing] =
InitialConditionTransformation(X1, X2, L)

x1 = X1/L;
x2 = X2/L;

theta1 = acos((-x1^2 - x2^2 + x1^4/(x1^2 + x2^2) + (x1^2*x2^2)/ ...
    (x1^2 + x2^2) - (x1*sqrt(4*x1^2*x2^2 - x1^4*x2^2 + 4*x2^4 -
2*x1^2*x2^4 - ...
    x2^6))/(x1^2 + x2^2))/(2*x2));

theta2 = -acos(x1^2/(2*x2) - x2/2 - x1^4/(2*x2*(x1^2 + x2^2)) -
(x1^2*x2)/ ...
    (2*(x1^2 + x2^2)) + (x1*sqrt(4*x1^2*x2^2 - x1^4*x2^2 + 4*x2^4 - ...
    2*x1^2*x2^4 - x2^6))/(2*x2*(x1^2 + x2^2)));

theta1 = pi - theta1;
theta2 = pi - theta2;

% plot([0,L*cos(theta1)],[0,L*sin(theta1)],'b-
',[L*cos(theta1),L*cos(theta1)+L*cos(theta2)],[L*sin(theta1),L*sin(the
ta1)+L*sin(theta2)],'r-')
% axis equal
% grid on

theta1b = theta1-pi/2;
theta2b = -1*(theta2+pi/2-2*pi);

% plot([0,-L*sin(theta1b)],[0,L*cos(theta1b)],'b-',[-L*sin(theta1b),-
L*sin(theta1b)-L*sin(theta2b)],[L*cos(theta1b),L*cos(theta1b)-
L*cos(theta2b)],'r-')
% axis equal
% grid on

theta_stance = theta1b*180/pi;
theta_swing = theta2b*180/pi;
```

InitStepToStepParams.m

```
function InitStepToStepParams(States_X)

% Currently starts simulation with Swing Leg just as it lands (before
% impulse and collision)
X1 = States_X(1); % Change in Height (Height_back_leg -
Height_front_leg)
X2 = States_X(2); % Horizontal Coordinate Change (Horz_coord_front_leg
- Horz_coord_back_leg)
X3 = States_X(3); % Stance Leg Angular Velocity
X4 = States_X(4); % Swing Leg Angular Velocity

Leg_length = 1.0;

[theta_stance, theta_swing] = InitialConditionTransformation(X1, X2,
Leg_length);

StOmega = X3;
SwOmega = X4;

%%
% Simulation Parameters
assignin('base','g', 9.81); %m/s^2
assignin('caller','g', 9.81);
% assignin('base','incline',0.5); %degrees

%%
% Component Parameters
assignin('base','k_spring',0); %Hip spring constant

%%
% Initial Conditions

%Body Parameters
% IC_MBody_position
assignin('base','IC_MBody_velocity',0);

%Stance Leg Parameters
assignin('base','IC_StLeg_position',[-X2, X1, 0]);
assignin('base','IC_Base_angle',theta_stance);
assignin('base','IC_StLeg_angvel',StOmega); %negative value indicates
"forward" swing

assignin('base','Stance_position',0);

%Swing Leg Parameters
assignin('base','IC_SwLeg_angle',theta_swing);
assignin('base','IC_SwLeg_angvel',SwOmega); %negative value indicates
"forward" fall
assignin('base','IC_StLeg_angle',0);
```

```
assignin('base','incline',0);
assignin('caller','incline',0);

assignin('caller','IC_StLeg_position',[-X2, X1, 0]);
assignin('caller','IC_Base_angle',theta_stance);
assignin('caller','IC_StLeg_angvel',StOmega); %negative value
indicates "forward" swing
assignin('caller','Stance_position',0);
assignin('caller','IC_SwLeg_angle',theta_swing);
assignin('caller','IC_SwLeg_angvel',SwOmega); %negative value
indicates "forward" fall
assignin('caller','IC_StLeg_angle',0);

%%
% Main Body Parameters
MBody_mass = 2;
assignin('base','MBody_mass',MBody_mass); %kg
assignin('base','MBody_inertia',diag([0.0001,0.0001,0.0001]));

assignin('caller','MBody_mass',MBody_mass); %kg
assignin('caller','MBody_inertia',diag([0.0001,0.0001,0.0001]));
%%
% Stance Leg Parameters
StLeg_mass = 2;
StLeg_length = Leg_length;
assignin('base','StLeg_mass',StLeg_mass);
assignin('base','StLeg_inertia',diag([0.0001,0.0001,0.0001]));
assignin('base','StLeg_length',StLeg_length);
assignin('base','StLCG_ratio',0.5); %ratio of distance from hip joint
to leg CG to length of leg (0.1 = CG is 10% down length of leg)

assignin('caller','StLeg_mass',StLeg_mass);
assignin('caller','StLeg_inertia',diag([0.0001,0.0001,0.0001]));
assignin('caller','StLeg_length',StLeg_length);
assignin('caller','StLCG_ratio',0.5);

%%
% Swing Leg Parameters
SwLeg_mass = 2;
SwLeg_length = Leg_length;
assignin('base','SwLeg_mass',SwLeg_mass);
assignin('base','SwLeg_inertia',diag([0.0001,0.0001,0.0001]));
assignin('base','SwLeg_length',SwLeg_length);
assignin('base','SwLCG_ratio',1-0.5); %ratio of distance from hip
joint to leg CG to length of leg (0.1 = CG is 10% down length of leg)

assignin('caller','SwLeg_mass',SwLeg_mass);
assignin('caller','SwLeg_inertia',diag([0.0001,0.0001,0.0001]));
assignin('caller','SwLeg_length',SwLeg_length);
assignin('caller','SwLCG_ratio',1-0.5);
%%
%Calculation of Initial Variables
```

```
% assignin('base','StLeg_angle',0);
% assignin('base','SwLeg_angle',2*angle1);
```

RunStochasticHeuristicSetup

```
% RunStochasticSetup

clc
% clear
close all

load dynamics_database

NUM_EPISODES = 1e5;

STATE_INITIALIZATION_ON = 0;
DETERMINED_START_STATE = 1;

USE_RANDOM_RESTART = 1;
RANDOM_RESTART_EVERY = 1000; %meters of travel

num_actions = length(alpha_vec);
num_weights = length(tradeoff_weighting_vec);

terrain_sigma = 0.01;

SUSTAINED_WALK_TRADEOFF_WEIGHTS = [1 1.25 0.0325]; %RES


starting_state = [0; 0.46; -70; -0];

[X1_bound_vec, X2_bound_vec, X3_bound_vec, X4_bound_vec, max_state_num]
= GetStateBoundaryVectors(X1_vec, X2_vec, X3_vec, X4_vec);

state_dimensions = [length(X1_vec), length(X2_vec), length(X3_vec),
length(X4_vec)];

% prob_distribution = [0 0 0 0 0 0.0060 0.0605 0.2420 0.3830 0.2420
0.0605 0.0060 0 0 0 0 0];
prob_distribution = ComputeProbDistribution(terrain_sigma, 0,
delta_terrain_vec);

cum_probs = cumsum(prob_distribution);

clock

stochastic_transition_database =
GenerateStochasticTransitionHeuristicTable(master_dynamics_database,
prob_distribution, max_state_num, num_actions, num_weights);


disp('Stochastic Generation Complete!  Learning Commencing...')

state_value_vector = -1*zeros(max_state_num, 1);
state_value_vector(1) = 0;
```

```
states_in = [-0.02, 0.3, -50, -0];

if(DETERMINED_START_STATE)
    current_state_num = GetStateNumber(starting_state, 0, X1_bound_vec,
X2_bound_vec, X3_bound_vec, X4_bound_vec, state_dimensions);
else
    current_state_num = GetStateNumber(states_in, 0, X1_bound_vec,
X2_bound_vec, X3_bound_vec, X4_bound_vec, state_dimensions);
end


step_count = 0;
step_num_tracker = -1*zeros(1, NUM_EPISODES);
walk_num = 1;

clock

total_distance_traveled = 0;
total_energy_consumed = 0;
total_time_taken = 0;


counter = 1;
for ep_num = 1:NUM_EPISODES
    [action_index, tradeoff_index, action_value] =
SelectHeuristicAction(current_state_num, state_value_vector,
stochastic_transition_database, prob_distribution,
master_dynamics_database, master_energy_database,
master_distance_database, master_time_database, num_actions,
num_weights);

    %     current_state_num

    state_value_vector(current_state_num) = action_value;
    %     action_index

    [current_state_num, possible_state_transitions, energy_expended,
distance_stepped, time_taken] = TakeHeuristicAction(action_index,
tradeoff_index, current_state_num, master_dynamics_database,
master_energy_database,  master_distance_database,  master_time_database,
cum_probs);

    total_distance_traveled = total_distance_traveled +
distance_stepped;
    total_energy_consumed = total_energy_consumed + energy_expended;
    total_time_taken = total_time_taken + time_taken;

    %     current_state_num
    %     possible_state_transitions
    %     pause(0.1)
```

```
    %     current_state_num
    %     pause(1)

    if(current_state_num == 1 || (USE_RANDOM_RESTART &&
(RANDOM_RESTART_EVERY < total_distance_traveled)))
        step_num_tracker(walk_num) = step_count;
        %         disp([num2str(step_count), ' step walk'])
        %         current_state_num = GetStateNumber(states_in, 0,
X1_bound_vec,
        %         X2_bound_vec, X3_bound_vec, X4_bound_vec,
state_dimensions);



        if(total_distance_traveled == 0 || total_energy_consumed == 0
|| total_time_taken == 0)
            disp('No steps taken this walk')
        else
            step_count
            total_distance_traveled
            specific_cost_of_transport =
total_energy_consumed/total_distance_traveled/3/9.81
            average_walk_speed =
total_distance_traveled/total_time_taken
        end

        if(counter <= max_state_num && STATE_INITIALIZATION_ON)
            current_state_num = counter;
            counter = counter + 1;

        elseif(DETERMINED_START_STATE)
            current_state_num = GetStateNumber(starting_state, 0,
X1_bound_vec, X2_bound_vec, X3_bound_vec, X4_bound_vec,
state_dimensions);
        else
            current_state_num = ceil(rand*max_state_num);
        end

        %         current_state_num

        %         current_state_num = find(min(state_value_vector),1)
        walk_num = walk_num + 1;
        step_count = 0;
        total_distance_traveled = 0;
        total_energy_consumed = 0;
        total_time_taken = 0;

    else
        step_count = step_count+1;
    end

    if(mod(ep_num,100000) == 0)
```

```
        disp(num2str(ep_num))
        min_value = min(state_value_vector)
        find(min(state_value_vector) == state_value_vector,5)

        if(total_distance_traveled == 0 || total_energy_consumed == 0
|| total_time_taken == 0)
            disp('No steps taken this walk')
        else
%             step_count
%             total_distance_traveled
            specific_cost_of_transport =
total_energy_consumed/total_distance_traveled/3/9.81
            average_walk_speed =
total_distance_traveled/total_time_taken
        end

        hist(state_value_vector, linspace(-10,0,11))
        axis([-11 1 0 30000])
        pause(0.05)
    end

end

clock

save run_all_vars

plot(step_num_tracker)

pause(0.1)

disp('Computing Markov Decision Process Matrix')
ComputeMDP
```

SelectHeuristicAction.m

```
function [action_index, tradeoff_index, action_value] =
SelectHeuristicAction(current_state_num, state_value_vector,
stochastic_transition_database, prob_distribution,
master_dynamics_database, master_energy_database,
master_distance_database, master_time_database, num_actions,
num_weights)

gamma = 0.9;

SUSTAINED_WALK_TRADEOFF_WEIGHTS =
evalin('base','SUSTAINED_WALK_TRADEOFF_WEIGHTS');

action_value_vector = zeros(num_weights, num_actions);

% min_action_value =
gamma*stochastic_transition_database{1}(current_state_num,:)*state_val
ue_vector + -1*(current_state_num > 1);

min_action_value = 0;
min_action_index = 1;
min_tradeoff_index = 1;

for tradeoff_index = 1:num_weights
    for m = 1:num_actions

        %         current_action_value =
gamma*stochastic_transition_database{tradeoff_index,m}(current_state_n
um,:)*state_value_vector +
(stochastic_transition_database{tradeoff_index,m}(current_state_num,1)
- 1);

        %  Robustness contribution
        future_value =
gamma*stochastic_transition_database{tradeoff_index,m}(current_state_n
um,:)*state_value_vector;

        robustness_action_value =
(stochastic_transition_database{tradeoff_index,m}(current_state_num,1)
- 1)*master_distance_database{tradeoff_index,m}(current_state_num);

        %  Energy contribution
        if(master_energy_database{tradeoff_index,m}(current_state_num)
~= 0)
            energy_action_value =
(stochastic_transition_database{tradeoff_index,m}(current_state_num,1)
-
1)*master_distance_database{tradeoff_index,m}(current_state_num)/maste
r_energy_database{tradeoff_index,m}(current_state_num);
        else
```

```
            energy_action_value = 0;
        end


        %  Speed contribution
        if(master_time_database{tradeoff_index,m}(current_state_num)
~= 0)

            %0.0001 added to avoid divide by zero error

%
size((master_dynamics_database{tradeoff_index,m}(current_state_num,:)
~= 1))
%
size((master_time_database{tradeoff_index,m}(current_state_num,:)+0.00
01))
%
size(master_distance_database{tradeoff_index,m}(current_state_num))
%
size(stochastic_transition_database{tradeoff_index,m}(current_state_nu
m,:))

            speed_action_value = sum(-
1*(master_dynamics_database{tradeoff_index,m}(current_state_num,:) ~=
1)./(master_time_database{tradeoff_index,m}(current_state_num,:)+0.000
1).*master_distance_database{tradeoff_index,m}(current_state_num).*(pr
ob_distribution));

        else
            speed_action_value = 0;
        end

        current_action_value = future_value +
SUSTAINED_WALK_TRADEOFF_WEIGHTS(1)*robustness_action_value +
SUSTAINED_WALK_TRADEOFF_WEIGHTS(2)*energy_action_value +
SUSTAINED_WALK_TRADEOFF_WEIGHTS(3)*speed_action_value;

        action_value_vector(tradeoff_index, m) = current_action_value;

        if(current_action_value <= min(min(action_value_vector)))
            min_action_value = current_action_value;

            min_action_index = m;
            min_tradeoff_index = tradeoff_index;
        end
    end
end

% current_state_num
% action_value_vector
action_index = min_action_index;
tradeoff_index = min_tradeoff_index;
```

```
action_value = min_action_value;

% pause
```

StepToStepGetEnergy.m

```
function [states_out, is_fallen, distance_traversed, time_elapsed,
energy_consumed, y_converged, min_stance_angvel] =
StepToStepGetEnergy(state_in, action, delta_terrain_vec,
threshold_values)

% Initialize Parameters

angle_des = action(1);
PGain = action(3);
DGain = action(4);
ACTIVATE_AT_LEG_CROSS = action(5);

% threshold_values format
% threshold_values = [minimum acceptable angle error (deg), minimum
% acceptable angular velocity error (deg/sec)]

assignin('base','angle_des',angle_des);
assignin('base','PGain',PGain);
assignin('base','DGain',DGain);
assignin('base','ACTIVATE_AT_LEG_CROSS',ACTIVATE_AT_LEG_CROSS);
assignin('base','terrain_height_vector',0);

InitStepToStepParams(state_in)

applied_impulse = action(2);

assignin('base','t_max',0);
evalin('base','BipedOneStepEOM');

StLCG_position = evalin('base','StLCG_position');
MBCG_position = evalin('base','MBCG_position');
SwLeg_position = evalin('base','SwLeg_position');
SwLCG_position = evalin('base','SwLCG_position');
MBCG_velocity = evalin('base','MBCG_velocity');
StLCG_velocity = evalin('base','StLCG_velocity');
Base_angvel = evalin('base','Base_angvel');
SwLCG_velocity = evalin('base','SwLCG_velocity');
SwLeg_angvel = evalin('base','SwLeg_angvel');
StLeg_angvel = evalin('base','StLeg_angvel');
MBCG_angvel = evalin('base','MBCG_angvel');
StLCG_angvel = evalin('base','StLCG_angvel');
SwLeg_angle = evalin('base','SwLeg_angle');
Base_angle = evalin('base','Base_angle');

final_index = 1;

EnergyComputationOneStep

SwitchStanceOneStepEOM
```

```
assignin('base','IC_StLeg_position',IC_StLeg_position);
assignin('base','IC_Base_angle',IC_Base_angle);
assignin('base','IC_StLeg_angle',IC_StLeg_angle);
assignin('base','IC_SwLeg_angle',IC_SwLeg_angle);
assignin('base','IC_StLeg_angvel',IC_StLeg_angvel);
assignin('base','IC_SwLeg_angvel',IC_SwLeg_angvel);
assignin('base','KE_vec',KE_vec);
assignin('base','PE_vec',PE_vec);

angle_des = action(1);
PGain = action(3);
DGain = action(4);

assignin('base','terrain_height_vector',delta_terrain_vec);

assignin('base','angle_des',angle_des);
assignin('base','PGain',PGain);
assignin('base','DGain',DGain);
assignin('base','ACTIVATE_AT_LEG_CROSS',ACTIVATE_AT_LEG_CROSS);

assignin('base','t_max',0.75);
evalin('base','BipedOneStepEOM');

StLCG_position = evalin('base','StLCG_position');
MBCG_position = evalin('base','MBCG_position');
SwLeg_position = evalin('base','SwLeg_position');
SwLCG_position = evalin('base','SwLCG_position');
MBCG_velocity = evalin('base','MBCG_velocity');
StLCG_velocity = evalin('base','StLCG_velocity');
Base_angvel = evalin('base','Base_angvel');
SwLCG_velocity = evalin('base','SwLCG_velocity');
SwLeg_angvel = evalin('base','SwLeg_angvel');
StLeg_angvel = evalin('base','StLeg_angvel');
MBCG_angvel = evalin('base','MBCG_angvel');
StLCG_angvel = evalin('base','StLCG_angvel');
SwLeg_angle = evalin('base','SwLeg_angle');
Base_angle = evalin('base','Base_angle');

SwLeg_angvel_joint = evalin('base','SwLeg_angvel_joint');

final_index = evalin('base','final_index');

assignin('base','final_index',final_index - 1);

EnergyComputationOneStep

impulse_work = KE_vec(2) - KE_vec(1);
collision_work = KE_vec(3) - KE_vec(2);
hip_actuator_work = energy_net;
hip_actuator_energy_added = energy_added;
gravity_work = -1*PE_delta;
hip_actuator_work = hip_actuator_work - gravity_work;
```

```
total_work = impulse_work + collision_work + hip_actuator_work +
gravity_work;
% added_kinetic_energy = KE_vec(1) - KE_vec_prev(1)

% energy_step_to_step = KE_vec(1)+PE_vec(1)-KE_vec_prev(1)-
PE_vec_prev(1);


%% DEBUG

% for(m = 1:length(HitCheck(1,:)))
%     index = find(HitCheck(:,m),1);
%     if(isempty(index))
%         states_out(:,m) = [0;0;0;0];
%         is_fallen(1,m) = 1;
%
%         distance_traversed(1,m) = 0;
%         time_elapsed(1,m) = 0;
%
%         %         energy_consumed(1,m) = impulse_work +
hip_actuator_energy_added;
%         energy_consumed(1,m) = hip_actuator_energy_added;
%
%         controller_error(1:2,m) = [angle_error(index);
angle_vel_error(index)];
%     else
%         X1out = IC_StLeg_position(2) - SwLeg_position(index,2);
%         X2out = SwLeg_position(index,1) - IC_StLeg_position(1);
%         X3out = Base_angvel(index);
%         X4out = SwLeg_angvel_joint(index);
%
%         distance_traversed(1,m) = MBCG_position(index,1) -
MBCG_position(1,1);
%         time_elapsed(1,m) = sim_time(index) - sim_time(1);
%
%         states_out(:,m) = [X1out; X2out; X3out; X4out];
%         is_fallen(1,m) = 0;
%         %         energy_consumed(1,m) = impulse_work +
hip_actuator_energy_added;
%         energy_consumed(1,m) = hip_actuator_energy_added;
%
%         controller_error(1:2,m) = [angle_error(index);
angle_vel_error(index)];
%     end
% end
%
% final_index = length(Base_angle)

final_index = evalin('base','final_index');
```

```
X1out = 0;
X2out = 0;
X3out = 0;
X4out = 0;

% X1out = IC_StLeg_position(2) - SwLeg_position(final_index,2);
% X2out = SwLeg_position(final_index,1) - IC_StLeg_position(1);
% X3out = Base_angvel(final_index);
% X4out = SwLeg_angvel_joint(final_index);

states_out = [X1out; X2out; X3out; X4out];

% size_SwLeg_position = evalin('base','size(SwLeg_position)');

% states_out = 0;
is_fallen = (final_index==evalin('base','length(HitCheck_raw)-1'));
% distance_traversed = evalin('base','SwLeg_position(final_index(1),1)
- IC_StLeg_position(1)');
% time_elapsed = evalin('base','final_index(1)*dt');


distance_traversed = 0;
time_elapsed = 0;

energy_consumed = energy_added + abs(energy_dissipated) + impulse_work;

% energy_added
% energy_dissipated
% impulse_work

% meet_threshold_vec = (abs(angle_des +
evalin('base','interleg_angle(1:final_index(1))')) <
threshold_values(1)).*(abs(evalin('base','interleg_velocity(1:final_in
dex(1))')) < threshold_values(2));
% index_meet_threshold = find(meet_threshold_vec);
%
% Swing_ypos = evalin('base','SwLeg_position(:,2)');


% terrain_cross = (evalin('base','SwLeg_position(1:final_index(1))')

% if(~isempty(index_meet_threshold))
%     y_converged = Swing_ypos(index_meet_threshold(1));
% else
%     is_fallen = 1;
%     y_converged = min(Swing_ypos(1:final_index));
% end
%
% min_stance_angvel = min(-1*Base_angvel(1:(numel(Base_angvel)-1)));

y_converged = 0;
min_stance_angvel = 0;
```

StepToStepGOA.m

```
function [states_out, is_fallen, distance_traversed, time_elapsed,
energy_consumed, y_converged, min_stance_angvel] =
StepToStepGOA(state_in, action, delta_terrain_vec, threshold_values)

% Initialize Parameters

angle_des = action(1);
PGain = action(3);
DGain = action(4);
ACTIVATE_AT_LEG_CROSS = action(5);

% threshold_values format
% threshold_values = [minimum acceptable angle error (deg), minimum
% acceptable angular velocity error (deg/sec)]

assignin('base','angle_des',angle_des);
assignin('base','PGain',PGain);
assignin('base','DGain',DGain);
assignin('base','ACTIVATE_AT_LEG_CROSS',ACTIVATE_AT_LEG_CROSS);
assignin('base','terrain_height_vector',0);

InitStepToStepParams(state_in)

applied_impulse = action(2);

assignin('base','t_max',0);
evalin('base','BipedOneStepEOM');

StLCG_position = evalin('base','StLCG_position');
MBCG_position = evalin('base','MBCG_position');
SwLeg_position = evalin('base','SwLeg_position');
SwLCG_position = evalin('base','SwLCG_position');
MBCG_velocity = evalin('base','MBCG_velocity');
StLCG_velocity = evalin('base','StLCG_velocity');
Base_angvel = evalin('base','Base_angvel');
SwLCG_velocity = evalin('base','SwLCG_velocity');
SwLeg_angvel = evalin('base','SwLeg_angvel');
StLeg_angvel = evalin('base','StLeg_angvel');
MBCG_angvel = evalin('base','MBCG_angvel');
StLCG_angvel = evalin('base','StLCG_angvel');
SwLeg_angle = evalin('base','SwLeg_angle');
Base_angle = evalin('base','Base_angle');

final_index = 1;

EnergyComputationOneStep

SwitchStanceOneStepEOM
```

```
assignin('base','IC_StLeg_position',IC_StLeg_position);
assignin('base','IC_Base_angle',IC_Base_angle);
assignin('base','IC_StLeg_angle',IC_StLeg_angle);
assignin('base','IC_SwLeg_angle',IC_SwLeg_angle);
assignin('base','IC_StLeg_angvel',IC_StLeg_angvel);
assignin('base','IC_SwLeg_angvel',IC_SwLeg_angvel);
assignin('base','KE_vec',KE_vec);
assignin('base','PE_vec',PE_vec);

angle_des = action(1);
PGain = action(3);
DGain = action(4);

assignin('base','terrain_height_vector',delta_terrain_vec);

assignin('base','angle_des',angle_des);
assignin('base','PGain',PGain);
assignin('base','DGain',DGain);
assignin('base','ACTIVATE_AT_LEG_CROSS',ACTIVATE_AT_LEG_CROSS);

assignin('base','t_max',0.75);
evalin('base','BipedOneStepEOM');

StLCG_position = evalin('base','StLCG_position');
MBCG_position = evalin('base','MBCG_position');
SwLeg_position = evalin('base','SwLeg_position');
SwLCG_position = evalin('base','SwLCG_position');
MBCG_velocity = evalin('base','MBCG_velocity');
StLCG_velocity = evalin('base','StLCG_velocity');
Base_angvel = evalin('base','Base_angvel');
SwLCG_velocity = evalin('base','SwLCG_velocity');
SwLeg_angvel = evalin('base','SwLeg_angvel');
StLeg_angvel = evalin('base','StLeg_angvel');
MBCG_angvel = evalin('base','MBCG_angvel');
StLCG_angvel = evalin('base','StLCG_angvel');
SwLeg_angle = evalin('base','SwLeg_angle');
Base_angle = evalin('base','Base_angle');

SwLeg_angvel_joint = evalin('base','SwLeg_angvel_joint');

final_index = evalin('base','final_index');

assignin('base','final_index',final_index - 1);

EnergyComputationOneStep

impulse_work = KE_vec(2) - KE_vec(1);
collision_work = KE_vec(3) - KE_vec(2);
hip_actuator_work = energy_net;
hip_actuator_energy_added = energy_added;
gravity_work = -1*PE_delta;
```

```
hip_actuator_work = hip_actuator_work - gravity_work;

total_work = impulse_work + collision_work + hip_actuator_work +
gravity_work;
% added_kinetic_energy = KE_vec(1) - KE_vec_prev(1)

% energy_step_to_step = KE_vec(1)+PE_vec(1)-KE_vec_prev(1)-
PE_vec_prev(1);


%% DEBUG

% for(m = 1:length(HitCheck(1,:)))
%     index = find(HitCheck(:,m),1);
%     if(isempty(index))
%         states_out(:,m) = [0;0;0;0];
%         is_fallen(1,m) = 1;
%
%         distance_traversed(1,m) = 0;
%         time_elapsed(1,m) = 0;
%
%         %         energy_consumed(1,m) = impulse_work +
hip_actuator_energy_added;
%         energy_consumed(1,m) = hip_actuator_energy_added;
%
%         controller_error(1:2,m) = [angle_error(index);
angle_vel_error(index)];
%     else
%         X1out = IC_StLeg_position(2) - SwLeg_position(index,2);
%         X2out = SwLeg_position(index,1) - IC_StLeg_position(1);
%         X3out = Base_angvel(index);
%         X4out = SwLeg_angvel_joint(index);
%
%         distance_traversed(1,m) = MBCG_position(index,1) -
MBCG_position(1,1);
%         time_elapsed(1,m) = sim_time(index) - sim_time(1);
%
%         states_out(:,m) = [X1out; X2out; X3out; X4out];
%         is_fallen(1,m) = 0;
%         %         energy_consumed(1,m) = impulse_work +
hip_actuator_energy_added;
%         energy_consumed(1,m) = hip_actuator_energy_added;
%
%         controller_error(1:2,m) = [angle_error(index);
angle_vel_error(index)];
%     end
% end
%
% final_index = length(Base_angle)

final_index = evalin('base','final_index');
```

```
X1out = IC_StLeg_position(2) - SwLeg_position(final_index,2);
X2out = SwLeg_position(final_index,1) - IC_StLeg_position(1);
X3out = Base_angvel(final_index);
X4out = SwLeg_angvel_joint(final_index);

states_out = [X1out; X2out; X3out; X4out];

% size_SwLeg_position = evalin('base','size(SwLeg_position)');

% states_out = 0;
is_fallen = (final_index==evalin('base','length(HitCheck_raw)-1'));
distance_traversed = evalin('base','SwLeg_position(final_index(1),1) -
IC_StLeg_position(1)');
time_elapsed = evalin('base','final_index(1)*dt');
energy_consumed = energy_added + abs(energy_dissipated) + impulse_work;

% energy_added
% energy_dissipated
% impulse_work

meet_threshold_vec = (abs(angle_des +
evalin('base','interleg_angle(1:final_index(1))')) <
threshold_values(1)).*(abs(evalin('base','interleg_velocity(1:final_in
dex(1))')) < threshold_values(2));
index_meet_threshold = find(meet_threshold_vec);

Swing_ypos = evalin('base','SwLeg_position(:,2)');


% terrain_cross = (evalin('base','SwLeg_position(1:final_index(1))')

if(~isempty(index_meet_threshold))
    y_converged = Swing_ypos(index_meet_threshold(1));
else
    is_fallen = 1;
    y_converged = min(Swing_ypos(1:final_index));
end

min_stance_angvel = min(-1*Base_angvel(1:(numel(Base_angvel)-1)));
```

StepToStepTFarchive.m

```
function [states_out, is_fallen, distance_traversed, time_elapsed,
energy_consumed] = StepToStepTFarchive(state_in, action,
delta_terrain_vec)

% Initialize Parameters

angle_des = 0;
PGain = 0;
DGain = 0;
ACTIVATE_AT_LEG_CROSS = 1;

assignin('base','angle_des',angle_des);
assignin('base','PGain',PGain);
assignin('base','DGain',DGain);
assignin('base','ACTIVATE_AT_LEG_CROSS',ACTIVATE_AT_LEG_CROSS);
assignin('base','terrain_height_vector',0);

InitStepToStepParams(state_in)

% pause

applied_impulse = action(2);

% sim('BipedSimOneStep',0)
assignin('base','t_max',0);
evalin('base','BipedOneStepEOM');

% pause

% assignin('base','StLCG_position',StLCG_position);
% assignin('base','MBCG_position',MBCG_position);
% assignin('base','SwLeg_position',SwLeg_position);
% assignin('base','SwLCG_position',SwLCG_position);
%
% assignin('base','MBCG_velocity',MBCG_velocity);
% assignin('base','StLCG_velocity',StLCG_velocity);
% assignin('base','Base_angvel',Base_angvel);
% assignin('base','SwLCG_velocity',SwLCG_velocity);
% assignin('base','SwLeg_angvel',SwLeg_angvel);
% assignin('base','StLeg_angvel',StLeg_angvel);

StLCG_position = evalin('base','StLCG_position');
MBCG_position = evalin('base','MBCG_position');
SwLeg_position = evalin('base','SwLeg_position');
SwLCG_position = evalin('base','SwLCG_position');
MBCG_velocity = evalin('base','MBCG_velocity');
StLCG_velocity = evalin('base','StLCG_velocity');
Base_angvel = evalin('base','Base_angvel');
SwLCG_velocity = evalin('base','SwLCG_velocity');
```

```
SwLeg_angvel = evalin('base','SwLeg_angvel');
StLeg_angvel = evalin('base','StLeg_angvel');
MBCG_angvel = evalin('base','MBCG_angvel');
StLCG_angvel = evalin('base','StLCG_angvel');
SwLeg_angle = evalin('base','SwLeg_angle');
Base_angle = evalin('base','Base_angle');

final_index = 1;

% pause

EnergyComputationOneStep

% pause

SwitchStanceOneStepEOM

% pause

assignin('base','IC_StLeg_position',IC_StLeg_position);
assignin('base','IC_Base_angle',IC_Base_angle);
assignin('base','IC_StLeg_angle',IC_StLeg_angle);
assignin('base','IC_SwLeg_angle',IC_SwLeg_angle);
assignin('base','IC_StLeg_angvel',IC_StLeg_angvel);
assignin('base','IC_SwLeg_angvel',IC_SwLeg_angvel);
assignin('base','KE_vec',KE_vec);
assignin('base','PE_vec',PE_vec);

% angle_des = action(1);
% PGain = -100;
% DGain = -10;
angle_des = action(1);
PGain = action(3);
DGain = action(4);

assignin('base','terrain_height_vector',delta_terrain_vec);

assignin('base','angle_des',angle_des);
assignin('base','PGain',PGain);
assignin('base','DGain',DGain);
assignin('base','ACTIVATE_AT_LEG_CROSS',ACTIVATE_AT_LEG_CROSS);

% sim('BipedSimOneStep')
assignin('base','t_max',1.5);
evalin('base','BipedOneStepEOM');

% assignin('base','StLCG_position',StLCG_position);
% assignin('base','MBCG_position',MBCG_position);
% assignin('base','SwLeg_position',SwLeg_position);
% assignin('base','SwLCG_position',SwLCG_position);
%
% assignin('base','MBCG_velocity',MBCG_velocity);
```

```
% assignin('base','StLCG_velocity',StLCG_velocity);
% assignin('base','Base_angvel',Base_angvel);
% assignin('base','SwLCG_velocity',SwLCG_velocity);
% assignin('base','SwLeg_angvel',SwLeg_angvel);
% assignin('base','StLeg_angvel',StLeg_angvel);

StLCG_position = evalin('base','StLCG_position');
MBCG_position = evalin('base','MBCG_position');
SwLeg_position = evalin('base','SwLeg_position');
SwLCG_position = evalin('base','SwLCG_position');
MBCG_velocity = evalin('base','MBCG_velocity');
StLCG_velocity = evalin('base','StLCG_velocity');
Base_angvel = evalin('base','Base_angvel');
SwLCG_velocity = evalin('base','SwLCG_velocity');
SwLeg_angvel = evalin('base','SwLeg_angvel');
StLeg_angvel = evalin('base','StLeg_angvel');
MBCG_angvel = evalin('base','MBCG_angvel');
StLCG_angvel = evalin('base','StLCG_angvel');
SwLeg_angle = evalin('base','SwLeg_angle');
Base_angle = evalin('base','Base_angle');

final_index = evalin('base','final_index');

%DEBUG
% EnergyComputationOneStep
%/DEBUG

impulse_work = KE_vec(2) - KE_vec(1);
collision_work = KE_vec(3) - KE_vec(2);
hip_actuator_work = energy_net;
hip_actuator_energy_added = energy_added;
gravity_work = -1*PE_delta;
hip_actuator_work = hip_actuator_work - gravity_work;

total_work = impulse_work + collision_work + hip_actuator_work +
gravity_work;
% added_kinetic_energy = KE_vec(1) - KE_vec_prev(1)

% energy_step_to_step = KE_vec(1)+PE_vec(1)-KE_vec_prev(1)-
PE_vec_prev(1);


HitCheck = evalin('base','HitCheck');
dt = evalin('base','dt');
num_max = evalin('base','num_max');
interleg_velocity = evalin('base','interleg_velocity');

index_hit_list = mod(find(HitCheck),length(delta_terrain_vec));

HitList(index_hit_list + (index_hit_list ==
0).*length(delta_terrain_vec)) =
floor(find(HitCheck)/length(delta_terrain_vec));
```

```
HitList = num_max*(HitList > num_max) + HitList.*(HitList <= num_max);

X1 = IC_StLeg_position(2) - SwLeg_position(HitList,2);
X2 = SwLeg_position(HitList,1) - IC_StLeg_position(1);
X3 = Base_angvel(HitList);
X4 = interleg_velocity(HitList);

% interleg_velocity

states_out = [X1 X2 X3 X4];
is_fallen = (HitList == num_max);

% length(HitCheck(1,:))

% for m = 1:length(HitCheck(1,:))
%     index = find(HitCheck(:,m),1);
%
%     if(isempty(index))
%
% %         states_out
%
% %         states_out(:,m) = [0;0;0;0];
%         is_fallen(1,m) = 1;
%
%         distance_traversed(1,m) = 0;
%         time_elapsed(1,m) = 0;
%
%         energy_consumed(1,m) = 0;
%     else
%
%         index
%
%         X1out = IC_StLeg_position(2) - SwLeg_position(index,2)
%         X2out = SwLeg_position(index,1) - IC_StLeg_position(1)
%         X3out = Base_angvel(index)
%         X4out = SwLeg_angvel(index)
%
%         distance_traversed(1,m) = MBCG_position(index,1) -
MBCG_position(1,2);
%         %         time_elapsed(1,m) = sim_time(index) - sim_time(1);
%         time_elapsed(1,m) = index.*dt;
%
%         states_out
%
%         states_out(:,m) = [X1out; X2out; X3out; X4out];
%         is_fallen(1,m) = 0;
%         energy_consumed(1,m) = impulse_work +
hip_actuator_energy_added;
%     end
% end
```

```
% energy_consumed = 0;
% time_elapsed = 0;
% distance_traversed = 0;

% MBCG_position

final_index = evalin('base','final_index');

% is_fallen = (final_index==evalin('base','length(HitCheck_raw)-1'))

if(sum(is_fallen == 0) > 0)

%     distance_traversed =
evalin('base','SwLeg_position(mod(final_index(1),t_max/dt),1) -
SwLeg_position(1,1)');

    distance_traversed = states_out(2);

%     time_elapsed = evalin('base','mod(final_index(1),t_max/dt)*dt');

    time_elapsed = evalin('base','dt')*HitList.*(1-is_fallen);

    energy_consumed = energy_added +
abs(energy_dissipated)+impulse_work;


%     Energy Computation fails to calculate energy added and
dissipated: use StepToStepGOA for Energy
%     impulse_work
%     energy_added
%     energy_dissipated

    controller_p_error = 0;
    controller_d_error = 0;

%     controller_p_error =
abs(angle_des+evalin('base','interleg_angle(mod(final_index(1),t_max/d
t))'));
%     controller_d_error =
abs(evalin('base','interleg_velocity(mod(final_index(1),t_max/dt))'));


else
    energy_consumed = 0;
    time_elapsed = 0;
    distance_traversed = 0;
end
```

SwitchStanceOneStepEOM.m

```
% SwitchStanceOneStep

% StLCG_position = evalin('base','StLCG_position');
% SwLCG_position = evalin('base','SwLCG_position');
% MBCG_position = evalin('base','MBCG_position');

[omega2, omega3, KE_vec, PE_vec] =
ImpulseComputationEOM(applied_impulse);

%Stance Leg Parameters
% IC_StLeg_position = [SwLeg_position(final_index, 1),
SwLeg_position(final_index, 2), 0];
IC_StLeg_position = [0, 0, 0];
% IC_Base_angle = -SwLeg_angle(final_index)
% IC_StLeg_angle = 0;
% IC_SwLeg_angle = -Base_angle(final_index)-90

IC_Base_angle = -SwLeg_angle(final_index);
IC_StLeg_angle = 0;
IC_SwLeg_angle = -Base_angle(final_index);

IC_StLeg_angvel = omega3;  %negative value indicates "forward" swing
IC_SwLeg_angvel = omega2;  %negative value indicates "forward" fall
Stance_position = SwLeg_position(final_index, 1);
```

TakeHeuristicAction.m

```
function [new_state_num, possible_state_transitions, energy_expended,
distance_stepped, time_taken] = TakeHeuristicAction(action_index,
tradeoff_index, current_state_num, master_dynamics_table,
master_energy_database,  master_distance_database,  master_time_database,
cum_probs)

rand_num = rand;

resulting_states = master_dynamics_table{tradeoff_index,
action_index}(current_state_num,:);
resulting_energies = master_energy_database{tradeoff_index,
action_index}(current_state_num);
resulting_distances = master_distance_database{tradeoff_index,
action_index}(current_state_num);
resulting_times = master_time_database{tradeoff_index,
action_index}(current_state_num,:);

result_index = find(rand_num < cum_probs, 1);

new_state_num = resulting_states(result_index);

possible_state_transitions = resulting_states;

energy_expended = resulting_energies; %ENERGY is assumed constant over
various terrain heights
distance_stepped = resulting_distances; %DISTANCE is assumed constant
over various terrain heights
time_taken = resulting_times(result_index);
```

Approximate Optimal Robustness Code

Step 1: Run "GenerateMasterDynamicsTable.m"
Step 2: Run "RunStochasticSetup.m"

BipedOneStepEOM.m

See page 109.

ComputeBestActionTransitions.m

See page 115.

ComputeMDP.m

See page 121.

ComputeProbDistribution.m

See page 122.

ContinueStochastic.m

```
% ContinueStochastic

% clc
% clear
close all

load dynamics_database

NUM_EPISODES = 1e5;

num_actions = length(alpha_vec);

[X1_bound_vec, X2_bound_vec, X3_bound_vec, X4_bound_vec, max_state_num]
= GetStateBoundaryVectors(X1_vec, X2_vec, X3_vec, X4_vec);

state_dimensions = [length(X1_vec), length(X2_vec), length(X3_vec),
length(X4_vec)];

prob_distribution = [0 0 0.0028 0.0092 0.0276 0.0657 0.1212 0.1743
0.1984 ...
    0.1743 0.1212 0.0657 0.0276 0.0092 0.0028 0 0];
cum_probs = cumsum(prob_distribution);
```

```
stochastic_transition_database =
GenerateStochasticTransitionTable(master_dynamics_database,
prob_distribution, max_state_num, num_actions);

disp('Stochastic Generation Complete!  Learning Commencing...')

% state_value_vector = -1*ones(max_state_num, 1);
% state_value_vector(1) = 0;

states_in = [-0.02, 0.3, -50, -0];

current_state_num = GetStateNumber(states_in, 0, X1_bound_vec,
X2_bound_vec, X3_bound_vec, X4_bound_vec, state_dimensions);

step_count = 0;
step_num_tracker = -1*zeros(1, NUM_EPISODES);
walk_num = 1;

clock

counter = 1;
for ep_num = 1:NUM_EPISODES
    [action_index, action_value] = SelectAction(current_state_num,
state_value_vector, stochastic_transition_database, num_actions);

    %     current_state_num

    state_value_vector(current_state_num) = action_value;
    %     action_index

    [current_state_num, possible_state_transitions] =
TakeAction(action_index, current_state_num, master_dynamics_database,
cum_probs);

    %     current_state_num
    %     possible_state_transitions
    %     pause(0.1)

    %     current_state_num
    %     pause(1)

    if(current_state_num == 1)
        step_num_tracker(walk_num) = step_count;
        %         disp([num2str(step_count), ' step walk'])
        %         current_state_num = GetStateNumber(states_in, 0,
X1_bound_vec,
        %         X2_bound_vec, X3_bound_vec, X4_bound_vec,
state_dimensions);

        if(counter <= max_state_num)
            current_state_num = counter;
```

```
            counter = counter + 1;
        else
            current_state_num = ceil(rand*max_state_num);
        end

%         current_state_num

        %         current_state_num = find(min(state_value_vector),1)
        walk_num = walk_num + 1;
        step_count = 0;
    else
        step_count = step_count+1;
    end

    if(mod(ep_num,1000) == 0)
        disp(num2str(ep_num))
        min_value = min(state_value_vector)
        find(min(state_value_vector) == state_value_vector,5)
    end

end

clock

save run_all_vars

plot(step_num_tracker)

pause(0.1)

disp('Computing Markov Decision Process Matrix')
ComputeMDP
```

EnergyComputatioOneStep.m

See page 123

GenerateMasterDynamicsTable.m

```
%GenerateMasterDynamicsTable

%SAVE:
% [numX1, [X1vec]]
% [numX2, [X2vec]]
% [numX3, [X3vec]]
% [numX4, [X4vec]]
% [numDelta, [delta_vec]]
% [numAlpha, [alpha_vec]]
% [masterDynamicsTable (alpha slice 1)]
% [masterDynamicsTable (alpha slice 2)]
% ...
% [masterDynamicsTable (alpha slice numAlpha)]

% clc
clear
close all

try


    EMAIL_ALERT = 1;
%     [last_update_time, last_update_text] = CheckUpdateRequests;

    addpath P:\UrbanRobots\private\Hubicki\Simulation\2009-12\Tools

    initial_time = clock;
    if(initial_time(5) < 10)
        initial_minutes = ['0' num2str(initial_time(5))];
    else
        initial_minutes = [num2str(initial_time(5))];
    end
    initial_hours = num2str(initial_time(4));
    initial_time_readout = [initial_hours ':' initial_minutes];

    text_body = ['Greetings,' 10 'Your simulation has commenced,
beginning at ' initial_time_readout ' local machine time.' 10
'Regards,' 10 '- CodeBot'];
    if(EMAIL_ALERT)
        EmailSimulationUpdate(['Simulation Commenced at ',
initial_time_readout], text_body)
    end


    % Hubicki state space discretization
```

```
    X1_vec = [-0.1, -0.05, -0.04, -0.03:0.005:0.03, 0.04, 0.05, 0.1];
    X2_vec = [0.16:0.06:0.7];
    X3_vec = [-140:10:0];
    X4_vec = [-20:5:20];

    % delta_terrain_vec = [0.029 0.02 0.01 0.0 -0.01 -0.02 -0.029];
    delta_terrain_vec = [0.05 0.04 0.03:-0.005:-0.03 -0.04 -0.05];
    alpha_vec = linspace(15, 40, 9);
    impulse_value = 2;

    % X3_vec = [-2.1:0.1:-1.4, -1.25, -1.1];
    % X4_vec = [-1, -0.7, -0.5:0.25:0.75, 1.1, 1.5];

    numX1 = length(X1_vec);
    numX2 = length(X2_vec);
    numX3 = length(X3_vec);
    numX4 = length(X4_vec);

    state_dimensions = [numX1, numX2, numX3, numX4];

    [X1_bound_vec, X2_bound_vec, X3_bound_vec, X4_bound_vec,
max_state_num] = GetStateBoundaryVectors(X1_vec, X2_vec, X3_vec,
X4_vec);

    % state_in = [-0.02, 0.3, -50, -0];

    angle_ratio_vec = [-1,1];
    gain_schedule = [10, 10, 10];
    ratio_schedule = [1 1 1];
    action = [25 2 1 1 1];

    % time1 = clock;
    % [states_out, is_fallen, distance_traversed, time_elapsed,
energy_consumed] = StepToStepTFarchive(state_in, action,
delta_terrain_vec);
    time2 = clock;

    state_num = zeros(1,length(delta_terrain_vec));

    % for m = 1:length(delta_terrain_vec)
    %     state_num(m) = GetStateNumber(states_out(m,:), is_fallen(m),
X1_bound_vec, X2_bound_vec, X3_bound_vec, X4_bound_vec,
state_dimensions);
    % end

    time1 = clock;

    blank_trans_matrix = zeros(max_state_num,
length(delta_terrain_vec));
    blank_trans_matrix(1,:) = ones(1, length(delta_terrain_vec));

    master_dynamics_database = cell(1,length(alpha_vec));
```

```
    for p = 1:length(alpha_vec)

        action = [alpha_vec(p) impulse_value 1 1 1];
        new_trans_matrix = blank_trans_matrix;

        for q = 2:max_state_num

            index_vector = GetStateIndices(q, state_dimensions);
            state_in = [X1_vec(index_vector(1)) X2_vec(index_vector(2))
X3_vec(index_vector(3)) X4_vec(index_vector(4))];

            [states_out, is_fallen, distance_traversed, time_elapsed,
energy_consumed] = StepToStepTFarchive(state_in, action,
delta_terrain_vec);

            state_num = zeros(1,length(delta_terrain_vec));

            for n = 1:length(delta_terrain_vec)
                state_num(n) = GetStateNumber(states_out(n,:),
is_fallen(n), X1_bound_vec, X2_bound_vec, X3_bound_vec, X4_bound_vec,
state_dimensions);
            end

            new_trans_matrix(q,:) = state_num;

        end

        master_dynamics_database{p} = new_trans_matrix;

    end

    time2 = clock;

    time1
    time2

%     cellplot(master_dynamics_database)


    final_time = clock;
    if(final_time(5) < 10)
        final_minutes = ['0' num2str(final_time(5))];
    else
        final_minutes = [num2str(final_time(5))];
    end
    final_hours = num2str(final_time(4));
    final_time_readout = [final_hours ':' final_minutes];

    elapsed_time = final_time - initial_time;
    elapsed_hours = num2str(elapsed_time*[0 0 24 1 0 0]');
    elapsed_minutes = num2str(elapsed_time(5));
```

```
    text_body = ['Greetings,' 10 'Your simulation beginning at '
initial_time_readout ' has executed without error.' 10 'Total run time:
' elapsed_hours ' hours and ' elapsed_minutes ' minutes.' ...
        10 'Regards,' 10 '- CodeBot'];
    if(EMAIL_ALERT)
        EmailSimulationUpdate(['Simulation Complete at '
final_time_readout '!'], text_body)
    end

    save 'dynamics_database.mat' master_dynamics_database X1_vec
X2_vec X3_vec X4_vec delta_terrain_vec alpha_vec impulse_value

catch ME
    rep = getReport(ME)
    rep_email = getReport(ME, 'extended', 'hyperlinks', 'off');
    text_body = ['The error report was recorded as follows:' 10 ' ' 10
rep_email 10 ' ' 10 'Regards,' 10 '- CodeBot'];
    if(EMAIL_ALERT)
        EmailSimulationUpdate('Simulation Update: Untimely
Termination', text_body)
    end
end
```

GenerateStochasticTransitionTable.m

```
function [stochastic_transition_database] =
GenerateStochasticTransitionTable(master_dynamics_database,
prob_distribution, max_state_num, num_actions)

blank_transition_table = sparse(max_state_num, max_state_num);

num_delta = length(prob_distribution);

stochastic_transition_database = cell(1,num_actions);

for p = 1:num_actions

    %     clock

    current_transition_table = blank_transition_table;

    for q = 1:max_state_num
        for r = 1:num_delta

current_transition_table(q,master_dynamics_database{p}(q,r)) =
current_transition_table(q,master_dynamics_database{p}(q,r)) +
prob_distribution(r);

            %
current_transition_table(q,master_dynamics_database{p}(q,r))
            %             pause(0.1)

        end
%         if(sum(current_transition_table(q,:) > 0.999) == 0)
%             q
%             current_transition_table(q,:)
%             pause(0.1)
%         end


    end

    stochastic_transition_database{p} = current_transition_table;
end
```

GetControlTorque.m

See page 132

GetStateBoundaryVectors.m

See page 133

GetStateIndicies.m

See page 134

GetStateNumber.m

See page 135

InitialConditionTransformation.m

See page 142

InitStepToStepParams.m

See page 143

RunStochasticSetup.m

```
% RunStochasticSetup

clc
clear
close all

load dynamics_database

NUM_EPISODES = 1e6;

num_actions = length(alpha_vec);

terrain_sigma = 0.01;

[X1_bound_vec, X2_bound_vec, X3_bound_vec, X4_bound_vec, max_state_num]
= GetStateBoundaryVectors(X1_vec, X2_vec, X3_vec, X4_vec);

state_dimensions = [length(X1_vec), length(X2_vec), length(X3_vec),
length(X4_vec)];

% prob_distribution = [0 0 0 0 0 0.0060 0.0605 0.2420 0.3830 0.2420
0.0605 0.0060 0 0 0 0 0];
prob_distribution = ComputeProbDistribution(terrain_sigma, 0,
delta_terrain_vec);

cum_probs = cumsum(prob_distribution);

stochastic_transition_database =
GenerateStochasticTransitionTable(master_dynamics_database,
prob_distribution, max_state_num, num_actions);

disp('Stochastic Generation Complete!  Learning Commencing...')

state_value_vector = -1*zeros(max_state_num, 1);
state_value_vector(1) = 0;

states_in = [-0.02, 0.3, -50, -0];

current_state_num = GetStateNumber(states_in, 0, X1_bound_vec,
X2_bound_vec, X3_bound_vec, X4_bound_vec, state_dimensions);

step_count = 0;
step_num_tracker = -1*zeros(1, NUM_EPISODES);
walk_num = 1;

clock

counter = 1;
for ep_num = 1:NUM_EPISODES
```

```
    [action_index, action_value] = SelectAction(current_state_num,
state_value_vector, stochastic_transition_database, num_actions);

    %     current_state_num

    state_value_vector(current_state_num) = action_value;
    %     action_index

    [current_state_num, possible_state_transitions] =
TakeAction(action_index, current_state_num, master_dynamics_database,
cum_probs);

    %     current_state_num
    %     possible_state_transitions
    %     pause(0.1)

    %     current_state_num
    %     pause(1)

    if(current_state_num == 1)
        step_num_tracker(walk_num) = step_count;
        %         disp([num2str(step_count), ' step walk'])
        %         current_state_num = GetStateNumber(states_in, 0,
X1_bound_vec,
        %         X2_bound_vec, X3_bound_vec, X4_bound_vec,
state_dimensions);

        if(counter <= max_state_num)
            current_state_num = counter;
            counter = counter + 1;
        else
            current_state_num = ceil(rand*max_state_num);
        end

%         current_state_num

        %         current_state_num = find(min(state_value_vector),1)
        walk_num = walk_num + 1;
        step_count = 0;
    else
        step_count = step_count+1;
    end

    if(mod(ep_num,1000) == 0)
        disp(num2str(ep_num))
        min_value = min(state_value_vector)
        find(min(state_value_vector) == state_value_vector,5)

        hist(state_value_vector, linspace(-10,0,11))
        axis([-11 1 0 10000])
        pause(0.05)
    end
```

```
end

clock

save run_all_vars

plot(step_num_tracker)

pause(0.1)

disp('Computing Markov Decision Process Matrix')
ComputeMDP
```

SelectAction.m

```
function [action_index, action_value] = SelectAction(current_state_num,
state_value_vector, stochastic_transition_database, num_actions)

gamma = 0.9;

action_value_vector = zeros(1, num_actions);

min_action_value =
gamma*stochastic_transition_database{1}(current_state_num,:)*state_val
ue_vector + -1*(current_state_num > 1);
min_action_index = 1;

for m = 1:num_actions
    %     current_action_value =
gamma*stochastic_transition_database{m}(current_state_num,:)*state_val
ue_vector + -1*(current_state_num > 1);
    current_action_value =
gamma*stochastic_transition_database{m}(current_state_num,:)*state_val
ue_vector + (stochastic_transition_database{m}(current_state_num,1) -
1);
    action_value_vector(m) = current_action_value;

    if(current_action_value <= min(action_value_vector))
        min_action_value = current_action_value;
        min_action_index = m;
    end
end

% current_state_num
% action_value_vector
action_index = min_action_index;
action_value = min_action_value;

% pause
```

StepToStepStochastic.m

```
function [states_out, is_fallen, distance_traversed, time_elapsed,
energy_consumed, controller_p_error, controller_d_error] =
StepToStepStochastic(state_in, action, delta_terrain_vec)

% Initialize Parameters

angle_des = action(1);
PGain = action(3);
DGain = action(4);
ACTIVATE_AT_LEG_CROSS = action(5);

assignin('base','angle_des',angle_des);
assignin('base','PGain',PGain);
assignin('base','DGain',DGain);
assignin('base','ACTIVATE_AT_LEG_CROSS',ACTIVATE_AT_LEG_CROSS);
assignin('base','terrain_height_vector',0);

InitStepToStepParams(state_in)

applied_impulse = action(2);

% sim('BipedSimGainSchedule',0)
assignin('base','t_max',0);
evalin('base','BipedOneStepEOM');

% pause

% assignin('base','StLCG_position',StLCG_position);
% assignin('base','MBCG_position',MBCG_position);
% assignin('base','SwLeg_position',SwLeg_position);
% assignin('base','SwLCG_position',SwLCG_position);
%
% assignin('base','MBCG_velocity',MBCG_velocity);
% assignin('base','StLCG_velocity',StLCG_velocity);
% assignin('base','Base_angvel',Base_angvel);
% assignin('base','SwLCG_velocity',SwLCG_velocity);
% assignin('base','SwLeg_angvel',SwLeg_angvel);
% assignin('base','StLeg_angvel',StLeg_angvel);

StLCG_position = evalin('base','StLCG_position');
MBCG_position = evalin('base','MBCG_position');
SwLeg_position = evalin('base','SwLeg_position');
SwLCG_position = evalin('base','SwLCG_position');
MBCG_velocity = evalin('base','MBCG_velocity');
StLCG_velocity = evalin('base','StLCG_velocity');
Base_angvel = evalin('base','Base_angvel');
SwLCG_velocity = evalin('base','SwLCG_velocity');
SwLeg_angvel = evalin('base','SwLeg_angvel');
StLeg_angvel = evalin('base','StLeg_angvel');
MBCG_angvel = evalin('base','MBCG_angvel');
```

```
StLCG_angvel = evalin('base','StLCG_angvel');
SwLeg_angle = evalin('base','SwLeg_angle');
Base_angle = evalin('base','Base_angle');


% pause
final_index = 1;

EnergyComputationOneStep

SwitchStanceOneStepEOM

assignin('base','IC_StLeg_position',IC_StLeg_position);
assignin('base','IC_Base_angle',IC_Base_angle);
assignin('base','IC_StLeg_angle',IC_StLeg_angle);
assignin('base','IC_SwLeg_angle',IC_SwLeg_angle);
assignin('base','IC_StLeg_angvel',IC_StLeg_angvel);
assignin('base','IC_SwLeg_angvel',IC_SwLeg_angvel);
assignin('base','KE_vec',KE_vec);
assignin('base','PE_vec',PE_vec);

% KE_vec
% PE_vec

angle_des = action(1);
PGain = action(3);
DGain = action(4);

assignin('base','terrain_height_vector',delta_terrain_vec);

assignin('base','angle_des',angle_des);
assignin('base','PGain',PGain);
assignin('base','DGain',DGain);
assignin('base','ACTIVATE_AT_LEG_CROSS',ACTIVATE_AT_LEG_CROSS);


% myopts = simset('MinStep', evalin('base','min_step_size'));

% sim('BipedSimGainSchedule', 10)
assignin('base','t_max',1.5);
evalin('base','BipedOneStepEOM');
% sim('BipedSimOneStep', 10, myopts)

% assignin('base','StLCG_position',StLCG_position);
% assignin('base','MBCG_position',MBCG_position);
% assignin('base','SwLeg_position',SwLeg_position);
% assignin('base','SwLCG_position',SwLCG_position);
%
% assignin('base','MBCG_velocity',MBCG_velocity);
% assignin('base','StLCG_velocity',StLCG_velocity);
% assignin('base','Base_angvel',Base_angvel);
% assignin('base','SwLCG_velocity',SwLCG_velocity);
```

```
% assignin('base','SwLeg_angvel',SwLeg_angvel);
% assignin('base','StLeg_angvel',StLeg_angvel);

StLCG_position = evalin('base','StLCG_position');
MBCG_position = evalin('base','MBCG_position');
SwLeg_position = evalin('base','SwLeg_position');
SwLCG_position = evalin('base','SwLCG_position');
MBCG_velocity = evalin('base','MBCG_velocity');
StLCG_velocity = evalin('base','StLCG_velocity');
Base_angvel = evalin('base','Base_angvel');
SwLCG_velocity = evalin('base','SwLCG_velocity');
SwLeg_angvel = evalin('base','SwLeg_angvel');
StLeg_angvel = evalin('base','StLeg_angvel');
MBCG_angvel = evalin('base','MBCG_angvel');
StLCG_angvel = evalin('base','StLCG_angvel');
SwLeg_angle = evalin('base','SwLeg_angle');
Base_angle = evalin('base','Base_angle');

final_index = evalin('base','final_index');

EnergyComputationOneStep

impulse_work = KE_vec(2) - KE_vec(1);
collision_work = KE_vec(3) - KE_vec(2);
hip_actuator_work = energy_net;
hip_actuator_energy_added = energy_added;
gravity_work = -1*PE_delta;
hip_actuator_work = hip_actuator_work - gravity_work;

total_work = impulse_work + collision_work + hip_actuator_work +
gravity_work;
% added_kinetic_energy = KE_vec(1) - KE_vec_prev(1)

% energy_step_to_step = KE_vec(1)+PE_vec(1)-KE_vec_prev(1)-
PE_vec_prev(1);


%% DEBUG

% for(m = 1:length(HitCheck(1,:)))
%     index = find(HitCheck(:,m),1);
%     if(isempty(index))
%         states_out(:,m) = [0;0;0;0];
%         is_fallen(1,m) = 1;
%
%         distance_traversed(1,m) = 0;
%         time_elapsed(1,m) = 0;
%
%         %         energy_consumed(1,m) = impulse_work +
hip_actuator_energy_added;
%         energy_consumed(1,m) = hip_actuator_energy_added;
%
```

```
%         controller_error(1:2,m) = [angle_error(index);
angle_vel_error(index)];
%     else
%         X1out = IC_StLeg_position(2) - SwLeg_position(index,2);
%         X2out = SwLeg_position(index,1) - IC_StLeg_position(1);
%         X3out = Base_angvel(index);
%         X4out = SwLeg_angvel_joint(index);
%
%         distance_traversed(1,m) = MBCG_position(index,1) -
MBCG_position(1,1);
%         time_elapsed(1,m) = sim_time(index) - sim_time(1);
%
%         states_out(:,m) = [X1out; X2out; X3out; X4out];
%         is_fallen(1,m) = 0;
%         %         energy_consumed(1,m) = impulse_work +
hip_actuator_energy_added;
%         energy_consumed(1,m) = hip_actuator_energy_added;
%
%         controller_error(1:2,m) = [angle_error(index);
angle_vel_error(index)];
%     end
% end

% final_index = length(Base_angle)

final_index = evalin('base','final_index');
% size_SwLeg_position = evalin('base','size(SwLeg_position)');

states_out = 0;
is_fallen = (final_index==evalin('base','length(HitCheck_raw)-1'));
distance_traversed = evalin('base','SwLeg_position(final_index(1),1) -
SwLeg_position(1,1)');
time_elapsed = evalin('base','final_index(1)*dt');
energy_consumed = energy_added + abs(energy_dissipated)+impulse_work;
controller_p_error =
abs(angle_des+evalin('base','interleg_angle(final_index(1))'));
controller_d_error =
abs(evalin('base','interleg_velocity(final_index(1))'));
```

StepToStepTFarchive.m

```
function [states_out, is_fallen, distance_traversed, time_elapsed,
energy_consumed] = StepToStepTFarchive(state_in, action,
delta_terrain_vec)

% Initialize Parameters

angle_des = 0;
PGain = 0;
DGain = 0;
ACTIVATE_AT_LEG_CROSS = 1;

assignin('base','angle_des',angle_des);
assignin('base','PGain',PGain);
assignin('base','DGain',DGain);
assignin('base','ACTIVATE_AT_LEG_CROSS',ACTIVATE_AT_LEG_CROSS);
assignin('base','terrain_height_vector',0);

InitStepToStepParams(state_in)

% pause

applied_impulse = action(2);

% sim('BipedSimOneStep',0)
assignin('base','t_max',0);
evalin('base','BipedOneStepEOM');

% pause

% assignin('base','StLCG_position',StLCG_position);
% assignin('base','MBCG_position',MBCG_position);
% assignin('base','SwLeg_position',SwLeg_position);
% assignin('base','SwLCG_position',SwLCG_position);
%
% assignin('base','MBCG_velocity',MBCG_velocity);
% assignin('base','StLCG_velocity',StLCG_velocity);
% assignin('base','Base_angvel',Base_angvel);
% assignin('base','SwLCG_velocity',SwLCG_velocity);
% assignin('base','SwLeg_angvel',SwLeg_angvel);
% assignin('base','StLeg_angvel',StLeg_angvel);

StLCG_position = evalin('base','StLCG_position');
MBCG_position = evalin('base','MBCG_position');
SwLeg_position = evalin('base','SwLeg_position');
SwLCG_position = evalin('base','SwLCG_position');
MBCG_velocity = evalin('base','MBCG_velocity');
StLCG_velocity = evalin('base','StLCG_velocity');
Base_angvel = evalin('base','Base_angvel');
SwLCG_velocity = evalin('base','SwLCG_velocity');
SwLeg_angvel = evalin('base','SwLeg_angvel');
```

```
StLeg_angvel = evalin('base','StLeg_angvel');
MBCG_angvel = evalin('base','MBCG_angvel');
StLCG_angvel = evalin('base','StLCG_angvel');
SwLeg_angle = evalin('base','SwLeg_angle');
Base_angle = evalin('base','Base_angle');

final_index = 1;

% pause

EnergyComputationOneStep

% pause

SwitchStanceOneStepEOM

% pause

assignin('base','IC_StLeg_position',IC_StLeg_position);
assignin('base','IC_Base_angle',IC_Base_angle);
assignin('base','IC_StLeg_angle',IC_StLeg_angle);
assignin('base','IC_SwLeg_angle',IC_SwLeg_angle);
assignin('base','IC_StLeg_angvel',IC_StLeg_angvel);
assignin('base','IC_SwLeg_angvel',IC_SwLeg_angvel);
assignin('base','KE_vec',KE_vec);
assignin('base','PE_vec',PE_vec);

% angle_des = action(1);
% PGain = -100;
% DGain = -10;
angle_des = action(1);
PGain = action(3);
DGain = action(4);

assignin('base','terrain_height_vector',delta_terrain_vec);

assignin('base','angle_des',angle_des);
assignin('base','PGain',PGain);
assignin('base','DGain',DGain);
assignin('base','ACTIVATE_AT_LEG_CROSS',ACTIVATE_AT_LEG_CROSS);

% sim('BipedSimOneStep')
assignin('base','t_max',1.5);
evalin('base','BipedOneStepEOM');

% assignin('base','StLCG_position',StLCG_position);
% assignin('base','MBCG_position',MBCG_position);
% assignin('base','SwLeg_position',SwLeg_position);
% assignin('base','SwLCG_position',SwLCG_position);
%
% assignin('base','MBCG_velocity',MBCG_velocity);
% assignin('base','StLCG_velocity',StLCG_velocity);
```

```
% assignin('base','Base_angvel',Base_angvel);
% assignin('base','SwLCG_velocity',SwLCG_velocity);
% assignin('base','SwLeg_angvel',SwLeg_angvel);
% assignin('base','StLeg_angvel',StLeg_angvel);

StLCG_position = evalin('base','StLCG_position');
MBCG_position = evalin('base','MBCG_position');
SwLeg_position = evalin('base','SwLeg_position');
SwLCG_position = evalin('base','SwLCG_position');
MBCG_velocity = evalin('base','MBCG_velocity');
StLCG_velocity = evalin('base','StLCG_velocity');
Base_angvel = evalin('base','Base_angvel');
SwLCG_velocity = evalin('base','SwLCG_velocity');
SwLeg_angvel = evalin('base','SwLeg_angvel');
StLeg_angvel = evalin('base','StLeg_angvel');
MBCG_angvel = evalin('base','MBCG_angvel');
StLCG_angvel = evalin('base','StLCG_angvel');
SwLeg_angle = evalin('base','SwLeg_angle');
Base_angle = evalin('base','Base_angle');

final_index = evalin('base','final_index');

%DEBUG
% EnergyComputationOneStep
%/DEBUG

impulse_work = KE_vec(2) - KE_vec(1);
collision_work = KE_vec(3) - KE_vec(2);
hip_actuator_work = energy_net;
hip_actuator_energy_added = energy_added;
gravity_work = -1*PE_delta;
hip_actuator_work = hip_actuator_work - gravity_work;

total_work = impulse_work + collision_work + hip_actuator_work +
gravity_work;
% added_kinetic_energy = KE_vec(1) - KE_vec_prev(1)

% energy_step_to_step = KE_vec(1)+PE_vec(1)-KE_vec_prev(1)-
PE_vec_prev(1);


HitCheck = evalin('base','HitCheck');
dt = evalin('base','dt');
num_max = evalin('base','num_max');
interleg_velocity = evalin('base','interleg_velocity');

index_hit_list = mod(find(HitCheck),length(delta_terrain_vec));

HitList(index_hit_list + (index_hit_list ==
0).*length(delta_terrain_vec)) =
floor(find(HitCheck)/length(delta_terrain_vec));
```

```
HitList = num_max*(HitList > num_max) + HitList.*(HitList <= num_max);

X1 = IC_StLeg_position(2) - SwLeg_position(HitList,2);
X2 = SwLeg_position(HitList,1) - IC_StLeg_position(1);
X3 = Base_angvel(HitList);
X4 = interleg_velocity(HitList);

% interleg_velocity

states_out = [X1 X2 X3 X4];
is_fallen = (HitList == num_max);

% for(m = 1:length(HitCheck(1,:)))
%     index = find(HitCheck(:,m),1);
%     if(isempty(index))
%         states_out(:,m) = [0;0;0;0];
%         is_fallen(1,m) = 1;
%
%         distance_traversed(1,m) = 0;
%         time_elapsed(1,m) = 0;
%
%         energy_consumed(1,m) = 0;
%     else
%         X1out = IC_StLeg_position(2) - SwLeg_position(index,2);
%         X2out = SwLeg_position(index,1) - IC_StLeg_position(1);
%         X3out = Base_angvel(index);
%         X4out = SwLeg_angvel(index);
%
%         distance_traversed(1,m) = MBCG_position(index,1) -
MBCG_position(1,2);
%         %         time_elapsed(1,m) = sim_time(index) - sim_time(1);
%         time_elapsed(1,m) = index.*dt;
%
%         states_out(:,m) = [X1out; X2out; X3out; X4out];
%         is_fallen(1,m) = 0;
%         energy_consumed(1,m) = impulse_work +
hip_actuator_energy_added;
%     end
% end

energy_consumed = 0;
time_elapsed = 0;
distance_traversed = 0;

% MBCG_position

% final_index = evalin('base','final_index');
%
% is_fallen = (final_index==evalin('base','length(HitCheck_raw)-1'));
% distance_traversed = evalin('base','SwLeg_position(final_index(1),1)
- SwLeg_position(1,1)');
```

```
% time_elapsed = evalin('base','final_index(1)*dt');
% energy_consumed = energy_added + abs(energy_dissipated)+impulse_work;
% controller_p_error =
abs(angle_des+evalin('base','interleg_angle(final_index(1))'));
% controller_d_error =
abs(evalin('base','interleg_velocity(final_index(1))'));
```

StepToStepTFEOM.m

```
function [states_out, is_fallen, distance_traversed, time_elapsed,
energy_consumed, controller_p_error, controller_d_error] =
StepToStepTFEOM(state_in, action, delta_terrain_vec)

% Initialize Parameters

angle_des = action(1);
PGain = action(3);
DGain = action(4);
ACTIVATE_AT_LEG_CROSS = action(5);

assignin('base','angle_des',angle_des);
assignin('base','PGain',PGain);
assignin('base','DGain',DGain);
assignin('base','ACTIVATE_AT_LEG_CROSS',ACTIVATE_AT_LEG_CROSS);
assignin('base','terrain_height_vector',0);

InitStepToStepParams(state_in)

applied_impulse = action(2);

% sim('BipedSimGainSchedule',0)
assignin('base','t_max',0);
evalin('base','BipedOneStepEOM');

% pause

% assignin('base','StLCG_position',StLCG_position);
% assignin('base','MBCG_position',MBCG_position);
% assignin('base','SwLeg_position',SwLeg_position);
% assignin('base','SwLCG_position',SwLCG_position);
%
% assignin('base','MBCG_velocity',MBCG_velocity);
% assignin('base','StLCG_velocity',StLCG_velocity);
% assignin('base','Base_angvel',Base_angvel);
% assignin('base','SwLCG_velocity',SwLCG_velocity);
% assignin('base','SwLeg_angvel',SwLeg_angvel);
% assignin('base','StLeg_angvel',StLeg_angvel);

StLCG_position = evalin('base','StLCG_position');
MBCG_position = evalin('base','MBCG_position');
SwLeg_position = evalin('base','SwLeg_position');
SwLCG_position = evalin('base','SwLCG_position');
MBCG_velocity = evalin('base','MBCG_velocity');
StLCG_velocity = evalin('base','StLCG_velocity');
Base_angvel = evalin('base','Base_angvel');
SwLCG_velocity = evalin('base','SwLCG_velocity');
SwLeg_angvel = evalin('base','SwLeg_angvel');
StLeg_angvel = evalin('base','StLeg_angvel');
```

```
MBCG_angvel = evalin('base','MBCG_angvel');
StLCG_angvel = evalin('base','StLCG_angvel');
SwLeg_angle = evalin('base','SwLeg_angle');
Base_angle = evalin('base','Base_angle');


% pause
final_index = 1;

EnergyComputationOneStep

SwitchStanceOneStepEOM

assignin('base','IC_StLeg_position',IC_StLeg_position);
assignin('base','IC_Base_angle',IC_Base_angle);
assignin('base','IC_StLeg_angle',IC_StLeg_angle);
assignin('base','IC_SwLeg_angle',IC_SwLeg_angle);
assignin('base','IC_StLeg_angvel',IC_StLeg_angvel);
assignin('base','IC_SwLeg_angvel',IC_SwLeg_angvel);
assignin('base','KE_vec',KE_vec);
assignin('base','PE_vec',PE_vec);

% KE_vec
% PE_vec

angle_des = action(1);
PGain = action(3);
DGain = action(4);

assignin('base','terrain_height_vector',delta_terrain_vec);

assignin('base','angle_des',angle_des);
assignin('base','PGain',PGain);
assignin('base','DGain',DGain);
assignin('base','ACTIVATE_AT_LEG_CROSS',ACTIVATE_AT_LEG_CROSS);


% myopts = simset('MinStep', evalin('base','min_step_size'));

% sim('BipedSimGainSchedule', 10)
assignin('base','t_max',1.5);
evalin('base','BipedOneStepEOM');
% sim('BipedSimOneStep', 10, myopts)

% assignin('base','StLCG_position',StLCG_position);
% assignin('base','MBCG_position',MBCG_position);
% assignin('base','SwLeg_position',SwLeg_position);
% assignin('base','SwLCG_position',SwLCG_position);
%
% assignin('base','MBCG_velocity',MBCG_velocity);
% assignin('base','StLCG_velocity',StLCG_velocity);
% assignin('base','Base_angvel',Base_angvel);
```

```
% assignin('base','SwLCG_velocity',SwLCG_velocity);
% assignin('base','SwLeg_angvel',SwLeg_angvel);
% assignin('base','StLeg_angvel',StLeg_angvel);

StLCG_position = evalin('base','StLCG_position');
MBCG_position = evalin('base','MBCG_position');
SwLeg_position = evalin('base','SwLeg_position');
SwLCG_position = evalin('base','SwLCG_position');
MBCG_velocity = evalin('base','MBCG_velocity');
StLCG_velocity = evalin('base','StLCG_velocity');
Base_angvel = evalin('base','Base_angvel');
SwLCG_velocity = evalin('base','SwLCG_velocity');
SwLeg_angvel = evalin('base','SwLeg_angvel');
StLeg_angvel = evalin('base','StLeg_angvel');
MBCG_angvel = evalin('base','MBCG_angvel');
StLCG_angvel = evalin('base','StLCG_angvel');
SwLeg_angle = evalin('base','SwLeg_angle');
Base_angle = evalin('base','Base_angle');

final_index = evalin('base','final_index');

EnergyComputationOneStep

impulse_work = KE_vec(2) - KE_vec(1);
collision_work = KE_vec(3) - KE_vec(2);
hip_actuator_work = energy_net;
hip_actuator_energy_added = energy_added;
gravity_work = -1*PE_delta;
hip_actuator_work = hip_actuator_work - gravity_work;

total_work = impulse_work + collision_work + hip_actuator_work +
gravity_work;
% added_kinetic_energy = KE_vec(1) - KE_vec_prev(1)

% energy_step_to_step = KE_vec(1)+PE_vec(1)-KE_vec_prev(1)-
PE_vec_prev(1);


%% DEBUG

% for(m = 1:length(HitCheck(1,:)))
%     index = find(HitCheck(:,m),1);
%     if(isempty(index))
%         states_out(:,m) = [0;0;0;0];
%         is_fallen(1,m) = 1;
%
%         distance_traversed(1,m) = 0;
%         time_elapsed(1,m) = 0;
%
%         %         energy_consumed(1,m) = impulse_work +
hip_actuator_energy_added;
%         energy_consumed(1,m) = hip_actuator_energy_added;
```

```
%
%         controller_error(1:2,m) = [angle_error(index);
angle_vel_error(index)];
%     else
%         X1out = IC_StLeg_position(2) - SwLeg_position(index,2);
%         X2out = SwLeg_position(index,1) - IC_StLeg_position(1);
%         X3out = Base_angvel(index);
%         X4out = SwLeg_angvel_joint(index);
%
%         distance_traversed(1,m) = MBCG_position(index,1) -
MBCG_position(1,1);
%         time_elapsed(1,m) = sim_time(index) - sim_time(1);
%
%         states_out(:,m) = [X1out; X2out; X3out; X4out];
%         is_fallen(1,m) = 0;
%         %         energy_consumed(1,m) = impulse_work +
hip_actuator_energy_added;
%         energy_consumed(1,m) = hip_actuator_energy_added;
%
%         controller_error(1:2,m) = [angle_error(index);
angle_vel_error(index)];
%     end
% end

% final_index = length(Base_angle)

final_index = evalin('base','final_index');
% size_SwLeg_position = evalin('base','size(SwLeg_position)');

states_out = 0;
is_fallen = (final_index==evalin('base','length(HitCheck_raw)-1'));
distance_traversed = evalin('base','SwLeg_position(final_index(1),1) -
SwLeg_position(1,1)');
time_elapsed = evalin('base','final_index(1)*dt');
energy_consumed = energy_added + abs(energy_dissipated)+impulse_work;
controller_p_error =
abs(angle_des+evalin('base','interleg_angle(final_index(1))'));
controller_d_error =
abs(evalin('base','interleg_velocity(final_index(1))'));
```

SwitchStanceOneStep.m

See page 168

TakeAction.m

```
function [new_state_num, possible_state_transitions] =
TakeAction(action_index, current_state_num, master_dynamics_table,
cum_probs)

rand_num = rand;

resulting_states =
master_dynamics_table{action_index}(current_state_num,:);

result_index = find(rand_num < cum_probs, 1);

new_state_num = resulting_states(result_index);

possible_state_transitions = resulting_states;
```

Genetic Optimization Algorithm Code

Step 1: Run "genopt3m1.m"

BipedOneStepEOM.m

```
%% Inputs:
% IC_StLeg_position
% IC_Base_angle
% IC_StLeg_angvel
% IC_SwLeg_angle
% IC_SwLeg_angvel
% terrain_height_vector
% ACTIVATE_AT_LEG_CROSS
%
% angle_des
% angle_ratio_vec
% gain_schedule
% ratio_schedule
%
% StLeg_mass
% StLeg_inertia
% StLeg_length
% StLCG_ratio
% SwLeg_mass
% SwLeg_inertia
% SwLeg_length
% SwLCG_ratio
% MBody_mass

%% Outputs:
% Base_angle
% Base_angvel
% StLCG_position
% StLCG_velocity
% StLCG_angvel
% StLeg_angle
% StLeg_angvel
% StLeg_angaccel
% MBCG_position
% MBCG_velocity
% MBCG_angvel
% MBCG_accel
% SwLeg_angle
% SwLeg_angvel_joint
% SwLeg_angaccel2
% SwLCG_position
```

```
% SwLCG_velocity
% SwLCG_angvel
% SwLeg_angle2
% SwLCG_accel
% SwLeg_angaccel
% interleg_angle
% interleg_velocity
% hip_torque
% SwLeg_position
% SwLeg_velocity
% SwLeg_accel

% HitCheck
% TotalHits
% FallCheck

%%

% close all

SLOMO = 1;
FRAMES_PER_SECOND = 30*SLOMO;
NUM_SAMPLES = 1;

ANIMATION_ON = 0;

theta1_init = 1*(IC_Base_angle*pi/180) + pi/2;
theta2_init = pi - IC_SwLeg_angle*pi/180 - IC_Base_angle*pi/180;
theta_dot1_init = IC_StLeg_angvel*pi/180;
theta_dot2_init = IC_SwLeg_angvel*pi/180;

%t_max = 2; % assigned
dt = 1e-3;
if(t_max == 0)
    num_max = 1;
else
    num_max = floor(t_max/dt);
end

theta1 = theta1_init;
theta2 = theta2_init;
theta_dot1 = theta_dot1_init;
theta_dot2 = theta_dot2_init;

tau = 0;
m = StLeg_mass;
mh = MBody_mass;
L = StLeg_length;
% g = 9.81; % Assigned elsewhere

a = StLCG_ratio*L;
b = SwLCG_ratio*L;
```

```
m1 = m + mh/2;
m2 = m1;
l1 = a + b;
l2 = l1;
lc1 = L - b*m/m1;
lc2 = L - lc1;
I1 = m*(b-lc2)^2 + 0.5*mh*lc2^2;
I2 = I1;

theta1_vec = zeros(num_max,1);
theta2_vec = zeros(num_max,1);
theta_dot1_vec = zeros(num_max,1);
theta_dot2_vec = zeros(num_max,1);
hip_torque = zeros(num_max,1);

interleg_angle = zeros(num_max,1);
interleg_velocity = zeros(num_max,1);

SwLeg_position = zeros(num_max,2);
SwLeg_velocity = zeros(num_max,2);

MBody_inertia(3,3) = 0.0001;
StLeg_inertia(3,3) = 0.0001;
SwLeg_inertia(3,3) = 0.0001;

for index = 2:num_max
    theta1_vec(index-1) = theta1;
    theta2_vec(index-1) = theta2;

    theta_dot1_vec(index-1) = theta_dot1;
    theta_dot2_vec(index-1) = theta_dot2;

    interleg_angle(index-1) = (pi - theta2)*180/pi;
    interleg_velocity(index-1) = theta_dot2*180/pi;

    SwLeg_position(index-1,1) = L*cos(theta1) + L*cos(theta1+theta2);
    SwLeg_position(index-1,2) = L*sin(theta1) + L*sin(theta1+theta2);

    SwLeg_velocity(index-1,1) = L*cos(theta_dot1) +
L*cos(theta_dot1+theta_dot2);
    SwLeg_velocity(index-1,2) = L*sin(theta_dot1) +
L*sin(theta_dot1+theta_dot2);

    hip_torque(index-1) = tau;

    d11 = m1*lc1^2 + m2*(l1^2+lc2^2+2*l1*lc2*cos(theta2)) + I1 + I2;
    d12 = m2*(lc2^2 + l1*lc2*cos(theta2)) + I2;
    d22 = m2*lc2^2 + I2;

    h1 = -m2*l1*lc2*sin(theta2)*theta_dot2^2 -
2*m2*l1*lc2*sin(theta2)*theta_dot2*theta_dot1;
```

```
    h2 = m2*l1*lc2*sin(theta2)*theta_dot1^2;

    p1 = (m1*lc1 + m2*l1)*g*cos(theta1) + m2*lc2*g*cos(theta1+theta2);
    p2 = m2*lc2*g*cos(theta1+theta2);

    tau = GetControlTorque(interleg_angle(index-1),
interleg_velocity(index-1), -angle_des, angle_ratio_vec, gain_schedule,
ratio_schedule);

    theta_dot_dot2 = (d11*(tau - h2 - p2) + d12*(h1 + p1))/(d11*d22 -
d12^2);
    theta_dot_dot1 = (d12*theta_dot_dot2 + h1 + p1)/(-d11);

    theta_dot1 = theta_dot_dot1*dt + theta_dot1;
    theta_dot2 = theta_dot_dot2*dt + theta_dot2;

    theta1 = theta_dot1*dt + theta1;
    theta2 = theta_dot2*dt + theta2;

    %%
%     interleg_angle(index-1)

end

theta1_vec(num_max) = theta1;
theta2_vec(num_max) = theta2;

if(num_max > 1)
    theta_dot1_vec(index-1) = theta_dot1;
    theta_dot2_vec(index-1) = theta_dot2;

    hip_torque(index-1) = tau;
else
    theta_dot1_vec(1) = theta_dot1;
    theta_dot2_vec(1) = theta_dot2;

    hip_torque(1) = tau;

    HitCheck = 1;
end

interleg_angle(num_max) = (pi + theta2)*180/pi;
interleg_velocity(num_max) = theta_dot2*180/pi;

clear MBCG_position
clear MBCG_velocity
clear StLCG_position
clear StLCG_velocity
clear SwLeg_position
clear SwLCG_position
clear SwLCG_velocity
clear Base_angvel
```

```
clear StLeg_angvel
clear StLCG_angvel
clear SwLeg_angvel

MBCG_position(:,1) = L*cos(theta1_vec)';
MBCG_position(:,2) = L*sin(theta1_vec)';

MBCG_velocity(:,1) = (-L*theta_dot1_vec.*sin(theta1_vec))';
MBCG_velocity(:,2) = (L*theta_dot1_vec.*cos(theta1_vec))';

StLCG_position(:,1) = StLCG_ratio*L*cos(theta1_vec)';
StLCG_position(:,2) = StLCG_ratio*L*sin(theta1_vec)';

StLCG_velocity(:,1) = (-
L*StLCG_ratio.*sin(theta1_vec).*theta_dot1_vec)';
StLCG_velocity(:,2) =
(L*StLCG_ratio.*cos(theta1_vec).*theta_dot1_vec)';

SwLeg_position(:,1) = L*cos(theta1_vec)' +
L*cos(theta1_vec+theta2_vec)';
SwLeg_position(:,2) = L*sin(theta1_vec)' +
L*sin(theta1_vec+theta2_vec)';

SwLCG_position(:,1) = L*cos(theta1_vec)' +
SwLCG_ratio*L*cos(theta1_vec+theta2_vec)';
SwLCG_position(:,2) = L*sin(theta1_vec)' +
SwLCG_ratio*L*sin(theta1_vec+theta2_vec)';

SwLCG_velocity(:,1) = (-L.*sin(theta1_vec).*theta_dot1_vec)' + (-
SwLCG_ratio*L*sin(theta1_vec+theta2_vec).*(theta_dot1_vec+theta_dot2_v
ec))';
SwLCG_velocity(:,2) = (L.*cos(theta1_vec).*theta_dot1_vec)' +
(SwLCG_ratio*L*cos(theta1_vec+theta2_vec).*(theta_dot1_vec+theta_dot2_
vec))';

Base_angle = (theta1_vec-pi/2)*180/pi;
SwLeg_angle = -theta2_vec*180/pi + 180 - Base_angle;

Base_angvel(:,1) = theta_dot1_vec*180/pi;
StLeg_angvel(:,1) = Base_angvel.*0;
StLCG_angvel(:,3) = theta_dot1_vec;
SwLeg_angvel(:,3) = (theta_dot1_vec+theta_dot2_vec);
MBCG_angvel = zeros(num_max,3);
SwLeg_angvel_joint = theta_dot2_vec*180/pi;

if(num_max > 1)
    left_height_vec = meshgrid([SwLeg_position(:,2);L],
terrain_height_vector);
    right_height_vec = meshgrid([-L;SwLeg_position(:,2)],
terrain_height_vector);

    terrain_mat = meshgrid(terrain_height_vector, ones(1,num_max+1))';
```

```
    HitCheck_raw = (left_height_vec <=
terrain_mat).*(right_height_vec >
terrain_mat).*([SwLeg_position(:,1)',-L] > 0.05);

    %Assumes only one terrain height
%     HitCheck_raw(length(HitCheck_raw)) = 1;

    final_index = find(HitCheck_raw);
    if(isempty(final_index))
        final_index = length(HitCheck_raw)-1;
    end

    HitCheck = zeros(1,final_index(1));
    HitCheck(final_index) = 1;
else
    HitCheck = 1;
    final_index = 1;
end

if(ANIMATION_ON)
%     hold on

    if(t_max == 0)
        time_interp = t_max;
        theta1_interp = theta1_vec;
        theta2_interp = theta2_vec;
    else
        time_interp = [0:1/FRAMES_PER_SECOND:t_max];
        theta1_interp = interp1(dt*[1:num_max], theta1_vec,
time_interp);
        theta2_interp = interp1(dt*[1:num_max], theta2_vec,
time_interp);
    end

    for index = [1:length(time_interp)]
        x1 = L*cos(theta1_interp(index));
        y1 = L*sin(theta1_interp(index));
        x2 = x1 + L*cos(theta1_interp(index)+theta2_interp(index));
        y2 = y1 + L*sin(theta1_interp(index)+theta2_interp(index));
        CMx1 = a*cos(theta1_interp(index));
        CMy1 = a*sin(theta1_interp(index));
        CMx2 = x1 + b*cos(theta1_interp(index)+theta2_interp(index));
        CMy2 = y1 + b*sin(theta1_interp(index)+theta2_interp(index));
        plot([0,x1], [0,y1], 'bo-', [x1,x2], [y1,y2], 'ro-', CMx1,
CMy1, 'bx', CMx2, CMy2, 'rx')
        axis equal
        axis([-2,2,-2,2])
        pause(1/FRAMES_PER_SECOND*SLOMO)
    end
end

% figure(4)
```

```
% plot(interleg_velocity)

% debug_BA = Base_angle(1)
% debug_SwA = SwLeg_angle(1)
%
% SwLeg_position
```

EnergyComputationOneStep.m

See page 123

GenerateTestSchedule.m

```
function test_schedule = GenerateTestSchedule(test_conditions,
num_tests)

% num_tests = 9;
test_schedule = zeros(5,num_tests);

X1_min = test_conditions(1,1);
X1_max = test_conditions(1,2);
X2_min = test_conditions(2,1);
X2_max = test_conditions(2,2);
X3_min = test_conditions(3,1);
X3_max = test_conditions(3,2);
X4_min = test_conditions(4,1);
X4_max = test_conditions(4,2);
terrain_mean = test_conditions(7,2);
terrain_sigma = test_conditions(7,2);

for m = 1:num_tests
    X1 = rand*(X1_max-X1_min)+X1_min;
    X2 = rand*(X2_max-X2_min)+X2_min;
    X3 = rand*(X3_max-X3_min)+X3_min;
    X4 = rand*(X4_max-X4_min)+X4_min;
    terrain_height = randn*terrain_sigma + terrain_mean;
    if(terrain_height > 3*terrain_sigma+terrain_mean)
        terrain_height = 0.1;
    elseif(terrain_height < -3*terrain_sigma+terrain_mean)
        terrain_height = -0.1;
    end

    test_schedule(:,m) = [X1; X2; X3; X4; terrain_height];
end
```

genopt3m1.m

```
% Gain Scheduling Format
% Ratio of desired interleg angle
% [-0.5, 0, 0.5]  length: n-1
% Initial Gain Selection (1st index indicates gain before crossing
angle 1)
% [10, 10, 10, 10]  length: n
% Kd/Kp ratio (1st index indicates ratio before crossing angle 1)
% [0.1, 0.1, 0.1, 0.1] length: n

try
    clc
    clear
    close all

    EMAIL_ALERT = 0;

    [last_update_time, last_update_text] = CheckUpdateRequests;

    addpath P:\UrbanRobots\private\Hubicki\Simulation\2009-12\Tools

    set_height = 0;

    MUTATION_SIGMA_GAIN = 0.125;
    MUTATION_SIGMA_RATIO = 0.01*5;
    MUTATION_SIGMA_IMPULSE = 0.125;

    MAX_GENERATIONS = 80;
    NUM_OFFSPRING = 50;
    NUM_DISCRETE_POINTS = 10;

    NUM_TESTS = 1;

    INITIAL_GAIN = 5;
    INITIAL_RATIO = 0.1*5;
    INITIAL_IMPULSE = 1;

    MIN_GAIN = 0;
    MIN_RATIO = 0;
    MIN_IMPULSE = 0;

    MAX_IMPULSE = 7;

    WEIGHTING = [150 0.375 10.625];

    SAVE_FILE_ON = 1;
    SAVE_EVERY = 10;

    current_time = clock;
```

```
    str_store = [num2str(current_time(1)) '_' num2str(current_time(2))
'_' num2str(current_time(3)) '_' num2str(current_time(4)) '_'
num2str(current_time(5)) '_' num2str(floor(current_time(6)))];
    savefile = ['GOAdata_' str_store '.txt'];

    halt_requested = 0;

    angle_ratio_vec = linspace(-1,1,NUM_DISCRETE_POINTS-1);
    gain_schedule = ones(1,NUM_DISCRETE_POINTS).*INITIAL_GAIN;
    ratio_schedule = ones(1,NUM_DISCRETE_POINTS).*INITIAL_RATIO;
    applied_impulse = INITIAL_IMPULSE;

    X1_min = 0;
    X1_max = 0;
    X2_min = 0.449;
    X2_max = 0.451;
    X3_min = -61; %-30;
    X3_max = -59; %-40;
    X4_min = -1; %25;
    X4_max = 1; %35;
    Impulse_min = 5;
    Impulse_max = 5;
    alpha_des_min = 25;
    alpha_des_max = 25;
    terrain_height_mean = 0.000;
    terrain_height_sigma = 0.00;

    test_conditions = [X1_min, X1_max;
        X2_min, X2_max;
        X3_min, X3_max;
        X4_min, X4_max;
        Impulse_min, Impulse_max;
        alpha_des_min, alpha_des_max;
        terrain_height_mean, terrain_height_sigma];

    parent_gain_schedule = gain_schedule;
    parent_ratio = ratio_schedule;
    parent_impulse = applied_impulse;

    gen = 0;
    done = 0;

    track_gen = zeros(MAX_GENERATIONS,3+2*NUM_DISCRETE_POINTS);

    while(~done && gen < MAX_GENERATIONS)

        gen = gen + 1;

        num_os = 1;

        [child_gain_matrix, dummy] = meshgrid(parent_gain_schedule,
ones(1,NUM_OFFSPRING));
```

```
        [child_ratio_matrix, dummy] = meshgrid(parent_ratio,
ones(1,NUM_OFFSPRING));
        child_impulse_matrix = ones(NUM_OFFSPRING,1).*parent_impulse;

        mutation_gain_matrix = randn(NUM_OFFSPRING,
NUM_DISCRETE_POINTS)*MUTATION_SIGMA_GAIN;
        mutation_ratio_matrix = randn(NUM_OFFSPRING,
NUM_DISCRETE_POINTS)*MUTATION_SIGMA_RATIO;
        mutation_impulse_matrix = randn(NUM_OFFSPRING,
1)*MUTATION_SIGMA_IMPULSE;

        child_gain_matrix = child_gain_matrix + mutation_gain_matrix;
        child_ratio_matrix = child_ratio_matrix +
mutation_ratio_matrix;
        child_impulse_matrix = child_impulse_matrix +
mutation_impulse_matrix;

        child_gain_matrix = (child_gain_matrix >=
MIN_GAIN).*child_gain_matrix + (child_gain_matrix <
MIN_GAIN).*MIN_GAIN;
        child_ratio_matrix = (child_ratio_matrix >=
MIN_RATIO).*child_ratio_matrix + (child_ratio_matrix <
MIN_RATIO).*MIN_RATIO;
        child_impulse_matrix = (child_impulse_matrix >=
MIN_IMPULSE).*child_impulse_matrix + (child_impulse_matrix <
MIN_IMPULSE).*MIN_IMPULSE;

        hold on

        test_schedule = GenerateTestSchedule(test_conditions,
NUM_TESTS);

        fitness = zeros(NUM_OFFSPRING,2);

        for m = 1:NUM_OFFSPRING
            %             m
            current_index = m;
            gain_schedule = child_gain_matrix(current_index,:);
            ratio_schedule = child_ratio_matrix(current_index,:);
            current_fit = GetFitnessTestSchedule(angle_ratio_vec,
child_gain_matrix(current_index,:),
child_ratio_matrix(current_index,:),
child_impulse_matrix(current_index), WEIGHTING, test_conditions,
test_schedule, set_height)
            fitness(current_index,:) = [current_fit, current_index];
            %             plot([angle_ratio_vec(1)-1,angle_ratio_vec],
child_gain_matrix(current_index,:), 'bx', [angle_ratio_vec(1)-
1,angle_ratio_vec],
child_ratio_matrix(current_index,:).*child_gain_matrix(current_index,:
), 'rx', -1.5, child_impulse_matrix(current_index), 'gx')
            plot([angle_ratio_vec(1)-1,angle_ratio_vec],
child_gain_matrix(current_index,:), 'bx', [angle_ratio_vec(1)-
```

```
1,angle_ratio_vec], child_ratio_matrix(current_index,:), 'rx', -1.5,
child_impulse_matrix(current_index), 'gx')
            pause(0.02)
            %         fitness(current_index,:)
            %         fitness
        end

        hold off
        %         close all
        clf

        adjust_mat = zeros(NUM_OFFSPRING,2);
        adjust_mat(:,1) = [1:NUM_OFFSPRING]'*0.00001;

        sorted_fitness = sortrows(fitness+adjust_mat);
        offsize = size(sorted_fitness);

        if(sorted_fitness(1,1) == 0)
            done = 1;
        end

        parent_gain_schedule =
child_gain_matrix(sorted_fitness(1,2),:);
        parent_ratio = child_ratio_matrix(sorted_fitness(1,2),:);
        parent_impulse = child_impulse_matrix(sorted_fitness(1,2));

        store_vec = [gen sorted_fitness(1,1) parent_gain_schedule
parent_ratio parent_impulse];

        track_gen(gen,:) = store_vec;

        RunUpdateRequestSystem

        if(mod(gen,SAVE_EVERY) == 0 && SAVE_FILE_ON)
            save_mat = track_gen(1:gen,:);

save(savefile,'WEIGHTING','NUM_DISCRETE_POINTS','NUM_OFFSPRING','NUM_T
ESTS','MUTATION_SIGMA_GAIN','MUTATION_SIGMA_RATIO','test_conditions','
save_mat','-ascii');
        end

        if(mod(gen,SAVE_EVERY) == 0 && halt_requested)
            done = 1;
        end

    end

    gain_schedule = parent_gain_schedule;
    ratio_schedule = parent_ratio;

    TestGS1
```

```
    text_body = ['Greetings,' 10 'Your simulation has executed without
error.' 10 'Regards,' 10 '- CodeBot'];
    if(EMAIL_ALERT)
        EmailSimulationUpdate('Simulation Complete!', text_body)
    end

catch ME
    rep = getReport(ME)
    rep_email = getReport(ME, 'extended', 'hyperlinks', 'off');
    text_body = ['The error report was recorded as follows:' 10 ' ' 10
rep_email 10 ' ' 10 'Regards,' 10 '- CodeBot'];
    if(EMAIL_ALERT)
        EmailSimulationUpdate('Simulation Update: Untimely
Termination', text_body)
    end
end
```

GetControlTorque.m

See page 132

GetFitnessTestSchedule.m

```
function fitness = GetFitnessTestSchedule(angle_ratio_vec,
gain_schedule, ratio_schedule, applied_impulse, weighting,
test_conditions, test_schedule, set_height)

% weighting

weight_time = weighting;
weight_energy = 1-weighting;

threshold_values(1) = 1;
threshold_values(2) = 5;

% test_conditions
%
% [X1_min, X1_max;
% [X2_min, X2_max;
% [X3_min, X3_max;
% [X4_min, X4_max;
% [Impulse_min, Impulse_max;
% [alpha_des_min, alpha_des_max]
% [terrain_height_min, terrain_height_max]

X1_min = test_conditions(1,1);
X1_max = test_conditions(1,2);
X2_min = test_conditions(2,1);
X2_max = test_conditions(2,2);
X3_min = test_conditions(3,1);
X3_max = test_conditions(3,2);
X4_min = test_conditions(4,1);
X4_max = test_conditions(4,2);
Impulse_min = test_conditions(5,1);
Impulse_max = test_conditions(5,2);
alpha_des_min = test_conditions(6,1);
alpha_des_max = test_conditions(6,2);
terrain_height_min = test_conditions(7,1);
terrain_height_max = test_conditions(7,2);

% assignin('base','angle_ratio_vec',angle_ratio_vec);
% assignin('base','gain_schedule',gain_schedule);
% assignin('base','ratio_schedule',ratio_schedule);

temp_size = size(test_schedule);
num_tests = temp_size(2);
```

```
fitness = 0;
for m = 1:num_tests
    X1 = test_schedule(1,m);
    X2 = test_schedule(2,m);
    X3 = test_schedule(3,m);
    X4 = test_schedule(4,m);

    angle_des = rand*(alpha_des_max-alpha_des_min)+alpha_des_min;
    impulse_magnitude = applied_impulse;
    PGain = 0;
    DGain = 0;
    ACTIVATE_AT_LEG_CROSS = 0;

    terrain_height_vec = test_schedule(5,m);

    states_in = [X1; X2; X3; X4];
    action = [angle_des, impulse_magnitude, PGain, DGain,
ACTIVATE_AT_LEG_CROSS];

    [states_out, is_fallen, distance_traversed, time_elapsed,
energy_consumed, y_converged, min_stance_angvel] =
StepToStepGOA(states_in, action, terrain_height_vec, threshold_values);

    scaling_factor = 1;

    yDiff = y_converged - set_height;
    %     is_fallen
    %     energy_consumed

    %     y_converged
    %     min_stance_angvel

    tripping_gradient_cost = 5*exp(-25*(y_converged-
terrain_height_vec));
    slipping_gradient_cost = 10*exp(-.25*min_stance_angvel);
%     min_stance_angvel

    if(is_fallen || y_converged < terrain_height_vec)
        fitness = fitness + weighting(1);
        disp('fell')
    else
        %         yCost = (exp(-6*(0-yDiff))).*(yDiff>0) + (-
100*yDiff+1).*(yDiff<=0) - 1;

        forward_step_distance = distance_traversed;
        biped_speed = forward_step_distance/time_elapsed;

        speed_cost = 1/biped_speed;
%         energy_consumed

        %         fitness = fitness + yCost + 2*energy_consumed +
speed_cost;
```

```
        fitness = fitness + weighting(2)*energy_consumed +
weighting(3)*speed_cost;
    end

    fitness = fitness + tripping_gradient_cost +
slipping_gradient_cost;
end

% pause
```

ImpulseComputationEOM.m

See page 136

InitialConditionTransformation.m

See page 142

InitStepToStepParams.m

See page 143

StepToStepGOA.m

```
function [states_out, is_fallen, distance_traversed, time_elapsed,
energy_consumed, y_converged, min_stance_angvel] =
StepToStepGOA(state_in, action, delta_terrain_vec, threshold_values)

% Initialize Parameters

angle_des = action(1);
PGain = action(3);
DGain = action(4);
ACTIVATE_AT_LEG_CROSS = action(5);

% threshold_values format
% threshold_values = [minimum acceptable angle error (deg), minimum
% acceptable angular velocity error (deg/sec)]

assignin('base','angle_des',angle_des);
assignin('base','PGain',PGain);
assignin('base','DGain',DGain);
assignin('base','ACTIVATE_AT_LEG_CROSS',ACTIVATE_AT_LEG_CROSS);
assignin('base','terrain_height_vector',0);

InitStepToStepParams(state_in)

applied_impulse = action(2);

assignin('base','t_max',0);
evalin('base','BipedOneStepEOM');

StLCG_position = evalin('base','StLCG_position');
MBCG_position = evalin('base','MBCG_position');
SwLeg_position = evalin('base','SwLeg_position');
SwLCG_position = evalin('base','SwLCG_position');
MBCG_velocity = evalin('base','MBCG_velocity');
StLCG_velocity = evalin('base','StLCG_velocity');
Base_angvel = evalin('base','Base_angvel');
```

```
SwLCG_velocity = evalin('base','SwLCG_velocity');
SwLeg_angvel = evalin('base','SwLeg_angvel');
StLeg_angvel = evalin('base','StLeg_angvel');
MBCG_angvel = evalin('base','MBCG_angvel');
StLCG_angvel = evalin('base','StLCG_angvel');
SwLeg_angle = evalin('base','SwLeg_angle');
Base_angle = evalin('base','Base_angle');

final_index = 1;

EnergyComputationOneStep

SwitchStanceOneStepEOM

assignin('base','IC_StLeg_position',IC_StLeg_position);
assignin('base','IC_Base_angle',IC_Base_angle);
assignin('base','IC_StLeg_angle',IC_StLeg_angle);
assignin('base','IC_SwLeg_angle',IC_SwLeg_angle);
assignin('base','IC_StLeg_angvel',IC_StLeg_angvel);
assignin('base','IC_SwLeg_angvel',IC_SwLeg_angvel);
assignin('base','KE_vec',KE_vec);
assignin('base','PE_vec',PE_vec);

angle_des = action(1);
PGain = action(3);
DGain = action(4);

assignin('base','terrain_height_vector',delta_terrain_vec);

assignin('base','angle_des',angle_des);
assignin('base','PGain',PGain);
assignin('base','DGain',DGain);
assignin('base','ACTIVATE_AT_LEG_CROSS',ACTIVATE_AT_LEG_CROSS);

assignin('base','t_max',2.0);
evalin('base','BipedOneStepEOM');

StLCG_position = evalin('base','StLCG_position');
MBCG_position = evalin('base','MBCG_position');
SwLeg_position = evalin('base','SwLeg_position');
SwLCG_position = evalin('base','SwLCG_position');
MBCG_velocity = evalin('base','MBCG_velocity');
StLCG_velocity = evalin('base','StLCG_velocity');
Base_angvel = evalin('base','Base_angvel');
SwLCG_velocity = evalin('base','SwLCG_velocity');
SwLeg_angvel = evalin('base','SwLeg_angvel');
StLeg_angvel = evalin('base','StLeg_angvel');
MBCG_angvel = evalin('base','MBCG_angvel');
StLCG_angvel = evalin('base','StLCG_angvel');
SwLeg_angle = evalin('base','SwLeg_angle');
Base_angle = evalin('base','Base_angle');
```

```
SwLeg_angvel_joint = evalin('base','SwLeg_angvel_joint');

final_index = evalin('base','final_index');

EnergyComputationOneStep

impulse_work = KE_vec(2) - KE_vec(1);
collision_work = KE_vec(3) - KE_vec(2);
hip_actuator_work = energy_net;
hip_actuator_energy_added = energy_added;
gravity_work = -1*PE_delta;
hip_actuator_work = hip_actuator_work - gravity_work;

total_work = impulse_work + collision_work + hip_actuator_work +
gravity_work;
% added_kinetic_energy = KE_vec(1) - KE_vec_prev(1)

% energy_step_to_step = KE_vec(1)+PE_vec(1)-KE_vec_prev(1)-
PE_vec_prev(1);


%% DEBUG

% for(m = 1:length(HitCheck(1,:)))
%     index = find(HitCheck(:,m),1);
%     if(isempty(index))
%         states_out(:,m) = [0;0;0;0];
%         is_fallen(1,m) = 1;
%
%         distance_traversed(1,m) = 0;
%         time_elapsed(1,m) = 0;
%
%         %         energy_consumed(1,m) = impulse_work +
hip_actuator_energy_added;
%         energy_consumed(1,m) = hip_actuator_energy_added;
%
%         controller_error(1:2,m) = [angle_error(index);
angle_vel_error(index)];
%     else
%         X1out = IC_StLeg_position(2) - SwLeg_position(index,2);
%         X2out = SwLeg_position(index,1) - IC_StLeg_position(1);
%         X3out = Base_angvel(index);
%         X4out = SwLeg_angvel_joint(index);
%
%         distance_traversed(1,m) = MBCG_position(index,1) -
MBCG_position(1,1);
%         time_elapsed(1,m) = sim_time(index) - sim_time(1);
%
%         states_out(:,m) = [X1out; X2out; X3out; X4out];
%         is_fallen(1,m) = 0;
%         %         energy_consumed(1,m) = impulse_work +
hip_actuator_energy_added;
```

```
%         energy_consumed(1,m) = hip_actuator_energy_added;
%
%         controller_error(1:2,m) = [angle_error(index);
angle_vel_error(index)];
%     end
% end
%
% final_index = length(Base_angle)

final_index = evalin('base','final_index');


X1out = IC_StLeg_position(2) - SwLeg_position(final_index,2);
X2out = SwLeg_position(final_index,1) - IC_StLeg_position(1);
X3out = Base_angvel(final_index);
X4out = SwLeg_angvel_joint(final_index);

states_out = [X1out; X2out; X3out; X4out];

% size_SwLeg_position = evalin('base','size(SwLeg_position)');

% states_out = 0;
is_fallen = (final_index==evalin('base','length(HitCheck_raw)-1'));
distance_traversed = evalin('base','SwLeg_position(final_index(1),1) -
IC_StLeg_position(1)');
time_elapsed = evalin('base','final_index(1)*dt');
energy_consumed = energy_added + abs(energy_dissipated) + impulse_work;

meet_threshold_vec = (abs(angle_des +
evalin('base','interleg_angle(1:final_index(1))')) <
threshold_values(1)).*(abs(evalin('base','interleg_velocity(1:final_in
dex(1))')) < threshold_values(2));
index_meet_threshold = find(meet_threshold_vec);

Swing_ypos = evalin('base','SwLeg_position(:,2)');

if(~isempty(index_meet_threshold))
    y_converged = Swing_ypos(index_meet_threshold(1));
else
    is_fallen = 1;
    y_converged = min(Swing_ypos(1:final_index));
end

min_stance_angvel = min(-1*Base_angvel(1:(numel(Base_angvel)-1)));
```

StepToStepTFEOM.m

```
function [states_out, is_fallen, distance_traversed, time_elapsed,
energy_consumed, controller_p_error, controller_d_error] =
StepToStepTFEOM(state_in, action, delta_terrain_vec)

% Initialize Parameters

angle_des = action(1);
PGain = action(3);
DGain = action(4);
ACTIVATE_AT_LEG_CROSS = action(5);

assignin('base','angle_des',angle_des);
assignin('base','PGain',PGain);
assignin('base','DGain',DGain);
assignin('base','ACTIVATE_AT_LEG_CROSS',ACTIVATE_AT_LEG_CROSS);
assignin('base','terrain_height_vector',0);

InitStepToStepParams(state_in)

applied_impulse = action(2);

% sim('BipedSimGainSchedule',0)
assignin('base','t_max',0);
evalin('base','BipedOneStepEOM');

% pause

% assignin('base','StLCG_position',StLCG_position);
% assignin('base','MBCG_position',MBCG_position);
% assignin('base','SwLeg_position',SwLeg_position);
% assignin('base','SwLCG_position',SwLCG_position);
%
% assignin('base','MBCG_velocity',MBCG_velocity);
% assignin('base','StLCG_velocity',StLCG_velocity);
% assignin('base','Base_angvel',Base_angvel);
% assignin('base','SwLCG_velocity',SwLCG_velocity);
% assignin('base','SwLeg_angvel',SwLeg_angvel);
% assignin('base','StLeg_angvel',StLeg_angvel);

StLCG_position = evalin('base','StLCG_position');
MBCG_position = evalin('base','MBCG_position');
SwLeg_position = evalin('base','SwLeg_position');
SwLCG_position = evalin('base','SwLCG_position');
MBCG_velocity = evalin('base','MBCG_velocity');
StLCG_velocity = evalin('base','StLCG_velocity');
Base_angvel = evalin('base','Base_angvel');
SwLCG_velocity = evalin('base','SwLCG_velocity');
SwLeg_angvel = evalin('base','SwLeg_angvel');
StLeg_angvel = evalin('base','StLeg_angvel');
```

```
MBCG_angvel = evalin('base','MBCG_angvel');
StLCG_angvel = evalin('base','StLCG_angvel');
SwLeg_angle = evalin('base','SwLeg_angle');
Base_angle = evalin('base','Base_angle');


% pause
final_index = 1;

EnergyComputationOneStep

SwitchStanceOneStepEOM

assignin('base','IC_StLeg_position',IC_StLeg_position);
assignin('base','IC_Base_angle',IC_Base_angle);
assignin('base','IC_StLeg_angle',IC_StLeg_angle);
assignin('base','IC_SwLeg_angle',IC_SwLeg_angle);
assignin('base','IC_StLeg_angvel',IC_StLeg_angvel);
assignin('base','IC_SwLeg_angvel',IC_SwLeg_angvel);
assignin('base','KE_vec',KE_vec);
assignin('base','PE_vec',PE_vec);

% KE_vec
% PE_vec

angle_des = action(1);
PGain = action(3);
DGain = action(4);

assignin('base','terrain_height_vector',delta_terrain_vec);

assignin('base','angle_des',angle_des);
assignin('base','PGain',PGain);
assignin('base','DGain',DGain);
assignin('base','ACTIVATE_AT_LEG_CROSS',ACTIVATE_AT_LEG_CROSS);


% myopts = simset('MinStep', evalin('base','min_step_size'));

% sim('BipedSimGainSchedule', 10)
assignin('base','t_max',1.5);
evalin('base','BipedOneStepEOM');
% sim('BipedSimOneStep', 10, myopts)

% assignin('base','StLCG_position',StLCG_position);
% assignin('base','MBCG_position',MBCG_position);
% assignin('base','SwLeg_position',SwLeg_position);
% assignin('base','SwLCG_position',SwLCG_position);
%
% assignin('base','MBCG_velocity',MBCG_velocity);
% assignin('base','StLCG_velocity',StLCG_velocity);
% assignin('base','Base_angvel',Base_angvel);
```

```
% assignin('base','SwLCG_velocity',SwLCG_velocity);
% assignin('base','SwLeg_angvel',SwLeg_angvel);
% assignin('base','StLeg_angvel',StLeg_angvel);

StLCG_position = evalin('base','StLCG_position');
MBCG_position = evalin('base','MBCG_position');
SwLeg_position = evalin('base','SwLeg_position');
SwLCG_position = evalin('base','SwLCG_position');
MBCG_velocity = evalin('base','MBCG_velocity');
StLCG_velocity = evalin('base','StLCG_velocity');
Base_angvel = evalin('base','Base_angvel');
SwLCG_velocity = evalin('base','SwLCG_velocity');
SwLeg_angvel = evalin('base','SwLeg_angvel');
StLeg_angvel = evalin('base','StLeg_angvel');
MBCG_angvel = evalin('base','MBCG_angvel');
StLCG_angvel = evalin('base','StLCG_angvel');
SwLeg_angle = evalin('base','SwLeg_angle');
Base_angle = evalin('base','Base_angle');

final_index = evalin('base','final_index');

EnergyComputationOneStep

impulse_work = KE_vec(2) - KE_vec(1);
collision_work = KE_vec(3) - KE_vec(2);
hip_actuator_work = energy_net;
hip_actuator_energy_added = energy_added;
gravity_work = -1*PE_delta;
hip_actuator_work = hip_actuator_work - gravity_work;

total_work = impulse_work + collision_work + hip_actuator_work +
gravity_work;
% added_kinetic_energy = KE_vec(1) - KE_vec_prev(1)

% energy_step_to_step = KE_vec(1)+PE_vec(1)-KE_vec_prev(1)-
PE_vec_prev(1);


%% DEBUG

% for(m = 1:length(HitCheck(1,:)))
%     index = find(HitCheck(:,m),1);
%     if(isempty(index))
%         states_out(:,m) = [0;0;0;0];
%         is_fallen(1,m) = 1;
%
%         distance_traversed(1,m) = 0;
%         time_elapsed(1,m) = 0;
%
%         %         energy_consumed(1,m) = impulse_work +
hip_actuator_energy_added;
%         energy_consumed(1,m) = hip_actuator_energy_added;
```

```
%
%         controller_error(1:2,m) = [angle_error(index);
angle_vel_error(index)];
%     else
%         X1out = IC_StLeg_position(2) - SwLeg_position(index,2);
%         X2out = SwLeg_position(index,1) - IC_StLeg_position(1);
%         X3out = Base_angvel(index);
%         X4out = SwLeg_angvel_joint(index);
%
%         distance_traversed(1,m) = MBCG_position(index,1) -
MBCG_position(1,1);
%         time_elapsed(1,m) = sim_time(index) - sim_time(1);
%
%         states_out(:,m) = [X1out; X2out; X3out; X4out];
%         is_fallen(1,m) = 0;
%         %         energy_consumed(1,m) = impulse_work +
hip_actuator_energy_added;
%         energy_consumed(1,m) = hip_actuator_energy_added;
%
%         controller_error(1:2,m) = [angle_error(index);
angle_vel_error(index)];
%     end
% end

% final_index = length(Base_angle)

final_index = evalin('base','final_index');
% size_SwLeg_position = evalin('base','size(SwLeg_position)');

states_out = 0;
is_fallen = (final_index==evalin('base','length(HitCheck_raw)-1'));
distance_traversed = evalin('base','SwLeg_position(final_index(1),1) -
SwLeg_position(1,1)');
time_elapsed = evalin('base','final_index(1)*dt');
energy_consumed = energy_added + abs(energy_dissipated)+impulse_work;
controller_p_error =
abs(angle_des+evalin('base','interleg_angle(final_index(1))'));
controller_d_error =
abs(evalin('base','interleg_velocity(final_index(1))'));
```

SwitchStanceOneStepEOM.m

See page 166

TestGS1.m

```
min_step_size = 0;

NUM_SAMPLES = 500;

current_time = clock;
str_store = [num2str(current_time(1)) '_' num2str(current_time(2)) '_'
num2str(current_time(3)) '_' num2str(current_time(4)) '_'
num2str(current_time(5)) '_' num2str(floor(current_time(6)))];
savefile = ['TestGS_' str_store '.txt'];

result_vector = zeros(NUM_SAMPLES, 6);

test_schedule = GenerateTestSchedule(test_conditions, NUM_SAMPLES);

Impulse_min = test_conditions(5,1);
Impulse_max = test_conditions(5,2);
alpha_des_min = test_conditions(6,1);
alpha_des_max = test_conditions(6,2);
terrain_height_min = test_conditions(7,1);
terrain_height_max = test_conditions(7,2);

for r = 1:NUM_SAMPLES

    X1 = test_schedule(1,r);
    X2 = test_schedule(2,r);
    X3 = test_schedule(3,r);
    X4 = test_schedule(4,r);

%     gain_schedule = parent_gain_schedule;
%     ratio_schedule = parent_ratio_schedule;

    angle_des = rand*(alpha_des_max-alpha_des_min)+alpha_des_min;
    impulse_magnitude = parent_impulse; %rand*(Impulse_max-
Impulse_min)+Impulse_min;
    PGain = 0;
    DGain = 0;
    ACTIVATE_AT_LEG_CROSS = 0;

    terrain_height_vec = test_schedule(5,r);

    states_in = [X1; X2; X3; X4];
    action = [angle_des, impulse_magnitude, PGain, DGain,
ACTIVATE_AT_LEG_CROSS];
```

```
    [states_out, is_fallen, distance_traversed, time_elapsed,
energy_consumed, y_converged] = StepToStepGOA(states_in, action,
terrain_height_vec, [1, 5]);

    sct = energy_consumed/(distance_traversed*3*9.81);

    result_vector(r,:) = [is_fallen energy_consumed time_elapsed
distance_traversed y_converged sct];

end

save(savefile,'WEIGHTING','NUM_DISCRETE_POINTS','NUM_OFFSPRING','NUM_T
ESTS','MUTATION_SIGMA_GAIN','MUTATION_SIGMA_RATIO','test_conditions','
angle_ratio_vec','gain_schedule','ratio_schedule','test_conditions','r
esult_vector','save_mat','-ascii');
```

Gradient-Descent Heuristic Parameter Tuning Code

Step 1: Run "GradientDescentHeuristic.m"

BipedOneStepEOM.m

See page 195

EnergyComputationOneStep.m

See page 123

GetHeuristicFitness.m

```
function [cost, new_heuristic_parameters, step_speed_vec, sct_vec,
fallen_vec] = GetHeuristicFitness(heuristic_parameters, num_points,
initial_states, angle_des, weights, angle_ratio_vec, gain_schedule,
ratio_schedule)

NUM_POINTS = num_points;

current_time = clock;
str_store = [num2str(current_time(1)) '_' num2str(current_time(2)) '_'
num2str(current_time(3)) '_' num2str(current_time(4)) '_'
num2str(current_time(5)) '_' num2str(floor(current_time(6)))];
savefile = ['HeuristicTest_' str_store '.txt'];

threshold_values(1) = 1;
threshold_values(2) = 5;

X1 = initial_states(1);
X2 = initial_states(2);
X3 = initial_states(3);
X4 = initial_states(4);

% angle_des = 20;
impulse_magnitude = 1.9;
PGain = -100*pi/180;
DGain = PGain/10;
ACTIVATE_AT_LEG_CROSS = 0;

% angle_ratio_vec = [-1    -0.75 -0.5  -0.25 0    0.25  0.5   0.75  1];
% gain_schedule = 1.0.*[3.217524076   2.854678338     3.476241818
     3.823100867     3.871491193     4.046413024     4.290005978
     4.634429007     4.844104976     5.049151904];
```

```
% ratio_schedule = [1.486095457  1.045793855     1.085561499
     0.993975082     0.903970409     0.804858655     0.695194153
     0.460624279     0.228208339     0.752267483];

root_gain_schedule = gain_schedule;
root_ratio_schedule = ratio_schedule;

terrain_height_vec = [0];

states_in = [X1; X2; X3; X4];
action = [angle_des, impulse_magnitude, PGain, DGain,
ACTIVATE_AT_LEG_CROSS];

% clock
q = 0;

impulse_magnitude_vec =
linspace(heuristic_parameters(1),heuristic_parameters(2),NUM_POINTS);
gain_scale =
linspace(heuristic_parameters(3),heuristic_parameters(4),NUM_POINTS);
ratio_scale =
linspace(heuristic_parameters(5),heuristic_parameters(6),NUM_POINTS);

% heuristic_parameters
% pause

for impulse_magnitude = impulse_magnitude_vec
    action = [angle_des, impulse_magnitude, PGain, DGain,
ACTIVATE_AT_LEG_CROSS];


    q = q + 1;

    gain_schedule = root_gain_schedule * gain_scale(q);
    ratio_schedule = root_ratio_schedule * ratio_scale(q);


    assignin('base','angle_ratio_vec',angle_ratio_vec);
    assignin('base','gain_schedule',gain_schedule);
    assignin('base','ratio_schedule',ratio_schedule);

    [states_out, is_fallen, distance_traversed, time_elapsed,
energy_consumed, y_converged, min_stance_angvel] =
StepToStepGOA(states_in, action, terrain_height_vec, threshold_values);

    sct = energy_consumed/((distance_traversed)*3*9.81);
    step_speed = (distance_traversed)/time_elapsed;

    fallen_vec(q) = is_fallen(1);
    y_converged_vec(q) = y_converged;
    min_stance_angvel_vec(q) = min_stance_angvel;
    sct_vec(q) = sct;
```

```
    step_speed_vec(q) = step_speed;


    %     is_fallen
    %     sct
    %     step_speed


end

fallen_vec = fallen_vec.*fallen_vec;

x_fitted = linspace(0.3, 1.3, 1000);
y_fitted = 0.9247*x_fitted.^2 - 0.1634*x_fitted + 0.2335;


%
save(savefile,'angle_ratio_vec','gain_schedule','ratio_schedule','impu
lse_magnitude_vec','gain_scale','ratio_scale','fallen_vec','sct_vec','
step_speed_vec','y_converged_vec','min_stance_angvel_vec','X1','X2','X
3','X4','angle_des','-ascii');
hold on
plot(step_speed_vec, sct_vec.*(fallen_vec), 'rx', step_speed_vec,
sct_vec.*(1-fallen_vec), 'b.', x_fitted, y_fitted, 'k--')
axis([0 1.4 0 1.8])
grid on


VEC = [0, 1-fallen_vec, 0];
heur_breadth = max(diff(find(1-VEC)))-1;
heur_index = find(find(max(diff(find(1-VEC))) == diff(find(1-VEC)),1)
== cumsum(1-VEC),1);

min_crop_index = heur_index
max_crop_index = heur_index+heur_breadth-1
cropped_indices = min_crop_index:max_crop_index;

if(min_crop_index == 0 || max_crop_index == 0 || max_crop_index >
length(step_speed_vec))
    speed_range = 0;
    sct_mean = 10;
    min_crop_index = 1;
    max_crop_index = length(step_speed_vec);
else
    min_speed = step_speed_vec(min_crop_index);
    max_speed = step_speed_vec(max_crop_index);


    speed_range = abs(max_speed-min_speed);
    sct_mean = mean(sct_vec(cropped_indices));
end
```

```
cost = sct_mean - 2*speed_range;
% find(1-fallen_vec)

new_heuristic_parameters = [impulse_magnitude_vec(min_crop_index),
impulse_magnitude_vec(max_crop_index), gain_scale(min_crop_index),
gain_scale(max_crop_index), ratio_scale(min_crop_index),
ratio_scale(max_crop_index)];
```

GradientDescentHeuristic.m

```
clear
clc

NUM_POINTS = 10;
NUM_GEN = 50;

% X1 = 0.05;
% X2 = 0.55;
% X3 = -30;
% X4 = 0;

X1 = 0.00;
X2 = 0.45;
X3 = -60;
X4 = 0;

drift_magnitude_vec = [0.1, 0.1, 0.1, 0.1, 0.05 0.05];

angle_des = 25;

current_time = clock;
str_store = [num2str(current_time(1)) '_' num2str(current_time(2)) '_'
num2str(current_time(3)) '_' num2str(current_time(4)) '_'
num2str(current_time(5)) '_' num2str(floor(current_time(6)))];
savefile = ['AutoTunerData_' str_store '.txt'];

initial_states = [X1; X2; X3; X4];

% heuristic_parameters = [2.6, 4.8, 0.4, 1.5, 0.6667, 1.0];
% heuristic_parameters = [3.25, 4.8, 0.4, 1.5, 0.6667, 1.0];

heuristic_parameters = [2.9284    5.6000    0.6506    2.3000    0.6560
1.4000];

root_angle_ratio_vec = [-1  -0.75 -0.5  -0.25 0     0.25  0.5   0.75  1];
root_gain_schedule = 1.0.*[3.2175240762.854678338     3.476241818
     3.823100867     3.871491193     4.046413024     4.290005978
     4.634429007     4.844104976     5.049151904];
root_ratio_schedule = [1.486095457    1.045793855     1.085561499
     0.993975082     0.903970409     0.804858655     0.695194153
     0.460624279     0.228208339     0.752267483];

current_heuristic_parameters = heuristic_parameters;

weighting = 0;

figure(2)
[cost, new_heuristic_parameters, step_speed_vec, sct_vec, fallen_vec]
= GetHeuristicFitness(heuristic_parameters, NUM_POINTS, initial_states,
```

```
angle_des, weighting, root_angle_ratio_vec, root_gain_schedule,
root_ratio_schedule)

cost
% pause
figure(1)

cost_gen(1) = cost;
nhp_gen(1,:) = new_heuristic_parameters;
ssm_gen(1,:) = step_speed_vec;
sct_gen(1,:) = sct_vec;
fallen_gen(1,:) = fallen_vec;

for gen = 2:NUM_GEN
    cost_vec = zeros(2^6,1);
    nhp_matrix = zeros(2^6,6);
    step_speed_matrix = zeros(2^6,NUM_POINTS);
    sct_matrix = zeros(2^6,NUM_POINTS);
    fallen_matrix = zeros(2^6,NUM_POINTS);

    for s = 1:(2^6)
        adjust_vec = (dec2binvec(s-1,6)*2-1).*drift_magnitude_vec;
        heuristic_parameters = current_heuristic_parameters +
adjust_vec;
        %
        figure(1)
        [cost, new_heuristic_parameters, step_speed_vec, sct_vec,
fallen_vec] = GetHeuristicFitness(heuristic_parameters, NUM_POINTS,
initial_states, angle_des, weighting, root_angle_ratio_vec,
root_gain_schedule, root_ratio_schedule);
        cost_vec(s) = cost;
        nhp_matrix(s,:) = new_heuristic_parameters;
        step_speed_matrix(s,:) = step_speed_vec;
        sct_matrix(s,:) = sct_vec;
        fallen_matrix(s,:) = fallen_vec;
        pause(0.05)
    end
    clf

    figure(2)
    if(gen > 2)
        plot(best_ssm, best_sct, 'b.')
    end

    minimum_cost = min(cost_vec);
    min_index = find(minimum_cost == cost_vec,1);
    best_hp = nhp_matrix(min_index,:);
    best_ssm = step_speed_matrix(min_index,:);
    best_sct = sct_matrix(min_index,:);
    best_fallen = fallen_matrix(min_index,:);

    cost_gen(gen) = minimum_cost;
```

```
    nhp_gen(gen,:) = best_hp;
    ssm_gen(gen,:) = best_ssm;
    sct_gen(gen,:) = best_sct;
    fallen_gen(gen,:) = best_fallen;


    plot(best_ssm, best_sct, 'g.')
    axis([0 1.4 0 1.8])
    title('Tuned Heuristic Progression')
    figure(1)

    current_heuristic_parameters = best_hp;


save(savefile,'cost_gen','nhp_gen','ssm_gen','sct_gen','fallen_gen','-
ascii');

end
```

ImpulseComputationEOM.m

See page 136

End of appendix.