View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by KU ScholarWorks

DEVELOPMENT OF A CONTROL SYSTEM FOR A 2D BIPED WALKING ROBOTIC TESTBED

by

Michael D. Knopp

Submitted to the Department of Mechanical Engineering

and the Graduate Faculty of the University of Kansas

in partial fulfillment of the requirements for the degree of

Master's of Science.

____

Terry N. Faddis, Chair

____

Carl W. Luchies, Committee Member

____

Robert C. Umholtz, Committee Member

Date Defended: ______________________________

The Thesis Committee for Michael D. Knopp

certifies that this is the approved version of the following thesis:

DEVELOPMENT OF A CONTROL SYSTEM FOR A 2D BIPED WALKING ROBOTIC TESTBED

____

Terry N. Faddis, Chair

Date approved: ________________________

ABSTRACT

Due to the great potential promise of bipedal walking robots, it was determined that the Jaywalker, a first generation bipedal walking testbed, should be designed and built as a platform for the study of bipedal walking over rough terrain and as a testbed for the research and development of the components necessary to develop a bipedal walking robot and for use in research beneficial to humanity that the testbed could facilitate. The Jaywalker adopts an approach of attempting to integrate the best of the ZMP bipedal walking robots characteristics with the best characteristics of the dynamic walkers and the basis for a control system capable of handling the distributed multiple inputs and multiple outputs necessary for an advanced bipedal walking robotic in a modular and easily updatable way.

ACKNOWLEDGEMENTS

I want to acknowledge several people without whose help and support I would not have been able to complete this work.

I want to thank my advisor for this thesis project and the head of the KU Intelligent Systems and Automation Lab, Dr. Terry Faddis. It was through his efforts and support that I came to the University of Kansas and found my way to the conclusion of this project.

I want to thank my fellow lab mates: Bryce Baker, Josh Williams, and Marc Ruiz, whose efforts were instrumental in the development and construction of the Jaywalker bipedal walking robot.

I want to thank the rest of my defense committee, Dr. Carl W. Luchies and Professor Robert C. Umholtz for the contribution of their time and efforts towards the completion of this work.

Lastly, I wish to convey my gratitude to my wife and children; Melodee, Ilyanna, Madelyn, and Gannon. Without their nearly limitless patience and support I would never have been able to complete my graduate degree.

TABLE OF CONTENTS

Abstract iii

Acknowledgements iv

Table of Contents v

List of Tables vi

List of Figures vii

1. Introduction 1
   - 1.1 Motivation 1
   - 1.2 Author's Contributions 2
   - 1.3 Overview of Jaywalker Design 2
   - 1.4 Hip 5
   - 1.5 Thigh 7
   - 1.6 Knee 7
   - 1.7 Ankle 8
   - 1.8 Foot 9
   - 1.9 Tether 10
   - 1.10 Human Gait Cycle 10
   - 1.11 Review of Bipedal Walking Robots 11

2. Design and Development 15
   - 2.1 Design Considerations 15
   - 2.2 Controls Overview 16
   - 2.3 Control Program Iterations 19
   - 2.4 Control Classes 21
   - 2.5 Interfacing Software 25
   - 2.6 User Interface 26

3. TUNING AND TESTING OF THE CONTROL SYSTEM 33
   - 3.1 Tuning Variables 33
   - 3.2 Testing Overview 35
   - 3.3 Single Step Testing 36
   - 3.4 Double Step Testing 39

4. Conclusions 40
   - 4.1 Analysis of Results 40
   - 4.2 Recommendations for Future Work 41

APPENDIX A: DESIGN SUPPLEMENT 44
   - A.1 C# Control System Code 44
   - A.2 C++ Interfacing Code 107

LIST OF TABLES

Table 1: The anthropomorphic leg section lengths of a 10-year-old male from Winter's biomechanics book and the actual Jaywalker leg section lengths. 4
Table 2: The anthropomorphic foot dimensions of a 10-year-old male, 140 cm (55 in) in height, and the actual Jaywalker's foot dimensions. 9
Table 3: Single step test results for the outside legs transitioning through the swing phase......... 37
Table 4: Single step test results for the inside leg transitioning through the swing phase. 38

LIST OF FIGURES

Figure 1: Hip Ratchet System (The gear and housing of the closest leg is hidden.) 5
Figure 2: Independent Hip Drive 6
Figure 3: Leg Extension Guidance System.. 7
Figure 4: Pneumatic Knee.. 8
Figure 5: Hybrid Parallel Ankle Actuator.. 8
Figure 6: Jaywalker's Curved Foot... 9
Figure 7: A single stride of a walking human's gait cycle and common nomenclature as from Inman [7]... 11
Figure 9: Jaywalker Accelerometer GUI Interface... 28
Figure 10: Jaywalker Hip GUI Interface 29
Figure 11: Jaywalker Thigh GUI Interface.. 30
Figure 12: Jaywalker Knee GUI Interface 31
Figure 13: Jaywalker Ankle GUI Interface.. 32
Figure 14: Jaywalker Foot Placeholder for future development..................................... 33

1. INTRODUCTION

1.1 MOTIVATION

The term "robot" was introduced into the human language in 1921 when Karel Capek released his play *Rossum's Universal Robots*. In this play, Capek envisioned robots that looked like humans and that navigated through the human world and interacted naturally with humans as they performed tasks. Thus, from the very first usage of the term robot, humans have strived to achieve a bipedal form that shares our environment and our benefits of locomotion.

Over the nearly ninety years since the term "robot" was introduced they have evolved in imaginative concept and in reality. Humans have imagined and created walking robots with numerous numbers of legs, wheeled robots, tracked robots, swimming robots, rolling robots, and even hovering robots. We have sent mobile robots to the farthest and most extreme parts of the Earth and beyond. Yet, despite all of this advancement we are still striving for the holy grail of mobile robotics, the practical bipedal walking robot that was first envisioned by Karel Capek.

Walking robots, in general, have an advantage over their wheeled and tracked brethren because the former requires only discrete and discontinuous stable footholds, while the later requires a continuous path from one point of travel to another [1]. Bipedal walking robots have a further advantage over multi-legged robots due to their lower number of legs requiring a lower number of stable footholds to travel the same path.

In addition to the increased terrain advantage of bipedal robots over multi-legged robots, bipedal robots carry similarities between themselves and humans. These similarities can be advantageous into the research of human locomotion, the design of human prosthetics, and the study of human leg and foot diseases. The bipedal robot can also be beneficial for the service

industry where the human similarities would benefit it in the working and living environments designed for humans. Additionally, a bipedal robot moves in a way more familiar to us and that serves to increase empathy from its human associates [2].

It is for these reasons that the Intelligent Systems and Automation Lab at the University of Kansas decided to build an energy efficient bipedal robot capable of walking over rough terrain. To aid in the design of this robot it was determined that a logical first step towards this goal was to build a bipedal test bed, named Jaywalker. This testbed would serve as the foundation for the testing and proofing out of various technologies to go into the final energy efficient bipedal walking robot.

1.2 AUTHOR'S CONTRIBUTIONS

For the Jaywalker to function properly a robust and easily extensible control program was necessary to allow for the creation of a stable gait. This control program needed to foremost allow for a stable gait cycle on the initial prototype of the Jaywalker, and then easily allow for the addition and customization of the control system to support the various technologies that the Jaywalker would need to incorporate for the necessary future research needed to complete the final energy efficient bipedal walking robot. The class based control program and graphical user interface deemed essential to meet the goals of the project were created and tested by the author for the biped test bed.

1.3 OVERVIEW OF JAYWALKER DESIGN

Prior research into dynamic walking robots has shown that adopting a naturally dynamic walking control scheme when a periodic gate is sufficient for stability is advantageous in terms of energy expenditure. With that idea in mind a design principle of the Jaywalker was to

maintain as close to a naturally dynamic design as possible to allow for a dynamic control scheme when practical, while still allowing for the full actuation that allows the zero moment point (ZMP) robots to traverse the rough terrain that a dynamic walker cannot [3]. These two goals are contradictory and required a series of trade-offs with economic reality often determining the lengths to which we could push each.

A 3D bipedal walking robot requires a shifting torso system to vary the center of gravity of the robot throughout the gait cycle to maintain stability and balance out lateral forces. However, it was determined that the cost, in both time and resources, were too great within the time available to develop the Jaywalker as a fully 3D bipedal walking robot. Thus it was determined that the first generation of the Jaywalker would be a 2D bipedal walking robotic design, similar to McGeer's 2D bipedal passive walking robot [4].

The 2D bipedal walking robot is a design that has three legs, and walks by synching the outside legs to move as if a single leg. Thus when the robot is viewed and analyzed in two dimensions from the sagittal plane it appears the same as a fully 3D bipedal walking robot. For the initial design of the Jaywalker bipedal testbed all of the three legs were built to identical criteria, with a hip ratcheting system, a leg extension guidance system in the thigh, a binary knee actuation system, a hybrid parallel ankle actuation system, and a passively curved foot [3]. For the initial testing of the Jaywalker the hip was set to a state where it was not actuated, and thus allowed to act in its passively dynamic state. This was done to ascertain the Jaywalker's ability to stably execute a dynamic gait cycle.

The choice of a 2D bipedal walking robotic system as the basis for the Jaywalker limited the rough terrain study and any future research to 2D, but had the added benefit of greatly reducing

the complexity of the control system required. This reduction in complexity led to a reduced time to testing readiness for the Jaywalker bipedal testbed.

Beyond the previously mentioned design criteria the design goal of building a bipedal walking robotic system which could use natural dynamics when practical along with the stated desire to use the Jaywalker for future research into human gait and research for prosthetics meant that there were further design constraints applied to the Jaywalker bipedal test bed.

The Jaywalker needed to keep as much of its mass as possible at or above the hip, as this helps maintain the inverted pendulum model of a natural dynamic walker. Further, the Jaywalker needed to conform as closely as possible to anthropomorphic human proportions, which were referenced from Winter's biomechanics book [5]. To keep the size, weight, and cost of the Jaywalker manageable it was determined to use an average 10-year-old male as the anthropomorphic human model for the Jaywalker.

LEG SECTION LENGTHS			
LEG SECTION	10-YEAR-OLD MALE	JAYWALKER	ERROR
THIGH	342.2 mm (13.47 in)	352.2 mm (13.86 in)	2.9%
SHANK	343.6 mm (13.53 in)	369.6 mm (14.55 in)	7.5%

Table 1: The anthropomorphic leg section lengths of a 10-year-old male from Winter's biomechanics book and the actual Jaywalker leg section lengths.

Table 1 lists both the anthropomorphic lengths for an average 10-year-old male given by Winter's biomechanics book and the actual lengths of the Jaywalker, along with their percent errors [3]. The deviations from the desired proportions were design trade-offs forced by the need to incorporate the actuators necessary to gain the non-periodic control characteristics from the ZMP bipedal walking robots into the Jaywalker and the economic realities of actuators with both the power requirements and package sizes to actually fit into the desired anthropomorphic proportions. Coincidental with the deviations from the anthropomorphic lengths, the inclusion of the actuators necessary made anthropomorphic masses an impossibility. Despite the

impossibility of matching the desired masses of each limb, the masses were kept as light as possible, with the Jaywalker's estimated mass, based on design models, being 20kg. The final weight of the completed Jaywalker was 28kg, or 140% of the estimated design weight. Despite this substantial increase in mass over the estimated weight, analysis of the actuation systems showed that no replacements were necessary.

The Jaywalker's physical design consisted of several prototype systems including, a powered hip system, an extending thigh system, an actuated knee, a hybrid pneumatic-electric ankle actuation system, and a curved foot.

1.4 HIP

Throughout the development and testing of the Jaywalker two different hip systems were incorporated. Both of these hip system's primary purpose was to control the swing leg's position and velocity throughout the swing phase of the gait cycle, while at a minimum matching an anthropomorphic range of motion at a speed of about 17 rpm.

The first hip system was the hip ratcheting system. The hip ratcheting system was designed with the goal of allowing the Jaywalker to be passively dynamic when actuation was not necessary for stability. However, to achieve the goal of allowing for the actuation when necessary the hip ratcheting system incorporated a stepper motor with an approximate 60:1 gear ratio, which drove a driveshaft through a timing

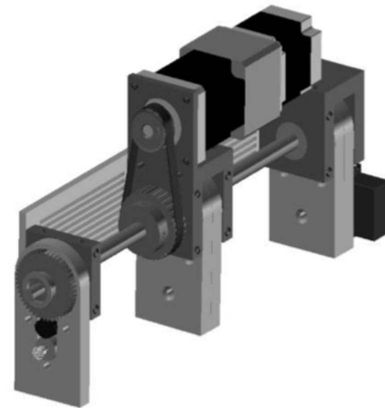

Figure 1: Hip Ratchet System (The gear and housing of the closest leg is hidden.)

belt. To engage or disengage the legs to this driveshaft the hip ratcheting system used an RC servomotor to engage and disengage a ratchet and pawl interface between the thigh and the drive shaft.

When it is determined that the Jaywalker needs hip actuation the pawl is engaged with the ratchet by actuating the servo. When it is determined that the Jaywalker does not require hip actuation the servo motor moves the pawl to a neutral position, thus removing the interface between the drive shaft and the thigh. Thus allowing the leg to swing freely through it natural dynamic cycle. Figure 1 shows a view of the design model for the hip ratcheting system.

Unfortunately, testing revealed that the natural dynamics of the Jaywalker, due to the legs being a substantial percentage (88.5%) of the hip mass and thus affecting the hip motion of the swing leg, did not give a long enough step length to successfully walk with the hip ratcheting system.

The second hip system was the independent hip drive system. The independent hip drive system was incorporated to address the shortcomings of the hip ratcheting system revealed in testing, specifically, the need to continuously actuate the hip of the Jaywalker during the gait cycle until a time that the proportional masses of the legs to the hip can be corrected. The independent hip drive system, design model shown in Figure 2, incorporates a pair of stepper motors and gear sets as the hip ratcheting system interfaced through a gear train to the outside legs. One advantage of the independent hip drive system over the hip ratcheting system is the ability for the independent hip drive system to control each of the outside legs independently, thus allowing for slightly different step lengths, which results in a gradual turn.

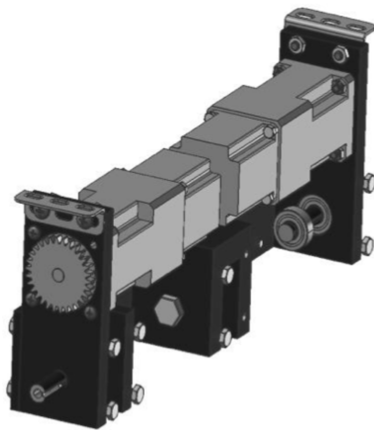

Figure 2: Independent Hip Drive

In both systems, the stepper motors and the middle leg are rigidly attached to the hip frame. This allows for the stepper motor to drive either the outside legs or the middle leg, depending upon which is in stance phase, thus in contact with the ground, and which is in swing phase.

1.5 THIGH

The leg extension guidance system was incorporated into the thigh, and allows the Jaywalker to vary its leg length. The leg extension guidance system has a maximum adjustability of 5 cm, in 0.63 cm increments. The leg extension guidance system's primary purpose was to add an additional degree of freedom to the Jaywalker. This additional degree of freedom could be used to adjust the potential energy of the leg during the gate cycle and thus making the Jaywalker more stable by reducing variance in the potential energy.

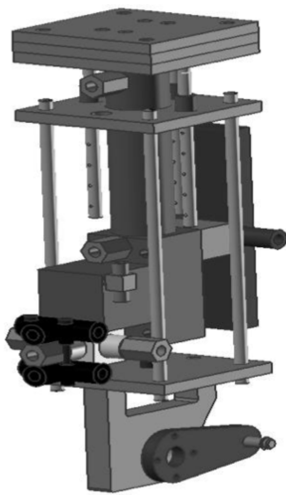

Figure 3: Leg Extension Guidance System

The leg extension guidance system incorporates an air cylinder with a rod lock and series of perforated linear guides. The perforations in the linear guides allows for the manual placement of stops which act as adjustments to the extension length of the leg extension guidance system when the air cylinder is extended. The rod lock was added when it was calculated that the cylinder force from air alone was not sufficient to adequately prevent the leg extension guidance system from acting as an air spring during the heel strike. Figure 3 shows a view of the design model for the leg extension guidance system.

1.6 KNEE

A binary knee actuation system was incorporated into the knee, and allows the Jaywalker to lock the knee in either a flex or extended state.

This design was chosen to reduce control and design complexity; it was decided to follow the example of McGeer and set mechanical limits on the knee. While McGeer used the natural dynamics of the swing leg and a suction cup with a bleed hole to acts as the mechanical limit, it was decided to use an air cylinder as an actuator and locking mechanism for the knee of the Jaywalker, due to the increased weight of the Jaywalker [4]. Due to space requirements, the knee of the Jaywalker does not have the typical 90° range of motion of a human, but is more than the minimum 70° flex necessary for clearance, at 75° of flexure. Figure 4 shows a view of the design model for the knee locking system.

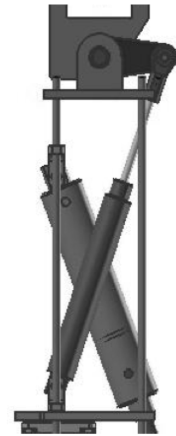

Figure 4: Pneumatic Knee

1.7 ANKLE

The hybrid parallel ankle actuator was incorporated into the ankle of the Jaywalker to allow for precise control of the ankle flexure while providing about 37.1 N-m of torque, and an anthropomorphic range of motion from about 14° dorsiflexion to about 47° plantarflexion.

Similar to the knee, the ankle is a simple hinge joint with an attached air cylinder for actuation. However, in addition to the air cylinder, the ankle has a stepper motor attached in parallel to the ankle through a 15:1 worm gear for additional torque and the positioning requirements necessary for the precise positioning necessary during the gate cycle.

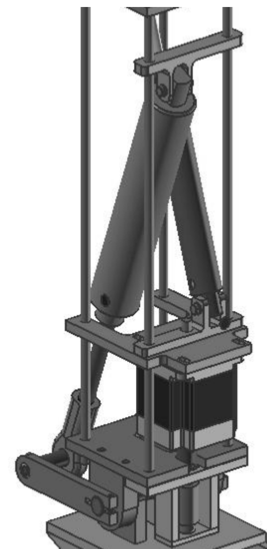

Figure 5: Hybrid Parallel Ankle Actuator

The hybrid parallel ankle actuator works by tuning the air cylinder to provide just enough output torque to allow for the stepper motor to control the precise position and velocity of the ankle. Figure 5 shows a view of the design model for the hybrid parallel ankle actuator.

1.8 FOOT

The foot of the Jaywalker was designed as a passive element to be used for the initial testing of the Jaywalker, in place of any future advanced robotic foot research.

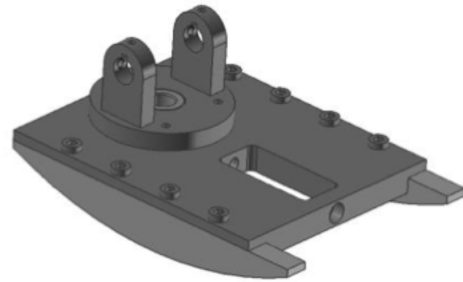

Figure 6: Jaywalker's Curved Foot

Through discussion and research it was determined to make the foot a curved foot to better mimic the natural curve of the human foot as it moves through a step in the gait cycle. A normalized radius of curvature of 0.25 was chosen for the radius of curvature of the Jaywalker's foot, this was based on the anthropomorphic normalized radius of curvature of 0.2 to 0.3 [6]. It has been shown that a curved foot reduces the amount of energy necessary to complete a step, in fact the larger the radius of curvature the less work required to complete a step [6].

FOOT DIMENSIONS		
DIMENSION	10-YEAR-OLD MALE	JAYWALKER
LENGTH	212 mm (8.36 in)	178 mm (7.0 in)
WIDTH	77 mm (3.03 in)	127 mm (5.0 in)
HEIGHT	54 mm (2.15 in)	59 mm (2.34 in)
RAD. OF CURVATURE	0.2 – 0.3	0.25

Table 2: The anthropomorphic foot dimensions of a 10-year-old male, 140 cm (55 in) in height, and the actual Jaywalker's foot dimensions.

The initial design of the foot was to have a passively compliant rotation about the longitudinal axis to aid in turning and any misalignment of the foot during the gait cycle. However, it was soon determined through testing that the passive compliance was detrimental to

the stability and repeatability. Therefore, steps were taken to remove the compliance from the foot.

It was determined during testing that there were slight differences in the leg lengths and that the robot was experiencing strong impulses from the heel strike. To help alleviate the problems from these two issues rubber strips were applied to the bottoms of the feet to help compensate for the differences in leg lengths and to absorb some of the impact of heel strike.

1.9 TETHER

To protect the Jaywalker a wheeled tether was designed to prevent damage to the Jaywalker during instability events and to hold the external control hardware necessary for the interfacing of the Jaywalker to electrical power, pneumatics, and the computer.

The tether was designed in such a way as to minimize its effect on the Jaywalker during its testing, but to still allow adequate protection in the instance of a fall. Towards these goals it was decided that a trapezoidal frame mounted on wheels and attached to the Jaywalker by a series of flexible tethers would be the best way to achieve this.

1.10 HUMAN GAIT CYCLE

The human walking gait cycle is comprised of several events that occur sequentially and periodically. Starting from the beginning of the double stance phase, with leg A forward and its heel in contact with the ground, the walker will plantarflex leg B's ankle. This will cause an extension of the overall leg length of leg B, which will cause the walker's center of mass to rise and shift forward. Eventually, leg B will move off of the ground with a toe-off event, this is the point at which the walker transitions from the double stance phase to the single stance phase. As the walker's center of gravity moves forward it will pass outside of the supporting foot's contact

envelope and the walker will begin a forward falling motion as leg B swings forward, dorsiflexing its foot, and extending the knee to bring the leg ahead of the walker. Eventually, leg B's heel will make contact with the ground and the walker has re-entered a double stance phase with leg B now forward and leg A as the back leg. From here leg A will go through a toe-off event and enter the swing phase while flexing the knee to allow for ground clearance as it swings forward. Ultimately, leg A will be extended in front of the walker and the heel will strike the ground ahead of the walker, bringing the gait cycle back to the original double stance phase, with leg A forward and its heel in contact with the ground.

As is evident from the above periodic events the human walking gait cycle will take the walker through two steps for each cycle, also colloquially known as a stride.

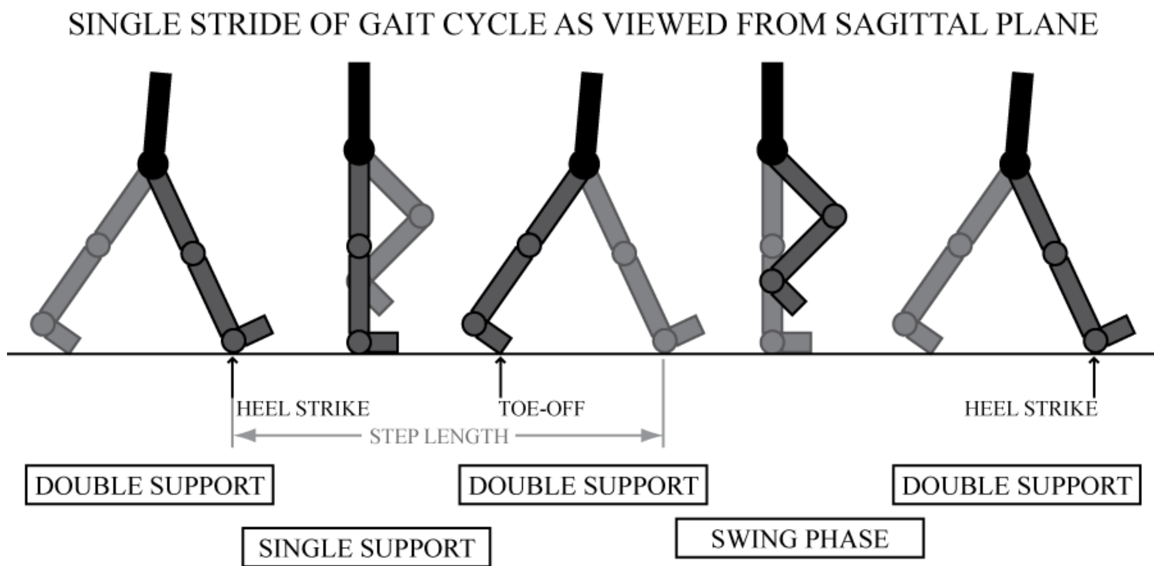

Figure 7: A single stride of a walking human's gait cycle and common nomenclature as from Inman [7].

1.11 REVIEW OF BIPEDAL WALKING ROBOTS

Despite all of our advances in robotics since Karel Capek introduced the world to his humanoid service robots, we had still not produced a robust and efficient bipedal robot which is capable of traverse rough terrain as well as humans.

We have made great strides in the last few decades in advancing bipedal robots. The late 1980s and early 1990s saw an explosion in bipedal walking robots, with Honda's ASIMO series [9] and McGeer's dynamic walker defining the two primary groups of walking robots: active walking robots and dynamic walking robots respectively. However, both of these paths have serious drawbacks for practical bipedal walking robots capable of navigating the environment as well as humans.

Active walking robots require constant actuation of the joints to maintain stability and to provide for locomotion. Of the active walkers the ZMP model is the most common. ZMP control systems use computer modeling and computational algorithms to determine the position of the desired zero moment point, the point on the ground where the sum of the forces acting on the robot equals zero, and controls the robot to adjust the actual ZMP's position to a desired point. A simple ZMP controller would attempt to ensure that the ZMP lies within the contact foot's profile while in single stance, or between the two feet in double stance to maintain a stable stance throughout the entire gait cycle.

Of this well known group, the most notable of these are the ASIMO line of biped robots designed and built by Honda. Through the generations of its controllers Honda has implemented the use of a second controller, a Ground Reaction Force controller, with the previously mentioned ZMP controller, which adjusts the positioning of the feet to control the center of actual total ground reaction force, C-ATGRF [8]. In addition to this they have added a third controller, which models the ZMP and adjusts the ideal trajectory to shift the desired ZMP to a stable position [8]. The difference between the desired ZMP and C-ATGRF is the tipping moment arm. Honda's new generation walker, ASIMO, uses a predictive movement controller added to these earlier controllers to anticipate movement and smooth walking [9].

The inherent stability of the ZMP control system has allowed for amazing strides in allowing robots to navigate their environment, from walking up and down stairs, inclines, moving laterally, playing soccer, and even righting themselves after a fall. Depending upon the computational power available the modeling and analysis can be done prior to a robot's traversal of an environment, or on-the-fly during the robot's travels.

The computational complexity and modeling required of the ZMP model mandates that ZMP robots have powerful computers available and have absolute control of all axis of freedom at all times. This makes ZMP robots highly inefficient and slow. Advancements in technology have allowed for the offset of these drawbacks as battery technology and computational power has become more efficient, but ZMP will always be less energy efficient than human walking due to the constant energy draw of the motors on each joint necessary to ensure that the robot remains stable. Further, even the most advanced on-the-fly ZMP controllers experience problems when the robot experiences an unexpected event or large perturbation as the complex algorithms required for the ZMP process extend the reaction time of the robot to unacceptable levels [3].

Dynamic walking robots do not require to constant actuation of the ZMP controlled walking robots to maintain stability nor for locomotion. Instead they rely on their dynamics and short inputs of energy to maintain stability and locomotion. Tad McGeer pioneered this segment of walking robotics with his work on a 2D passive walking robot, which relied on gravity for power and dynamics for control. This walker served as the predecessor for the majority of the dynamic walker research done to date [3, 10].

Two of these predecessors are the researcher groups at Cornell and Delft Universities. Both of these two research groups have developed minimally actuated dynamic bipedal walkers based largely upon McGeer's earlier work. Both of these bipedal walkers introduce a minimal actuation

to their walkers to allow for short inputs of energy during the walking cycle. This change to McGeer's earlier design allows for these two lab's bipedal dynamic walkers to traverse level ground, whereas McGeer's purely dynamic walker could only walk down an inclined plain. Cornell's minimally actuated dynamic walker gains energy through the use of actuated ankles that input energy during the toe-off phase of the gait cycle. While Delft University's minimally actuated dynamic bipedal walker uses actuated hips to input energy into the gait cycle.

Beyond the placement of the minimal actuation both of these labs have opted for a very simplistic control system, which uses ground sensors as inputs and actuation of the actuators as outputs [11]. This keeps the energy draw and complexity of the system very low. Which in turn keeps the energy required for the control system very low, which is in keeping with the goal of most dynamic walking robot designers; energy efficiency.

In keeping with the above stated goal, research into minimally actuated biped robots has yielded energy efficient bipedal robots that do not require the energy intensive control technology or algorithms of the ZMP bipedal walking robots to produce a stable walking gait. In fact, the Cornell research group has achieved parity with a human in regards to the transport specific energy cost and is only 10% less efficient than a human in the mechanical specific energy cost [11]. This is significantly more efficient than a typical ZMP bipedal walking robot like ASIMO, which is sixteen times less efficient in transport specific energy cost and 220% less efficient than a human in the mechanical specific energy cost [11].

Despite the advantages of the dynamic bipedal walking robots, there is one major area where they are deficient, rough terrain traversal. Due to the reliance of the dynamic walking robots on their natural dynamics they will converge on a particular periodic gate. This convergence does not allow dynamic walking bipedal robots to traverse many types of rough terrains as well as the

more actively controlled ZMP bipedal walking robots, whose periodic gate can be altered by their active control system.

2. DESIGN AND DEVELOPMENT

2.1 DESIGN CONSIDERATIONS

As discussed earlier, ZMP walking robots use a computationally intensive control algorithm that calculates and controls the zero moment point of the walking robot to allow for a stable gait cycle. Dynamic walkers, meanwhile, have either no controls, other than their natural dynamics, and must move down an inclined plane or a very simple state based logic control system that can trigger an energy input, which allows for traversal of flat terrain. Since the Jaywalker was designed to be a combination of these two types of bipedal walking robots, both control schemes were considered. It was determined that a more complex form of state based logic, or conditional logic, would suffice for the Jaywalker.

To the ends of this goal a few things were decided upon. The conditional logic of the Jaywalker's control system would be based upon sensor inputs placed on the feet, shanks, thighs, and hip so as to provide good situational awareness of the Jaywalker's position and orientation. The Jaywalker's control system would be written in a C based object oriented language to modularize the code as much as possible, which would not only help in the coding of the original Jaywalker control system, but should help future programmers port the control scheme to another system and expand as needed for future control scheme expansions. The primary control hardware was determined to be a multiprocessor Microsoft Windows based PC, which would handle the control system execution, data recording, and user interface for the Jaywalker.

The hardware components of the Jaywalker's control system consist of four general hardware subsystems: the Windows PC, the input sensors, the actuators, and the interfacing circuitry.

2.2 CONTROLS OVERVIEW

The first hardware selected for the Jaywalker were the actuators as the requirements for the actuators was dictated by the mechanics and dynamics of the system.

As a consequence of the design decision to make the Jaywalker a hybrid of the ZMP and dynamic bipedal walking robots two primary considerations were given to the selection of actuators. The first requirement was the selection of actuators with the proper motion type; some of the motions required a linear motion, while some required a rotary motion. Within each type of motion range the designer determined the power requirements of each actuator to provide the necessary input of energy into the system to allow for walking along level ground and for future implementation of rough terrain walking. Once these requirements were determined the actuators were chosen to minimize the mass of the actuator while maintaining a particular price range.

The combination of the high power output per unit weight and price, lead to all of the linear actuators being chosen as pneumatic air cylinders. Electric linear actuators were explored for these applications, but in every case either the necessary response speed was too low, the cost per force output was too low, or the cost per unit force output was too high. There was also research into the emerging field of artificial muscle, but these products are still in their infancy in terms of manufacturing and thus, they have a very high price per unit force output, and current generations typically have a stroke which was shorter than required for proper function of the actuator.

For the rotary actuators, high output stepper motors were chosen for their high power output per unit price and weight. In addition to these stepper motors, when lower RPMs and higher torques were required, as with the hip actuators, planetary gearboxes were added to the stepper motors to meet the output requirements.

Two types of sensors were determined necessary for the Jaywalker's state input, ground contact sensors for foot contact position reporting and orientation sensors for limb orientation data.

The first sensors chosen were lever style contact switches for placement on the feet as toe-off and heel-strike binary sensors. Through experimentation the exact placement of both the heel-strike and toe-off sensors was a critical variable input in the calibration of the Jaywalker for a stable gait cycle. To allow for the wide range of placement of these contact sensors and to minimize catastrophic damage to these sensors in the event of stable gait cycle failure these sensors were attached to the side of the foot with a pliable adhesive, which allowed for easy movement of the sensor to widely different locations along the foot and also provided an amount of give for the sensor in the event of the application of a force sufficient to destroy a rigidly mounted contact sensor.

The second sensors for the Jaywalker were three-axis accelerometers that were placed on each foot, shank, and thigh, with a single one attached to the hip. The mounting of the accelerometers was achieved through the same pliable adhesive as was used for the mounting of the foot contact switches. However, this time the mounting material was chosen for its damping qualities, as the pliable material reduced the ambient vibrations produced by the walking gait which tended to swamp the orientation data produced by the accelerometers. The signals from

these accelerometers were then analyzed and used as an orientation sensor to determine the relative position of each part of the leg in relation to each other.

To control the Jaywalker's actuators and receive feedback from the input sensors, interface circuitry was either purchased or designed. To move all of the input and output from the Jaywalker to the computer used for the control program a National Instruments USB interface was implemented. The NI-USB-6509 was chosen due to its inclusion of the necessary number of I/O ports available and its modularity, which made it capable of passing all data back and forth from the Jaywalker to the computer over a single USB cable.

The control interface for the pneumatic actuators linked an electronic solenoid to a control circuit using an array of Darlington transistors as isolation switches to separate the control voltage and amperage supplied from the NI-USB interface from the voltage and current necessary to switch the solenoids.

The stepper motors implemented on the Jaywalker were driven by 10 microstep drives interfaced with an Oregon Micro Systems MaxP motion controller card, which was interfaced to an internal expansion port of the computer. The selection of the stepper motor drives and OMS controller card were dictated by the recommendation of the stepper motors manufacturer, US Digital, due to their own performance curve testing showing the stepper motors being capable of outputting 6.85 in-lbs of torque at 500 RPM.

The accelerometers chosen for the Jaywalker required the most elaborate interfacing due to their serial communication output. The NI-USB interface lacked the ability to switch a line I/O fast enough to interface with the serial signal from the accelerometers. Therefore, the signal from the accelerometers was fed through an SX microprocessor for the translation of the ten accelerometers from a serial signal to a parallel signal, which the NI-USB interface could pass to

the controller. To minimize the number of SX microprocessors necessary an array of logical or integrated circuits were used to allow for the selection of a single accelerometer for its data to pass over to the controller, see Baker for a scaled down example of the accelerometer interfacing circuit [3].

To control the Jaywalker it was determined that several parallel processes had to happen simultaneously. Therefore, a custom computer was built from off the shelf parts. The critical part of this computer was the quad-core hyperthreaded Intel processor. This processor made the control structure capable of true multiprocess functions. Additionally, the computer required a proper PCI internal interface slot for the physical interfacing with the OMS MaxP motion controller card and a free USB port for the physical interfacing with the NI-USB interface. To ease in software development it was determined that Microsoft Windows should be installed as the operating system of this computer. This was due to the availability of code libraries for the Windows operating system from both National Instruments and Oregon Micro Systems.

2.3 CONTROL PROGRAM ITERATIONS

The Jaywalker's control program went through two major iterations.

The first iteration of the Jaywalker's control program was developed and written in ANSI C++ and ran on a single core Microsoft Windows computer. This choice was primarily made for C++'s object oriented nature and the fact that the OMS motion controller card included libraries compatible with C++.

Upon development and testing the control scheme was proven out piece by piece as each class and class method was tested and refined. The problem with the single core hardware did not become apparent until our initial attempt for the Jaywalker to take a step. When the first

iteration control program was ran through the single step testing the Jaywalker would not take a stable step. While the swing phase leg was in motion the stance phase leg would not actuate the ankle to add energy to the stride and position the Jaywalker for the next step in the cycle.

The issue was finally tracked down to the fact that each leg has to run through its gait process simultaneously and the looping algorithm necessary to check orientation of each leg throughout its cycle was not conducive to a single process / single threaded application such as the first iteration of the control program written in C++ and running on a single core computer.

Therefore, it was concluded that to move the project forward a multi-core computer capable of true multiprocess / multithreaded applications was necessary. The shift in programming languages to C# utilizing the .Net framework from the ANSI C++ programming language was decided after several tests were made using various C-based languages and frameworks with an eye towards ease of object-oriented programming, multiprocess / multithreaded programming, and interfacing hardware libraries.

The only drawback to this decision was the outdated state that Oregon Microsystems left their libraries in with regard to modern programming languages or frameworks. However, the lack of a function library for the .Net programming environment from Oregon Microsystems was solved by stripping the relevant parts for controlling the stepper motors from the first iteration of the C++ control program, integrating it with a command line interface, and running it as a background process which was called from the new C# based main control program.

In general, the current iteration of the Jaywalker control program conforms very closely to the schema outlined in section 2.2. The primary differences in terms of added methods occur in the hip class, where two additional methods, Hip.Neutral and Hip.Lock, were added specifically for the hip ratcheting system to lock and unlock the paw in the ratchet mechanism. The other

class with extra methods is the addition of methods to the thigh class that allow for the control of the L.E.G.S. System. In this particular instance the thigh class includes an additional accessor method in the Thigh.IsExtended accessor, which returns only the state of the L.E.G.S. Systems thigh. Also in regard to the particular needs of the L.E.G.S. System the class includes two methods, which extend the thigh, Thigh.Extend, and retract the thigh, Thigh.Retract. At the time of this writing the L.E.G.S. System code has not been implemented into the control programming, but the schema is there and will require very little effort by the designer to implement, as was the goal of the modular, object-oriented approach of the Jaywalker's control system.

2.4 CONTROL CLASSES

It was determined very early into the design process that an object oriented programming approach and control software development structure would be the best way to design the control software for future progression and development of the Jaywalker. This would allow for a class to be developed for each limb and component separately, and then easily allow for the rapid compiling of a control structure capable of swapping out new and varied limbs for multiple students and testing.

To achieve this the control program was broken down into their real world object counterparts.

The robot class was developed as the root class for the Jaywalker. This is the class through which the user interface couples with the control program for each individual part of the Jaywalker. The primary control methods for the robot are simply the Robot.Walk command and the Robot.Stop command. These commands take care of the underlying steps necessary to make

the Jaywalker start its gait cycle and to stop its gait cycle. Additionally, the Robot class has methods for passing parameters from the GUI to the Jaywalker through the Robot.DistributeData method, and for recording the input data from the Jaywalker to the computer's hard drive through the Robot.StartRecording and Robot.StopRecording methods.

The hip class is the only limb class that is not a child class of the Leg class. The hip class is responsible for the control and interfacing of the Jaywalker's hip. There is one sensor input method, Hip.GetAngle, for the hip class that returns orientation of the hip. In addition to this single input class there are two movement methods for the hip class in general which control the actuation of the hip, Hip.Actuate and Hip.Stop, which begin hip actuation and stop hip actuation respectively.

The leg class is the parent class of all remaining limb and joint objects comprising the Jaywalker control schema. It primarily serves as a handler for the coordination of all components of each individual leg as the Jaywalker moves through its gait cycle. The leg class contains three accessor methods: the Leg.leg accessor, the Leg.StancePhase accessor, and finally the Leg.SetRobot Accessor. The Leg.leg accessor allows for each leg to have its leg value set and returned. The Leg.StancePhase accessor allows for each leg to have its stance phase set and returned. The Leg.SetRobot accessor allows for the setting only of the robot to which the leg belongs, which serves as a callback for the legs to pass data to each other.

Since the leg itself has no sensor inputs all of the sensing methods are simply pass-through methods for child objects. The three actuation methods available for public interfacing to this class are the Leg.Actuate method which loops the leg through the gait cycle and the Leg.Stop method which stops the gait cycle for the leg.

The thigh class controls the sensor feedback from the thigh of the Jaywalker. To achieve this the thigh class contains two accessor methods: the Thigh.Leg accessor which allows for each thigh to have its leg value set and returned and the Thigh.SetAngle accessor which allows for the setting only of the orientation angle of the thigh. The thigh class also contains a single sensing method in Thigh.GetAngle, which returns the angle of the thigh from its orientation sensor.

The knee class controls the actuation of the knee. The knee class contains two accessor methods: the Knee.Leg accessor which allows for each knee to have its leg value set and returned and the Knee.IsExtended accessor which allows for reading only of the state of the knee. In addition to these accessor methods, the knee class contains two actuation methods: the Knee.FlexKnee method, which causes knee flexion and Knee.ExtendKnee that causes knee extension.

Similar to the thigh, the shank class controls the sensor feedback from the shank of the Jaywalker. To achieve this the shank class contains the same two accessor methods that the thigh does: the Shank.Leg accessor which allows for each shank to have its leg value set and returned and the Shank.SetAngle accessor which allows for the setting only of the orientation angle of the shank. The shank class also contains a single sensing method in Shank.GetAngle, which returns the angle of the shank from its orientation sensor.

The ankle class controls the actuation of the ankle of the Jaywalker. The ankle contains two accessor methods: the Ankle.Leg accessor which allows for each ankle to have its leg value set and returned and the Ankle.Velocity accessor which allows for the ankle to have the velocity, in stepper motor steps per second, with which an actuation method will execute, set or read. Due to the more complex nature of the actuation of the ankle there are four primary actuation methods and one convenience method in the ankle class. The Ankle.MoveTo method allows for the ankle

to be moved to a provided position, in stepper motor steps. The Ankle.Dorsiflex and Ankle.Plantarflex methods move the ankle to its respective stops in their respective directions. The Ankle.Stop method stops all motion of the ankle, regardless of which other actuation method has been invoked. Then, finally, the Ankle.StepReset method is provided as a convenient way for the operator to return the ankle to a hard coded position for the start of a gait cycle, as opposed to resetting the position manually after each test.

As with the other limbs the foot class is primarily responsible for the state reporting of its real world counterpart. The foot class contains the same two accessor methods that the other limb classes contain: the Foot.Leg accessor which allows for each foot to have its leg value set and returned and the Foot.SetAngle accessor which allows for the setting only of the orientation angle of the foot. Unlike the previous limbs, the foot class has three sensing methods: the Foot.GetAngle method returns the orientation of the foot, the Foot.GetHeelStrike method returns the state of the heel-strike sensor, and the Foot.GetToeOff method which returns the state of the toe-off sensor.

The toe class controls the actuation of the toe. The toe class contains two accessor methods: the Toe.Leg accessor which allows for each toe to have its leg value set and returned and the Toe.IsExtended accessor which allows for reading only of the state of the toe. In addition to these accessor methods, the toe class contains two actuation methods: the Toe.FlexToe and Toe.ExtendToe methods, which actuate their relative joints in the indicated direction.

Two interface classes were developed and implemented for the Jaywalker testbed, the mpu class and its child the accelerometer class.

The mpu class is used to control the SX microprocessor in passing the data from the accelerometers on the Jaywalker to the computer running the control program. The mpu class

contains only two read methods, the MPU.ReadData method and the MPU.Stop method. The MPU.ReadData method loops through the accelerometers, reads the data from them, and propagates that data to each element of the robot. The MPU.Stop method stops the reading and updating loop.

The accelerometer class is responsible for the actual signal output to the microprocessor,setting which accelerometer is being polled and the reading and formatting of the returned data from the accelerometer. To this end the accelerometer has only a single method Accelerometer.Read, which manages the task assigned to it.

2.5 INTERFACING SOFTWARE

In addition to the two interface classes, and additional three helper classes were developed to aid in the programming process.

The constant class is a singleton class used as a central repository for named constants used throughout the program. This singleton class was developed to make updates and modifications easier, as any changes to the pin selection of the NI-USB or OMS MaxP card can be updated in one place and propagate throughout the entire program from there.

The remaining helper class is the position class. This class is designed to allow for the easy storage and retrieval of Cartesian and orientation values into a single variable.

In addition to the software written as the primary controller for the Jaywalker, software was necessary to be written and maintained separately from the main control program. This was necessary for the SX microprocessor that interfaced the accelerometers and the OMS MaxP motion controller card.

The SX microprocessor required a separate program code due to the fact that it ran on separate hardware. This code was developed and written in conjunction with Bryce Baker, of the KU ISA Lab, in the PBasic language [3].

The Oregon Microsystems motion controller program was written as a standalone program in C++. While, the OMS motion control card is integrated into the computer controlling the Jaywalker, the libraries maintained by Oregon Microsystems are outdated, and thus are incompatible with the .Net framework that the final controller program was written in.

To facilitate the multiprocess application the OMS motion controller program was written to run as a background process with the motion commands passed from the main control program to the OMS motion controller program as command-line arguments.

2.6 USER INTERFACE

To allow for control and feedback from the Jaywalker a graphical user interface was developed using Microsoft Visual Studio and integrating this with the C# based Jaywalker control program. To conserve screen real estate and to minimize clutter while focusing on controlling or monitoring the Jaywalker a tabbed interface design paradigm was selected. The GUI includes seven tabs: Jaywalker, accelerometers, hip, thigh, knee, ankle, and foot.

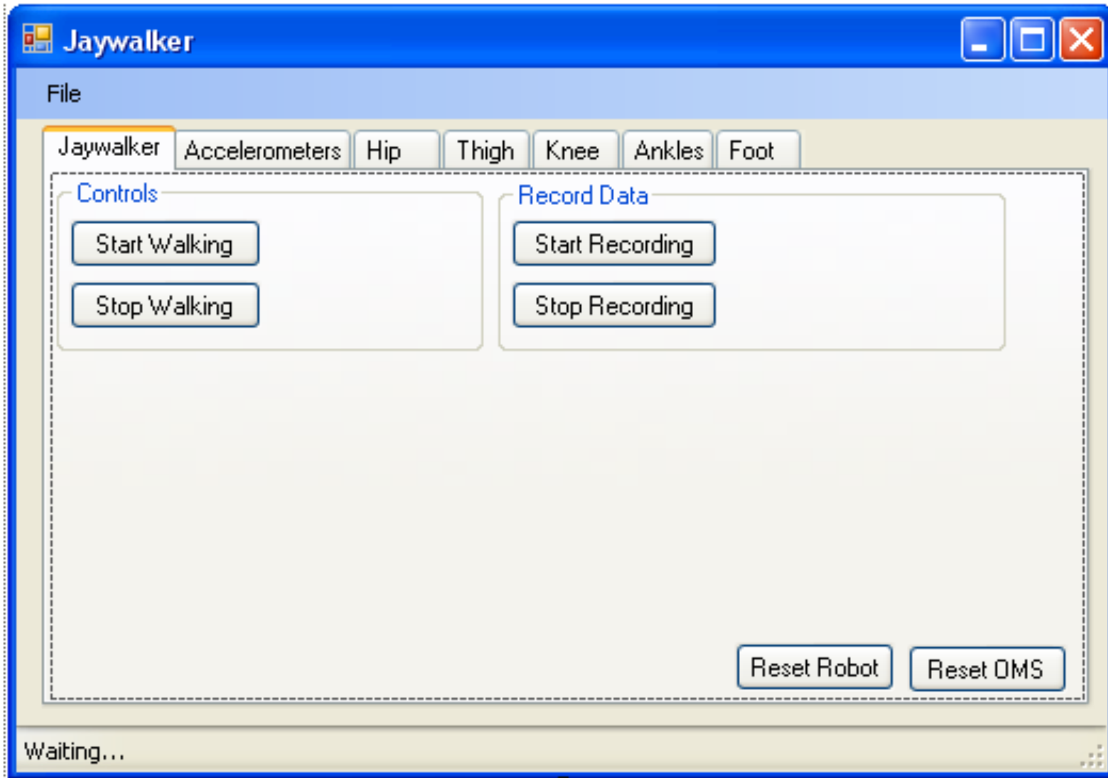

Figure 8: Main Jaywalker GUI Interface

The Jaywalker tab contains six inputs. Two inputs control the starting and stopping of the walking process. It signals the Jaywalker object to initiate the walk method or the stop method. Two inputs control the recording of the accelerometer data by signaling the Jaywalker object to trigger the StartRecording and the StopRecording methods. The final two controls act as resets for the jaywalker object and the OMS card.

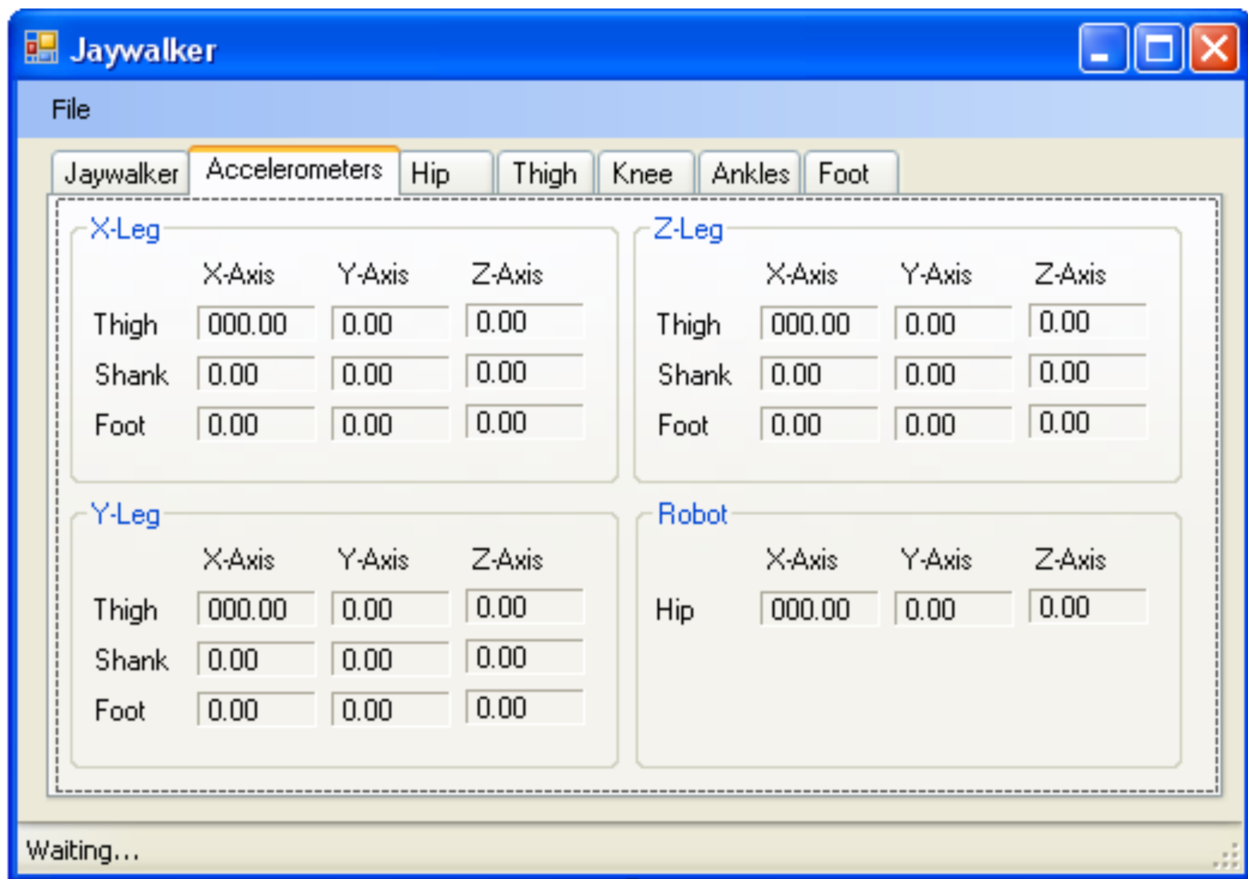

Figure 9: Jaywalker Accelerometer GUI Interface

The accelerometer tab displays the feedback from the ten accelerometers mounted on the Jaywalker. This allows for the user to observe the feedback while the Jaywalker is walking, and to observe trends or problems.

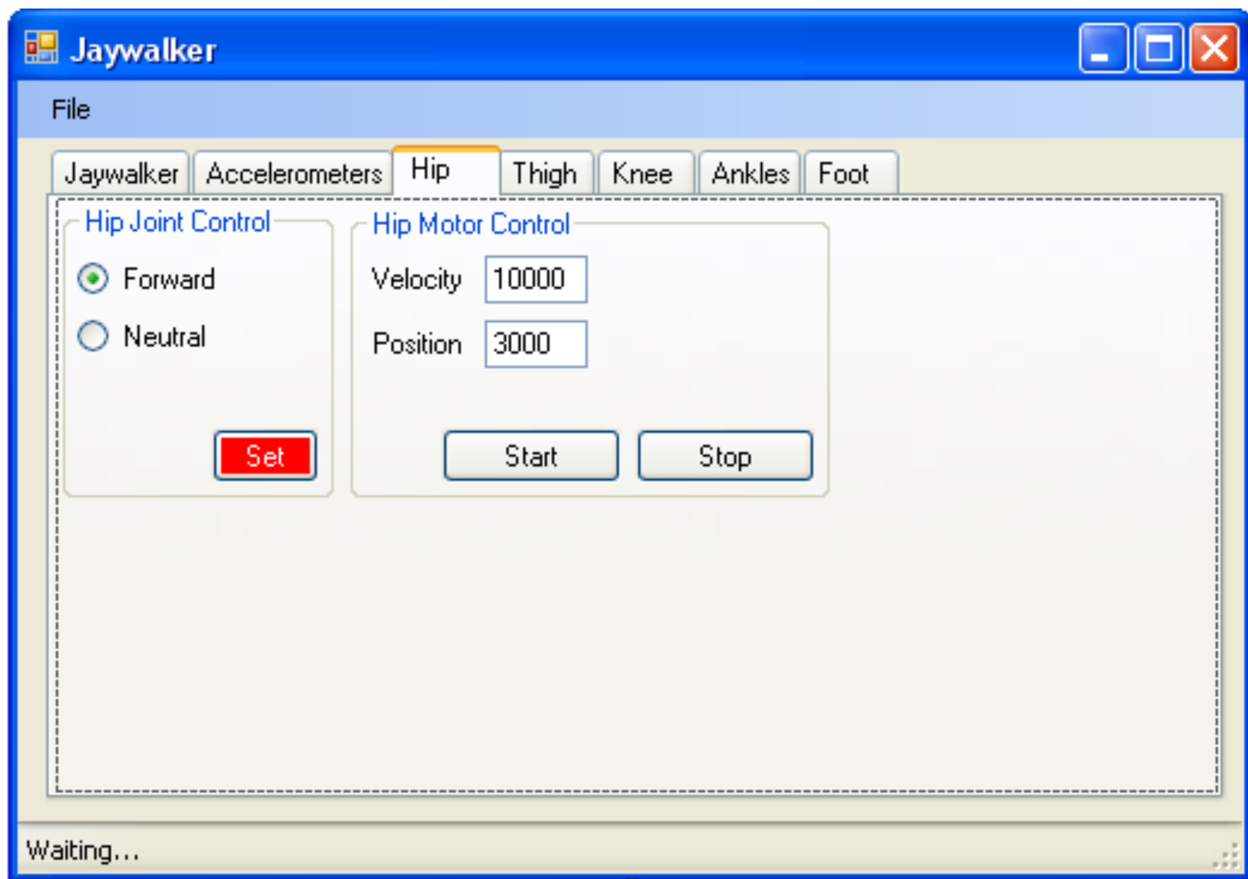

Figure 10: Jaywalker Hip GUI Interface

The hip tab allows for the selection of engaging the hip motors or allowing them to swing freely by triggering respectively the hip's Lock method or Neutral method. The hip tab also contains the controls necessary for setting the velocity and desired position of the hip motors along with the controls necessary for triggering the Actuate and Stop methods.

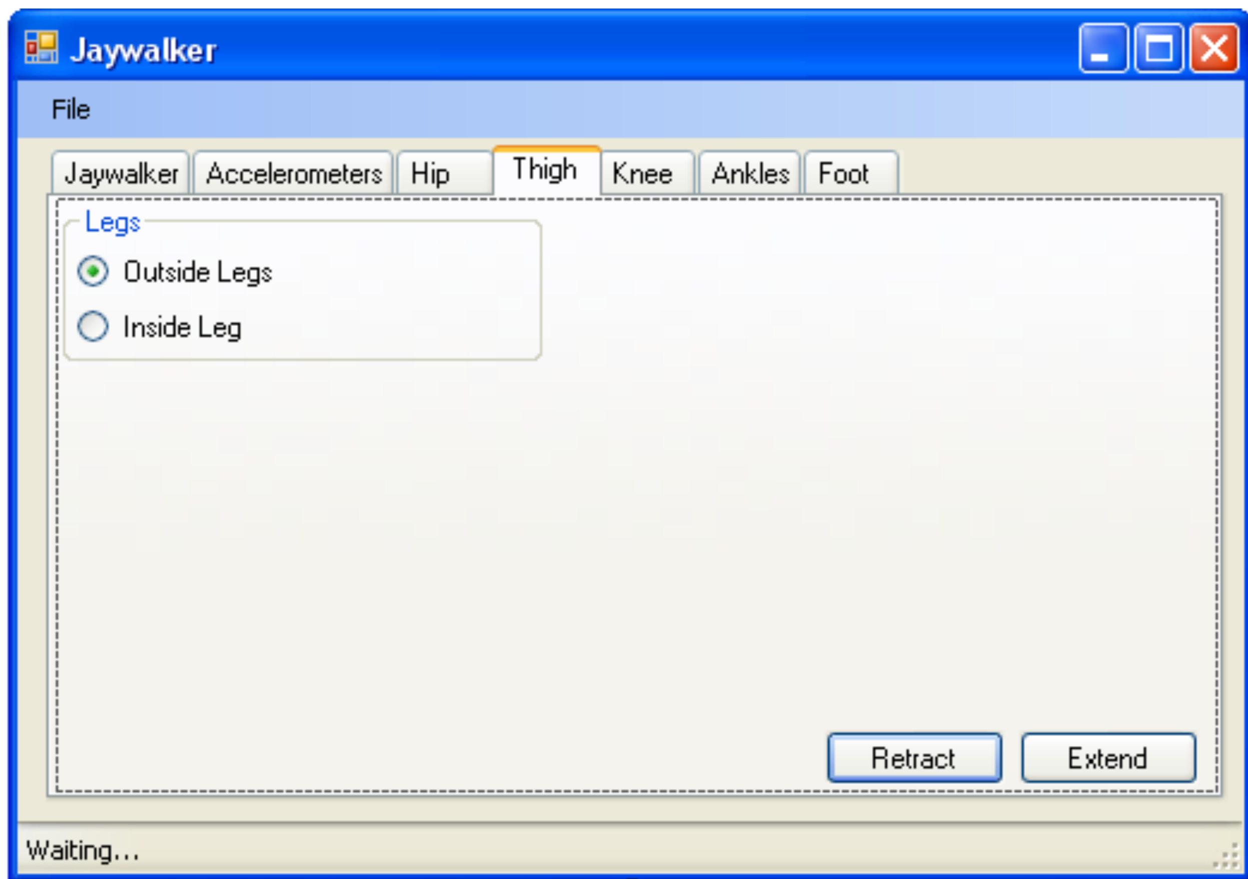

Figure 11: Jaywalker Thigh GUI Interface

The thigh tab contains the controls necessary for selecting the thigh(s) to actuate and the controls necessary for extending and retracting the selected thigh(s). These controls along with their related class methods are placeholders for future researchers to use as hooks into the control of the L.E.G.S. System, which was not completed at the end of testing.

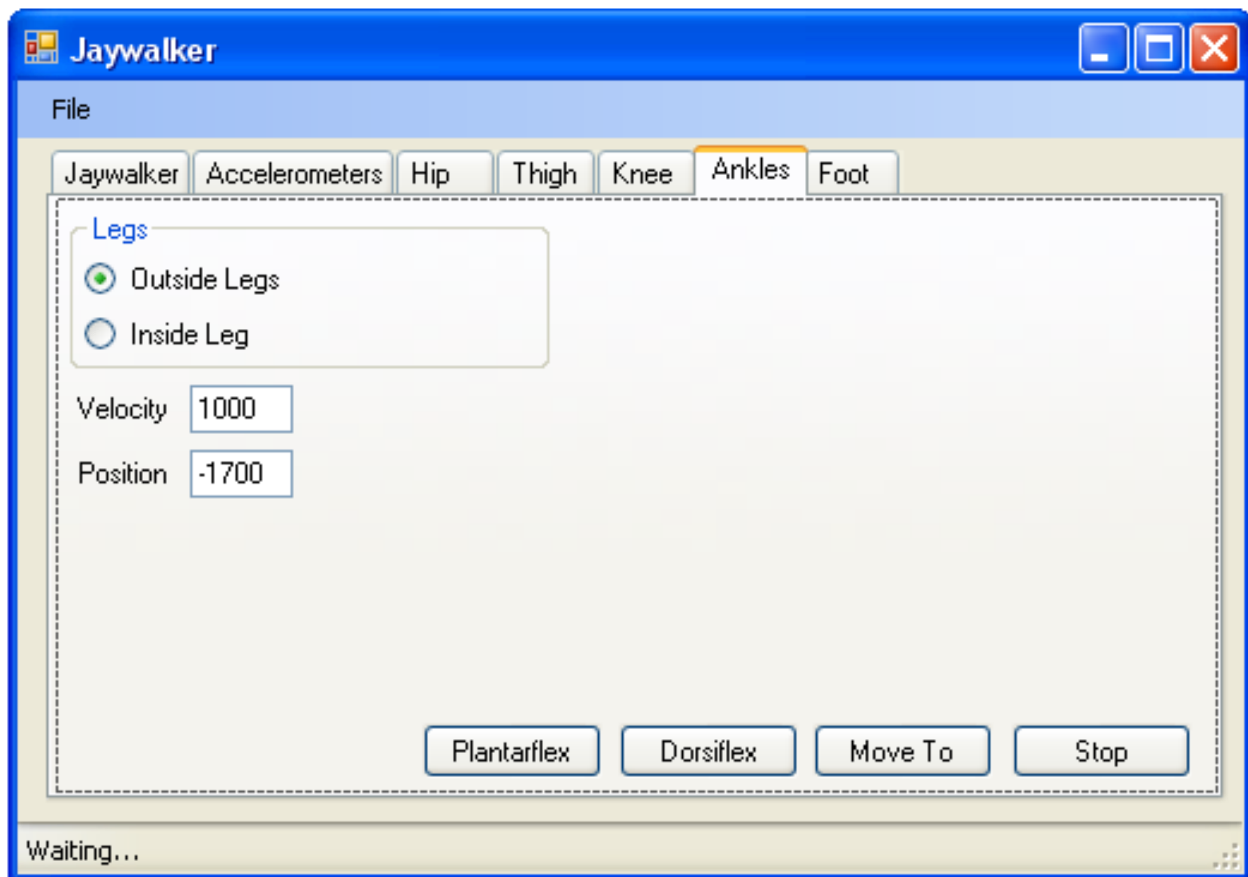

Figure 12: Jaywalker Knee GUI Interface

The knee tab contains the controls necessary for selecting the knee(s) to actuate and the controls necessary for extending and flexing the selected knee(s) through the triggering of the knee class's ExtendKnee and FlexKnee methods respectively.

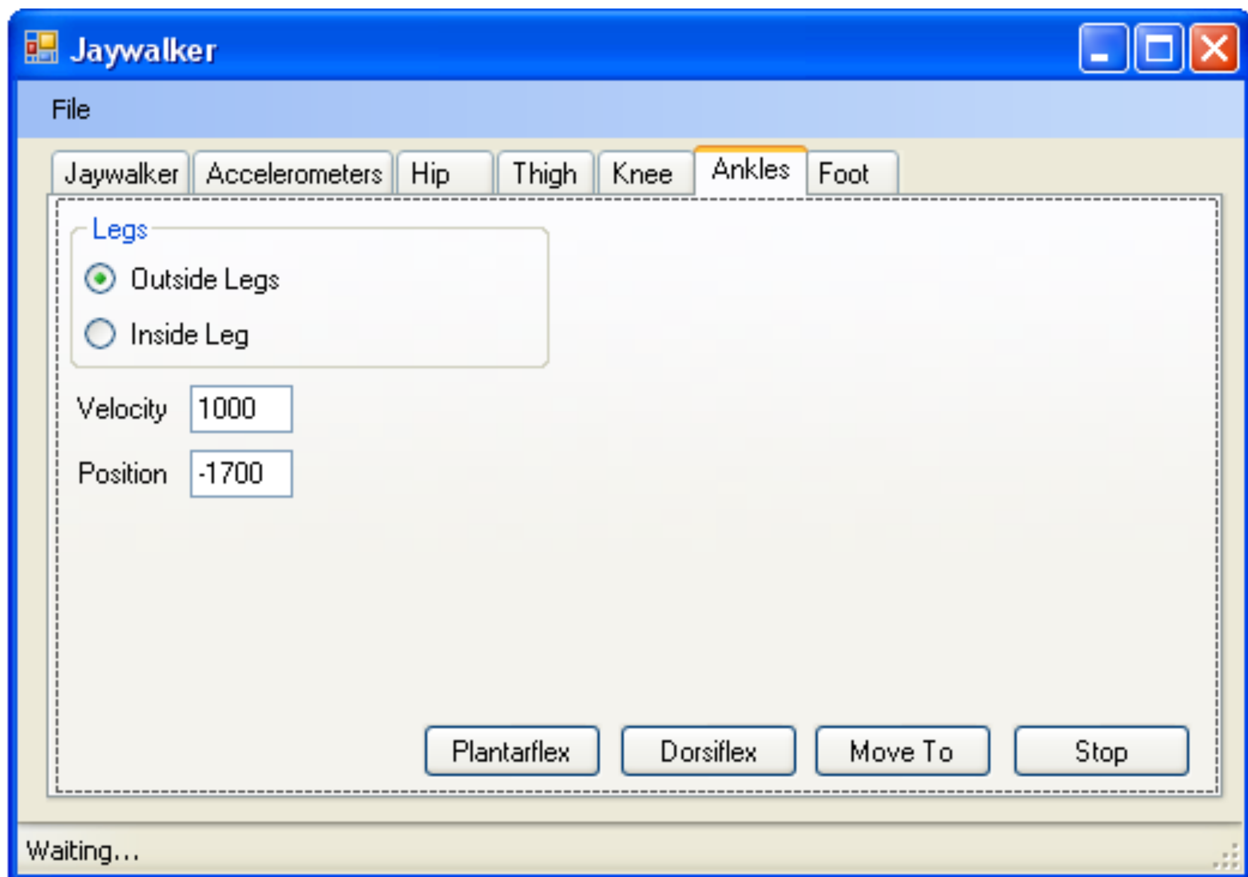

Figure 13: Jaywalker Ankle GUI Interface

The ankle tab contains the controls necessary to choose which ankles are going to be actuated, at what speed they are going to be actuated, and to what position the ankles will be moved, if it is appropriate. These controls function by interfacing with the ankle class's Plantarflex method, which takes a velocity value; the Dorsiflex method, which takes a velocity value; the MoveTo method, which takes both a position and velocity value; and the Stop method.

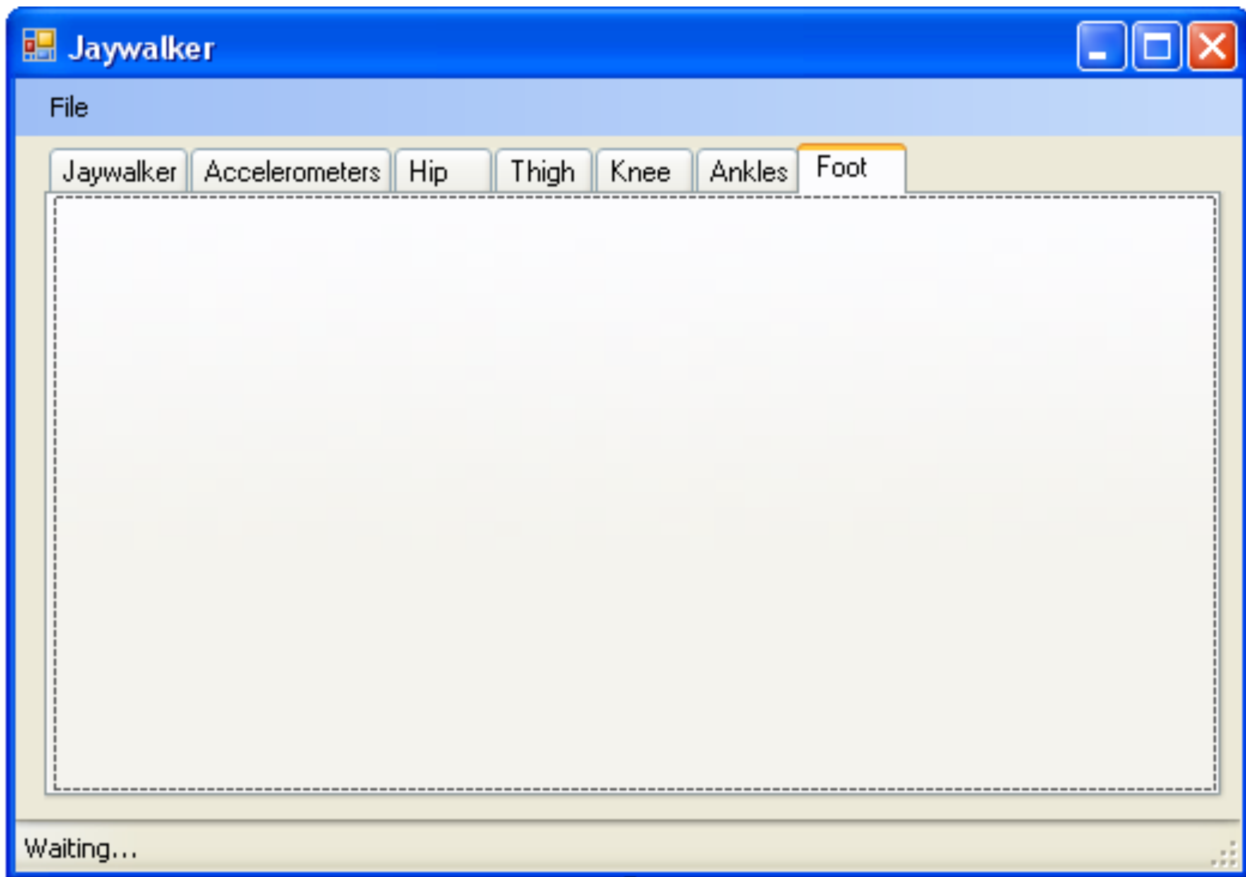

Figure 14: Jaywalker Foot Placeholder for future development

The foot tab is simply a placeholder for future development of foot actuation.

The final piece of the GUI is the status bar that runs along the bottom of the window. The status bar is used by various classes to display update information and during the initial testing of the control structure was often called into duty as a debugging tool for tracking the exact step of the control programming being executed.

3. TUNING AND TESTING OF THE CONTROL SYSTEM

3.1 TUNING VARIABLES

There are several variables which can be adjusted to produce a stable gait cycle: toe-off position of the back foot, initial step length (as measured toe to mid-foot), initial flexure of the

front foot, ankle velocity at toe-off, and position of the swing leg during knee extension. Of these variables, the control program sets or controls the initial flexure of the front foot, the ankle velocity at toe-off, and the position of the swing leg during knee extension.

The toe-off position of the back foot was controlled by the placement of the toe-off sensor on the foot. By shifting the sensor toward the toe the toe-off position becomes more plantarflexed during toe-off, and shifting the sensor toward the heel the toe-off position becomes less plantarflexed during toe-off. The toe-off position of the foot can shift the hip's mass relative to the front foot during the transition from stance to swing phase, thus effectively changing the length of the back leg during toe-off [3].

The initial step length was controlled manually before each test and works in concert with the toe-off position to produce a stable gait cycle [3]. Over the step trials this value ranged from 11.4 cm (4.50 in.) to 15.2 cm (6.00 in.).

The initial flexure of the front foot during the stance phase alters the effective height of the hip's mass and the step length, thus altering the necessary back leg length to produce a stable gait [3]. Over the step trials this value ranged from 0 steps (neutral) to 1750 steps (~21°) plantarflexion.

The ankle velocity at toe-off alters the energy input into the system and thus alters the velocity of the hip, which can affect the stability of the system [3].

The position of the swing leg during knee extension can occur at any time after the foot of the swing leg has cleared the ground. However, experimentation with the Jaywalker revealed that the momentum imparted into the system by the extension of the knee caused the swing leg's angle relative to horizontal to decrease. Therefore, the timing of the knee extension needs to

occur just before the point of maximum extension, which allows for the leg's forward momentum to negate the relative downward motion caused by the knee extension.

3.2 TESTING OVERVIEW

To best allow the Jaywalker to serve its purpose as a testbed for the development of an efficient bipedal walking robot many steps were taken in its construction to mimic humans and their highly efficient walking gate [12]. The first determination was to implement an intelligent control system [13 - 14] that would respond to the conditions of the Jaywalker instead of implementing a computationally intensive model based control system [15 - 16]. Due to the project's constraints a true fuzzy logic control system was not feasible, but a logic based control system implementing human experience in tuning several of the logical rules was within the project's parameters.

To allow for the tuning of the rules a systematic testing regimen was outlined which would allow for empirical feedback. The regimen consisted of a series of single step tests, followed by a series of double step tests, and ended with a series of continuous walking tests.

The single step tests consisted of a single step where the front leg's heel strike sensor was manually tripped, which lead to the back leg initiating a toe-off motion, followed by a knee flexure through the swing phase, and finally a knee extension and heel strike concluding the single step test. This test was designed to allow for the tuning of the rules concerning the initial step length and the initial flexure of the feet. A series of these tests were planned on both the inside and the outside legs to allow for any variation in the kinematics due to the 2D walker design implemented in the initial Jaywalker. If it was found that a stable and effective final step length could be achieved then this series of tests would be followed by the double step tests.

The double step tests consisted of two consecutive single step tests where the final position of the first step test became the initial conditions of the second step. This test was designed to allow for the tuning of the rules concerning the toe-off position of the back foot, ankle velocity at toe-off, and the position of the swing leg during knee extension. If it was found that the Jaywalker could execute a stable double stance after a single stride and that an adequate final step length could be achieved then this series of test would be followed by the continuous walking test.

The continuous walking test consisted of a continuous sequence of five strides (double step tests). This test was designed to allow for the final fine tuning in of any rules which might lead to accumulated errors over several strides.

Due to the curved nature of the Jaywalker's feet the heel of the front foot is usually not in contact with the ground, thus it was decided to forego the usual measure of the step length by measuring heel-to-heel of the feet and instead to measure from the heel of the front foot to the midpoint of the bottom of the back foot. This allowed for a more consistent and accurate step length measurement from trial to trial.

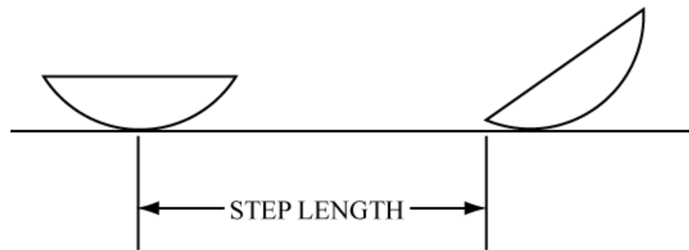

Figure 15: The methodology used to measure the step length during the Jaywalker testing.

3.3 SINGLE STEP TESTING

A total of 179 single step tests were performed, 50 with the outside legs transitioning through the swing phase and 129 with the inside leg transitioning through the swing phase. To aid in the

analysis of any failures and the effects of changing parameters, accelerometer data and/or video was recorded of each trial.

The first series of single step test performed were the 50 trials where the outside legs transitioned from the back legs through the swing phase to the front legs. Throughout these tests the initial step length was varied from 114.3 mm (4.5 in) to 196.9 mm (7.75 in). As shown in Table 3 the overall success rate for the single step test with the outside legs was 28.0%.

SINGLE STEP TEST RESULTS FOR THE OUTSIDE LEGS			
INITIAL STEP [mm (in)]	TOTAL TRIALS	SUCCESSFUL TRIALS	PERCENTAGE SUCCESS
114.3 (4.5)	1	0	0%
127 (5)	4	0	0%
139.7 (5.5)	15	6	40.0 %
146.1 (5.75)	8	3	37.5%
152.4 (6)	8	0	0%
158.8 (6.25)	1	0	0%
165.1 (6.5)	4	2	50.0%
171.5 (6.75)	3	1	33.3%
177.8 (7)	1	0	0%
190.5 (7.5)	2	0	0%
193.7 (7.625)	1	1	100.0%
195.7 (7.7)	1	1	100.0%
196.9 (7.75)	1	0	0%
TOTALS	50	14	28.0%

Table 3: Single step test results for the outside legs transitioning through the swing phase.

The second series of single step test performed were the 129 trials where the inside leg transitioned from the back leg through the swing phase to the front leg. Throughout these tests the initial step length was varied from 114.3 mm (4.5 in) to 152.4 mm (6.0 in). As shown in Table 4 the overall success rate for the single step test with the inside leg was 50.4%.

Throughout all of the tests the ankle velocity varied between 2500 and 5000 steps/s, the toe-off position varied between 6.4 mm (.25 in) and 8.9 mm (.35 in), and the initial plantarflexion was varied between 0 and 1750 steps. Over the testing the rules for the control system and the

SINGLE STEP TEST RESULTS FOR INSIDE LEG			
INITIAL STEP [mm (in)]	TOTAL TRIALS	SUCCESSFUL TRIALS	PERCENTAGE SUCCESS
114.3 (4.5)	1	0	0%
127 (5)	57	21	36.8%
133.4 (5.25)	66	42	63.6%
139.7 (5.5)	4	2	50.0%
152.4 (6)	1	0	0%
TOTALS	129	65	50.4%

Table 4: Single step test results for the inside leg transitioning through the swing phase.

testing parameters were finalized on the parameters which provided the highest probability of a successful step: an ankle velocity of 5000 steps/s, a toe-off position of 6.4 mm (.25 in) and initial plantarflexion of 0 steps. These were the parameters used for the last 41 trials of the single step tests.

In analyzing the video and data for the single step tests a few things became clear.

First, during the point in the swing phase where the knee extends in preparation for the heel strike the Jaywalker's thigh starts to extend until the leg comes into contact with the ground, thus ending the step. The result of this unnatural motion is that the terminal step length of the Jaywalker is shorter than in human walking since a human's knee extends to its maximum angle when the thigh reaches it most flexed position in the step cycle, at which point the human shifts their center of mass forward causing the heel of the forward leg to strike the ground [3].

Second, the Jaywalker's back leg maintains a fully extended position until toe-off and thus the Jaywalker does not flex the knee until the leg is in the swing phase. A human's knee, however, will begin to flex before the back leg toe-offs thus providing more energy to the swing leg [3].

Finally, the Jaywalker's lack of sufficient upper body counter weight coupled with the free swinging nature of the hip caused unnatural and undesirable hip flexion and extension during the

step cycle, at times leading to a leg becoming hyper-extended or under-extended. These instances of hyper and under extension caused short steps and overbalancing of the Jaywalker, which would cause to the tether to catch the Jaywalker.

Of special note was an occurrence that only presented itself when the Jaywalker took a step with the outside legs. Due to the 2D bipedal walker nature of the Jaywalker the existence of two actual legs for the outside leg motion led to a sufficient upward force during toe off that the forward, inside stance leg would lose contact with the floor and swing backwards a short distance. This in effect acted to shorten the initial step length of the back leg.

3.4 DOUBLE STEP TESTING

The double step test was devised to ascertain whether the terminal step length of a single step could adequately serve as the initial step length for a follow up step. After analyzing the results of the single step test it was decided that given the significantly higher success rate of the inside leg achieving a successful step and the desired terminal step length that the highest probability of success would occur if the inside leg was used as the back leg and thus was the first leg to transition through a swing phase.

Due to the necessity of acquiring accurate interim step lengths for the double step test a pause was incorporated into the test to allow for the testers to record the step length after the first step of the double step test. At the time of the data collection the dorsiflexion of all feet was adjusted to the desired values.

Throughout the double step tests the ankle velocity was set to 5000 steps/s and toe-off position was set to 6.4 mm (.25 in.) for all legs. The initial plantarflexion was set to 1700 steps

for the inside leg's transition through the swing phase and 0 steps for the outside legs' transition through the swing phase.

Over the total 84 double step tests ran only 7.1% of them ended in a successful double step for the Jaywalker. However, when considering only at the final 25 trials of the test, that success rate rose to 16.0%. While this increase in success rate over the length of the testing demonstrates that the control system increased in robustness, it also still shows that the current design of the Jaywalker with a passive hip is not capable of reaching an acceptable level of success as to warrant the continuous walking tests.

4. CONCLUSIONS

4.1 ANALYSIS OF RESULTS

While the Jaywalker bipedal walking test bed was not capable of adequately completing all of the desired goals due to the passive hip, it did demonstrate that a single step can successfully be taken with a terminal step length appropriate for a follow up step.

Analysis of where the Jaywalker excelled and where it failed pointed to the coupling of the inside leg to the hip, and thus its larger and higher mass acting as a counterbalance to the inside leg's motion. This is not an unexpected result given that the desire to create a walking bipedal robot capable of navigating uneven and rough terrain mandated that the natural dynamics of the passive walker, where the mass of the legs is inconsequential compared to the mass of the hip and upper body, could not be maintained because of the necessity for actuators in the legs.

Given the lack of this counterbalance for the outside legs it was thus shown that they were not capable of the same success as the more balanced inside leg. Couple this with the fact that

during the outside leg's transition through the swing phase the legs essentially have twice the mass of the single inside leg and the problem becomes even more pronounced.

4.2 RECOMMENDATIONS FOR FUTURE WORK

Given the success seen by the inside leg and the results of the testing conducted the following recommendations for future research include:

- The integration of proper counterbalancing for all legs to improve the terminal step length.
- To counteract the unnatural extension of the swing leg during knee extension future hip designs should incorporate some form of active or passive element to improve the terminal step length.
- To allow for more control and better response to changing conditions a compliant foot with a series of sensors place throughout should be added to allow for both the improved terminal step length and the better control of when to flex the knee on toe-off.
- To allow for better response to changing conditions the centralized personal computer control system running on a commercial operating system should be replaced with a custom distributed control system.

These recommendations should greatly improve the Jaywalker's ability to adjust for deviations in the initial step length, from the terminal step length of the previous step, and to react accordingly to any disturbances from uneven or rough terrain.

5. REFERENCES

[1] Raibert, M. H., 1986, Legged Robots That Balance, MIT Press, Cambridge, Mass., pp. 1-3, Chap. 1.

[2] Figlioni, G., Ceccarelli, M., 1999, “Walking programming for an electropneumatic biped robot,” Mechatronics, 9(8), pp. 941-964.

[3] Baker, B., 2010, “Development of a hybrid powered 2D biped walking machine designed for rough terrain locomotion,” Ph.D. thesis, the University of Kansas, Lawrence, KS.

[4] McGeer, T., 1990, "Passive walking with knees," Proceedings of the 1990 IEEE International Conference on Robotics and Automation, May 13-18 1990, Cincinnati, OH, USA, pp. 1640-1645.

[5] Winter, D., 1990, Biomechanics and Motor Control of Human Movement, Wiley, New York, N.Y., pp. 52-56, Chap. 3.

[6] Adamczyk, P. G., Collins, S. H. and Kuo, A. D., 2006, "The advantages of a rolling foot in human walking," The Journal of Experimental Biology 209(20), pp. 3953-3963.

[7] Inman, V., Ralston, H., Todd, F., and Lieberman, J., 1981, Human Walking, Williams & Wilkins, Baltimore, MD, pp. 26.

[8] Hirai, K., Hirose, M., Haikawa, Y. and Takenaka, T., 1998, "Development of Honda Humanoid Robot," Proceedings of the 1998 IEEE International Conference on Robotics and Automation. Part 2 (of 4), May 16-20 1998, Leuven, Belgium, 2, pp. 1321-1326.

[9] Honda, "ASIMO's Walking Control," at http://world.honda.com/ASIMO/technology/walking_02.html (last visited on September, 15 2011).

[10] McGeer, T., 1990, "Passive dynamic walking," International Journal of Robotics Research, 9(2), pp. 62-82.

[11] Collins, S., Ruina, A., Tedrake, R. and Wisse, M., 2005, "Efficient bipedal robots based on passive-dynamic walkers," Science, 37, pp. 1082-1085.

[12] Cunningham, C. B., Schilling, N., Anders, C. and Carrier, D. R., 2009, “The influence of foot posture on the cost of transport in humans,” The Journal of Experimental Biology 213, pp. 790-797.

[13] Bebek, B. and Erbatur, K., 2003, "A Fuzzy System for Gait Adaptation of Biped Walking Robots," in Proceedings of IEEE Conference on Control Applications (CCA), Istanbul, Turkey, 1, pp. 669-673.

[14] Tabrizi, S. S. and Bagheri, S., 2004, "Robustness of A biped robot controller developed by human expertise extraction against changes in terrain," Proceedings of the IASTED International Conference on Artificial Intelligence and Applications (as part of the 22nd IASTED International Multi-Conference on Applied Informatics, Febrary 16, 2004 - Febrary 18, 2004, Innsbruck, Austria, pp. 68-73.

[15] Kajita, S., Kanehiro, F., Kaneko, K., Fujiwara, K., Yokoi, K. and Hirukawa, H., 2003, "Biped walking pattern generation by a simple three-dimensional inverted pendulum model," Advanced Robotics, 17(Compendex), pp. 131-147.

[16] Denk, J. and Schmidt, G., 2003, "Synthesis of walking primitive databases for biped robots in 3D-environments," 2003 IEEE International Conference on Robotics and Automation, Sep 14-19 2003, Taipei, Taiwan, 1, pp. 1343-1349.

APPENDIX A: DESIGN SUPPLEMENT

A.1 C# CONTROL SYSTEM CODE

```
///////////////////////////////////////////////////////////////////
// Form1.cs
//
// Last Modified: 11/22/09
//
///////////////////////////////////////////////////////////////////

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.IO;
using System.Threading;
using System.Diagnostics;

using NationalInstruments.DAQmx;

namespace Jaywalker
{
   public partial class Form1 : Form
   {
      #region "Object Variables"
      //**********************//
      //    OBJECT VARIABLES   //
      //**********************//
```

```
private Robot _jaywalker;

Accelerometer sx;
Ankle outAnkle;
Ankle inAnkle;
Knee xKnee;
Knee yKnee;
Knee zKnee;
Hip outHip;

static object statusLock = new Object();
#endregion

#region "Constructor"
//***********************//
//     CONSTRUCTOR      //
//***********************//
/// <summary>
/// Default constructor for the main form of Jaywalker
/// </summary>
public Form1()
{
   InitializeComponent();
   _jaywalker = new Robot(this);

   sx = new Accelerometer(this);
   outAnkle = new Ankle(Constants.X_LEG);
   inAnkle = new Ankle(Constants.Y_LEG);
   xKnee = new Knee(Constants.X_LEG);
   yKnee = new Knee(Constants.Y_LEG);
   zKnee = new Knee(Constants.Z_LEG);
   outHip = new Hip(Constants.X_LEG);
}
#endregion
```

```
#region "Jaywalker Tab Controls"
//**********************//
// Jaywalker Tab Controls
//**********************//
private void btnStartWalk_Click(object sender, EventArgs e)
{
   btnStartWalk.Enabled = false;
   btnStopRecording.Enabled = true;
   _jaywalker.Walk();
}

private void btnStopWalk_Click(object sender, EventArgs e)
{
   _jaywalker.Stop();
}

private void btnStartRecord_Click(object sender, EventArgs e)
{
   btnStartRecord.Enabled = false;
   _jaywalker.StartRecording();
}

private void btnStopRecording_Click(object sender, EventArgs e)
{
   _jaywalker.StopRecording();
}

private void btnResetRobot_Click(object sender, EventArgs e)
{
   // Set current robot to null to delete
   _jaywalker = null;
   // Create a new robot
   _jaywalker = new Robot(this);
```

```
    // Renable buttons
    btnStartWalk.Enabled = true;
    btnStartRecord.Enabled = true;
}

private void btnReset_Click(object sender, EventArgs e)
{
    // Start OmsMove and pass the arguement string
    Process.Start("C:\\Jaywalker_Data\\OmsMove.exe", "r");
}
#endregion

#region "Accelerometer Controls"
//***********************//
//Accelerometer Controls //
//***********************//
/// <summary>
/// Displays the accelerometer data.
/// </summary>
/// <param name="data">The accelerometer data in an array.</param>
public void AccelDataUpdate(Position[] data)
{
    //UpdateAccelText(X_ThighX, data[Constants.X_THIGH_ACCEL].XAngle);
    //UpdateAccelText(X_ThighY, data[Constants.X_THIGH_ACCEL].YAngle);
    UpdateAccelText(X_ThighZ, data[Constants.X_THIGH_ACCEL].ZAngle);
    //UpdateAccelText(X_ShankX, data[Constants.X_SHANK_ACCEL].XAngle);
    //UpdateAccelText(X_ShankY, data[Constants.X_SHANK_ACCEL].YAngle);
    //UpdateAccelText(X_ShankZ, data[Constants.X_SHANK_ACCEL].ZAngle);
    //UpdateAccelText(X_FootX, data[Constants.X_FOOT_ACCEL].XAngle);
    //UpdateAccelText(X_FootY, data[Constants.X_FOOT_ACCEL].YAngle);
    //UpdateAccelText(X_FootZ, data[Constants.X_FOOT_ACCEL].ZAngle);

    //UpdateAccelText(Y_ThighX, data[Constants.Y_THIGH_ACCEL].XAngle);
    //UpdateAccelText(Y_ThighY, data[Constants.Y_THIGH_ACCEL].YAngle);
```

```
    UpdateAccelText(Y_ThighZ, data[Constants.Y_THIGH_ACCEL].ZAngle);
    //UpdateAccelText(Y_ShankX, data[Constants.Y_SHANK_ACCEL].XAngle);
    //UpdateAccelText(Y_ShankY, data[Constants.Y_SHANK_ACCEL].YAngle);
    //UpdateAccelText(Y_ShankZ, data[Constants.Y_SHANK_ACCEL].ZAngle);
    //UpdateAccelText(Y_FootX, data[Constants.Y_FOOT_ACCEL].XAngle);
    //UpdateAccelText(Y_FootY, data[Constants.Y_FOOT_ACCEL].YAngle);
    //UpdateAccelText(Y_FootZ, data[Constants.Y_FOOT_ACCEL].ZAngle);

    //UpdateAccelText(Z_ThighX, data[Constants.Z_THIGH_ACCEL].XAngle);
    //UpdateAccelText(Z_ThighY, data[Constants.Z_THIGH_ACCEL].YAngle);
    //UpdateAccelText(Z_ThighZ, data[Constants.Z_THIGH_ACCEL].ZAngle);
    //UpdateAccelText(Z_ShankX, data[Constants.Z_SHANK_ACCEL].XAngle);
    //UpdateAccelText(Z_ShankY, data[Constants.Z_SHANK_ACCEL].YAngle);
    //UpdateAccelText(Z_ShankZ, data[Constants.Z_SHANK_ACCEL].ZAngle);
    //UpdateAccelText(Z_FootX, data[Constants.Z_FOOT_ACCEL].XAngle);
    //UpdateAccelText(Z_FootY, data[Constants.Z_FOOT_ACCEL].YAngle);
    //UpdateAccelText(Z_FootZ, data[Constants.Z_FOOT_ACCEL].ZAngle);

    //UpdateAccelText(Robot_HipX, data[Constants.HIP_ACCEL].XAngle);
    //UpdateAccelText(Robot_HipY, data[Constants.HIP_ACCEL].YAngle);
    //UpdateAccelText(Robot_HipZ, data[Constants.HIP_ACCEL].ZAngle);
}

private void UpdateAccelText(Label label, double data)
{
    lock (label)
    {
        if (label.InvokeRequired)
        {
            label.Invoke((MethodInvoker)delegate()
            {
                label.Text = data.ToString();
                label.Update();
            });
```

```
        }
        else
        {
            label.Text = data.ToString();
            label.Update();
        }
    }
}
#endregion

#region "Hip Controls"
private void btnSetHipState_Click(object sender, EventArgs e)
{
    // Check which radio button is selected
    if (radioHipForward.Checked == true)
    {
        outHip.Lock();
    }
    else
    {
        outHip.Neutral();
    }
}

private void btnHipStart_Click(object sender, EventArgs e)
{
    long pos;
    long vel;

    // Convert string to numbers
    long.TryParse(txtHipPosition.Text, out pos);
    long.TryParse(txtHipVelocity.Text, out vel);

    outHip.Actuate(pos, vel);
```

```
}

private void btnHipStop_Click(object sender, EventArgs e)
{
   outHip.Stop();
}
#endregion

#region "Ankle Controls"
//**********************//
//     Ankle Controls    //
//**********************//
/// <summary>
/// Plantarflexes the foot to the limit.
/// </summary>
/// <param name="sender">The Plantarflex button.</param>
/// <param name="e">The event arguement.</param>
private void btnPlantarflex_Click(object sender, EventArgs e)
{
   long velocity;

   if (long.TryParse(txtVelocity.Text, out velocity))
   {
      if (radioOutsideAnkles.Checked == true)
      {
         outAnkle.Plantarflex(velocity);
      }
      else
      {
         inAnkle.Plantarflex(velocity);
      }
   }
   else
   {
```

```
        routineStatus.Text = "Invalid velocity entered";
        statusStrip1.Update();
    }
}

/// <summary>
/// Dorsiflexes the foot to the limit
/// </summary>
/// <param name="sender">The Dorsiflex button.</param>
/// <param name="e">The event arguement.</param>
private void btnDorsiflex_Click(object sender, EventArgs e)
{
    long velocity;

    if (long.TryParse(txtVelocity.Text, out velocity))
    {
        if (radioOutsideAnkles.Checked == true)
        {
            outAnkle.Dorsiflex(velocity);
        }
        else
        {
            inAnkle.Dorsiflex(velocity);
        }
    }
    else
    {
        routineStatus.Text = "Invalid velocity entered";
        statusStrip1.Update();
    }
}

/// <summary>
/// Moves the foot to the stored position.
```

```
/// </summary>
/// <param name="sender">The Move To button.</param>
/// <param name="e">The event arguement.</param>
private void btnMoveTo_Click(object sender, EventArgs e)
{
    long velocity;
    long position;

    if (long.TryParse(txtVelocity.Text, out velocity) &&
        long.TryParse(txtPosition.Text, out position))
    {
        if (radioOutsideAnkles.Checked == true)
        {
            outAnkle.MoveTo(position, velocity);
        }
        else
        {
            inAnkle.MoveTo(position, velocity);
        }
    }
    else
    {
        routineStatus.Text = "Invalid velocity or position entered";
        statusStrip1.Update();
    }
}

/// <summary>
/// Stops the foot.
/// </summary>
/// <param name="sender">The Stop button.</param>
/// <param name="e">The event arguement.</param>
private void btnStop_Click(object sender, EventArgs e)
{
```

```
    if (radioOutsideAnkles.Checked == true)
    {
        outAnkle.Stop();
    }
    else
    {
        inAnkle.Stop();
    }
}
#endregion

#region "Knee Controls"
//**********************//
//Knee Controls
//**********************//
/// <summary>
/// Flexes the knee.
/// </summary>
/// <param name="sender">The Flex Knee button.</param>
/// <param name="e">The event arguement.</param>
private void btnFlexKnee_Click(object sender, EventArgs e)
{
    if (radioOutsideKnees.Checked == true)
    {
        xKnee.FlexKnee();
        zKnee.FlexKnee();
    }
    else
    {
        yKnee.FlexKnee();
    }

}
```

```
/// <summary>
/// Extends the knee.
/// </summary>
/// <param name="sender">The Extend Knee button.</param>
/// <param name="e">The event arguement.</param>
private void btnExtendKnee_Click(object sender, EventArgs e)
{
   if (radioOutsideKnees.Checked == true)
   {
      xKnee.ExtendKnee();
      zKnee.ExtendKnee();
   }
   else
   {
      yKnee.ExtendKnee();
   }
}
#endregion

#region "Foot Controls"
//**********************//
//Foot Controls
//**********************//
#endregion

#region "Status Bar Methods"
/// <summary>
/// Writes to the status strip and updates.
/// </summary>
/// <param name="Display">The string to be displayed in the statusStrip.</param>
public void StatusUpdate(string Display)
{
   lock (statusLock)
   {
```

```
                if (statusStrip1.InvokeRequired)
                {
                    statusStrip1.Invoke((MethodInvoker)delegate() {
                        routineStatus.Text = Display;
                        statusStrip1.Update();
                    });
                }
                else
                {
                    routineStatus.Text = Display;
                    statusStrip1.Update();
                }
            }
        }
        #endregion
    }
}

////////////////////////////////////////////////////////////////////////
// Robot.cs
//
// Last Modified: 3/14/10
//
////////////////////////////////////////////////////////////////////////

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.IO;

namespace Jaywalker
{
```

```
/// <summary>
/// Robot is the root class for the jaywalker.
/// </summary>
class Robot
{
    #region "Object Variables"
    //**********************//
    //    CLASS VARIABLES    //
    //**********************//
    private Leg _xLeg;
    private Thread _xThread;
    private Leg _yLeg;
    private Thread _yThread;
    private MPU _ni;
    private Thread _niThread;
    private bool _recordFlag;
    private Thread _recordThread;
    private Form1 _frm;
    private bool _dorsiFlag;
    #endregion

    #region "Constructor"
    //**********************//
    //      CONSTRUCTOR      //
    //**********************//
    public Robot(Form1 frm)
    {
        _xLeg = new Leg(frm, Constants.X_LEG);
        _xThread = new Thread(new ThreadStart(_xLeg.Actuate));
        _xLeg.SetRobot(this);

        _yLeg = new Leg(frm, Constants.Y_LEG);
        _yThread = new Thread(new ThreadStart(_yLeg.Actuate));
        _yLeg.SetRobot(this);
```

```
    _ni = new MPU(this, frm);
    _niThread = new Thread(new ThreadStart(_ni.ReadData));

    _recordFlag = false;
    _recordThread = new Thread(new ThreadStart(RecordData));

    _dorsiFlag = false;
    _frm = frm;
}
#endregion

#region "Accessor Methods"
//**********************//
//   ACCESSOR METHODS    //
//**********************//
public bool dorsiFlag
{
    get { return _dorsiFlag; }
    set { _dorsiFlag = value; }
}
#endregion

#region "Data Methods"
//**********************//
//     DATA METHODS      //
//**********************//
public void DistributeData(Position[] data)
{
    _xLeg.SetThighAngle(data[Constants.X_THIGH_ACCEL].ZAngle);
    _xLeg.SetShankAngle(data[Constants.X_SHANK_ACCEL].YAngle);
    _xLeg.SetFootAngle(data[Constants.X_FOOT_ACCEL].ZAngle);
    _yLeg.SetThighAngle(data[Constants.Y_THIGH_ACCEL].ZAngle);
    _yLeg.SetShankAngle(data[Constants.Y_SHANK_ACCEL].YAngle);
```

```
        _yLeg.SetFootAngle(data[Constants.Y_FOOT_ACCEL].ZAngle);
    }

    private void RecordData()
    {
        // Declare Variables
        FileStream fs = File.Open("C:\\Jaywalker_Data\\test.txt", FileMode.Create,
FileAccess.Write);
        StreamWriter sw = new StreamWriter(fs);
        long start = DateTime.Now.Ticks;
        long oldTime = start;
        long newTime;

        // Set the record flag to true
        _recordFlag = true;

        do
        {
            newTime = DateTime.Now.Ticks;
            if (newTime != oldTime)
            {
                sw.Write((newTime));
                sw.WriteLine("\t{0}\t{1}", _xLeg.GetThighAngle(), _yLeg.GetThighAngle());
                oldTime = newTime;
            }
        } while (_recordFlag);

        sw.Close();
        sw = null;
    }

    public void StartRecording()
    {
        if (_recordThread.IsAlive == false)
```

```
    {
        _recordThread.Start();
    }
}

public void StopRecording()
{
    _recordFlag = false;
    _recordThread.Abort();
}
#endregion

#region "Movement Methods"
//**********************//
//   MOVEMENT METHODS    //
//**********************//
/// <summary>
/// Robot.Walk starts the robot walking.
/// </summary>
public void Walk()
{
    _frm.StatusUpdate("Start Walking");
    _niThread.Start();
    _xThread.Start();
    _yThread.Start();
}

/// <summary>
/// Robot.Stop stops the robot.
/// </summary>
public void Stop()
{
    _frm.StatusUpdate("Stop Walking");
    _ni.Stop();
```

```
            _xLeg.Stop();
            _yLeg.Stop();
            _niThread.Abort();
            _xThread.Abort();
            _yThread.Abort();
        }
        #endregion
    }
}

//////////////////////////////////////////////////////////////////////
// Hip.cs
//
// Last Modified: 11/22/09
//
//////////////////////////////////////////////////////////////////////

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Diagnostics;

using NationalInstruments.DAQmx;

namespace Jaywalker
{
    /// <summary>
    /// Hip is the class that controls the hip.
    /// </summary>
    class Hip
    {
        #region "Object Variables"
        //**********************//
```

```
//    OBJECT VARIABLES   //
//**********************//
private long _leg;
private bool _hipActuateFlag;
private Task hipTask;
private DigitalSingleChannelWriter hipChannel;
#endregion

#region "Constructor"
//**********************//
//      CONSTRUCTOR      //
//**********************//
public Hip(long leg)
{
   _hipActuateFlag = false;
   string port = "";
   _leg = leg;

   switch (leg)
   {
      case Constants.X_LEG:
         port = "Dev1/port8/line7";
         break;
      case Constants.Y_LEG:
         port = "Dev1/port8/line6";
         break;
      case Constants.Z_LEG:
         port = "Dev1/port8/line7";
         break;
   }

   hipTask = new Task();
   hipTask.DOChannels.CreateChannel(
      port,
```

```
        "hipChannel",
        ChannelLineGrouping.OneChannelForAllLines);
    hipChannel = new DigitalSingleChannelWriter(hipTask.Stream);

    hipChannel.WriteSingleSampleSingleLine(true, Constants.LOW);
}
#endregion

#region "Sensing Methods"
//**********************//
//    SENSING METHODS    //
//**********************//
////////////////////////////////////////
/// <summary>
/// Hip.GetAngle returns the tilt angle of the hip.
/// </summary>
/// <returns>A double representing the tilt of the hip.</returns>
////////////////////////////////////////
public double GetAngle()
{
    //TODO: ADD CODE HERE
    //to read the tilt from the accelerometer
    return 0.0;
}

#endregion

#region "Movement Methods"
//**********************//
//   Movement Methods    //
//**********************//
public void Actuate(long pos, long vel)
{
    // Ignore this command for the Y-Leg
```

```
    if (_leg == Constants.X_LEG)
    {
        // Generate arguement string
        string args = "h " + vel.ToString() + " " + pos.ToString();

        // Start OmsMove and pass the arguement string
        Process.Start("C:\\Jaywalker_Data\\OmsMove.exe", args);
    }
}

public void Stop()
{
    // Ignore this command for the Y-Leg
    if (_leg == Constants.X_LEG)
    {
        // Generate arguement string
        string args = "k";

        // Start OmsMove and pass the arguement string
        Process.Start("C:\\Jaywalker_Data\\OmsMove.exe", args);
    }
}

////////////////////////////////////////////
// Hip Neutral Method
/// <summary>
/// Hip.Neutral allows the thigh to free swing
/// </summary>
////////////////////////////////////////////
public void Neutral()
{
    hipChannel.WriteSingleSampleSingleLine(true, Constants.LOW);
}
```

```
        ///////////////////////////////////////
        // Hip Lock Method
        /// <summary>
        /// Hip.Lock engages the thighs to the hip
        /// </summary>
        ///////////////////////////////////////
        public void Lock()
        {
            hipChannel.WriteSingleSampleSingleLine(true, Constants.HIGH);
        }
        #endregion
    }
}

////////////////////////////////////////////////////////////////
// Leg.cs
//
// Last Modified: 12/06/09
//
////////////////////////////////////////////////////////////////

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

using NationalInstruments.DAQmx;

namespace Jaywalker
{
    class Leg
    {
        #region "Object Variables"
```

```
//**********************//
//    CLASS VARIABLES    //
//**********************//
private long _leg;
private bool _stancePhase;
private bool _walkFlag;
private Thigh _thigh;
private Knee _knee;
private Shank _shank;
private Ankle _ankle;
private Foot _foot;
private Toe _toe;
private Hip _hip;

private Form1 _frm;
private Robot _jw;

private Task sensorTactileTask;
private DigitalSingleChannelReader sensorTactileChannel;
#endregion

#region "Constructor"
//**********************//
//     CONSTRUCTOR      //
//**********************//
public Leg(Form1 frm, long leg)
{
   _stancePhase = false;
   _leg = leg;
   _walkFlag = false;
   _thigh = new Thigh(frm, leg);
   _knee = new Knee(leg);
   _shank = new Shank(frm, leg);
   _ankle = new Ankle(leg);
```

```
    _foot = new Foot(frm, leg);
    _toe = new Toe(leg);
    _hip = new Hip(leg);
    _frm = frm;

    string port = "";

    switch (leg)
    {
        case Constants.X_LEG:
            port = "Dev1/port7/line0";
            break;
        case Constants.Y_LEG:
            port = "Dev1/port6/line4";
            break;
    }

    sensorTactileTask = new Task();
    sensorTactileTask.DIChannels.CreateChannel(
        port,
        "sensorTactileChannel",
        ChannelLineGrouping.OneChannelForAllLines);
    sensorTactileChannel = new DigitalSingleChannelReader(sensorTactileTask.Stream);
}
#endregion

#region "Accessor Methods"
//***********************//
//   ACCESSOR METHODS    //
//***********************//
public long leg
{
    get { return _leg; }
    set { _leg = value; }
```

```
}

public bool StancePhase
{
   get { return _stancePhase; }
   set { _stancePhase = value; }
}

public void SetRobot(Robot j)
{
   _jw = j;
}
#endregion

#region "Sensing Methods"
//***********************//
//    SENSING METHODS    //
//***********************//
public double GetThighAngle()
{
   return _thigh.GetAngle();
}

public void SetThighAngle(double angle)
{
   _thigh.SetAngle(angle);
}

public double GetShankAngle()
{
   return _shank.GetAngle();
}

public void SetShankAngle(double angle)
```

```
{
   _shank.SetAngle(angle);
}

public double GetFootAngle()
{
   return _foot.GetAngle();
}

public void SetFootAngle(double angle)
{
   _foot.SetAngle(angle);
}

public bool GetHeelStrike()
{
   return _foot.GetHeelStrike();
}

public bool GetToeOff()
{
   return _foot.GetToeOff();
}
#endregion

#region "Movement Methods"
//***********************//
//   Movement Methods    //
//***********************//
public void Actuate()
{
   _walkFlag = true;
   bool heelFlag = true;
   bool toeFlag = false;
```

```
bool resetFlag = true;

do
{
    // Check for heel strike sensor
    if (_foot.GetHeelStrike() && heelFlag)
    {
        _hip.Neutral();
        switch (leg)
        {
            case Constants.X_LEG:
                _ankle.Plantarflex(4000);
                break;
            case Constants.Y_LEG:
                _ankle.Plantarflex(5000);
                break;
        }
        _jw.StartRecording();
        heelFlag = false;
        toeFlag = true;
        resetFlag = true;
        Console.WriteLine(_leg.ToString() + " Heel Strike");
    }

    // Check toeoff sensor
    if (_foot.GetToeOff() && toeFlag)
    {
        _knee.FlexKnee();
        _hip.Neutral();
        _ankle.Dorsiflex(7000);
        heelFlag = true;
        resetFlag = true;
        toeFlag = false;
        Console.WriteLine(_leg.ToString() + " Toe Off");
```

```
        }

        // Check for heel strike sensor
        if (sensorTactileChannel.ReadSingleSampleSingleLine() && resetFlag)
        {
            if (_leg == Constants.X_LEG)
            {
                _ankle.StepReset(1000);
                Console.WriteLine(_leg.ToString() + " Reset for Step");
                resetFlag = false;
            }
        }

        // Check leg position
        if (_thigh.GetAngle() <= 20  && _leg == Constants.X_LEG)
        {
            _knee.ExtendKnee();
            _hip.Lock();
        }

        if (_thigh.GetAngle() <= 0 && _leg == Constants.Y_LEG)
        {
            _knee.ExtendKnee();
            _hip.Lock();
        }

    }while(_walkFlag);
}

public void Stop()
{
    _walkFlag = false;
}
#endregion
```

```
    }
}


//////////////////////////////////////////////////////////////////////
// Thigh.cs
//
// Last Modified: 3/14/10
//
//////////////////////////////////////////////////////////////////////

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Jaywalker
{
    class Thigh
    {
        #region "Object Variables"
        //**********************//
        //    CLASS VARIABLES    //
        //**********************//
        private long _leg;

        private double _previousAngle;
        private double _angle;
        private bool _isExtended;

        Form1 _frm;
        #endregion

        #region "Constructor"
```

```
//**********************//
//     CONSTRUCTOR     //
//**********************//
public Thigh(Form1 frm, long leg)
{
   _leg = leg;
   _angle = 0.0;
   _previousAngle = Constants.MAX_ACCEL + 100.0;
   _isExtended = false;

   _frm = frm;
}
#endregion

#region "Accessor Methods"
//**********************//
//   ACCESSOR METHODS   //
//**********************//
public long Leg
{
   get { return _leg; }
   set { _leg = value; }
}

public bool IsExtended
{
   get { return _isExtended; }
   // no setter method
}

public void SetAngle(double currentAngle)
{
   double average;
   long threshold = Constants.THRESHOLD;
```

```
    // Check for first reading
    if (_previousAngle > Constants.MAX_ACCEL)
    {
        _previousAngle = currentAngle;
        average = Constants.MAX_ACCEL + 100.0;
    }
    else
    {
        // check for a change in position beyond the threshold
        if ((_previousAngle - currentAngle) > threshold)
        {
            currentAngle = _previousAngle - threshold;
        }
        else if ((currentAngle - _previousAngle) > threshold)
        {
            currentAngle = _previousAngle + threshold;
        }

        // average the previous angle with the current
        average = (currentAngle + _previousAngle) / 2;

        // store the new previous angle
        _previousAngle = currentAngle;
    }

    _angle = average;

    //_angle = currentAngle;
}
#endregion

#region "Sensing Methods"
//**********************//
```

```
//    SENSING METHODS    //
//***********************//
/// <summary>
/// Thigh.GetAngle returns the tilt angle of the thigh.
/// </summary>
/// <returns>A double representing the tilt of the thigh.</returns>
public double GetAngle()
{
   return _angle;
}
#endregion

#region "Movement Methods"
//***********************//
//   Movement Methods    //
//***********************//
////////////////////////////////////////////
/// <summary>
/// Thigh.Extend extends the thigh.
/// </summary>
////////////////////////////////////////////
public void Extend()
{
   //TODO: ADD CODE HERE
}

////////////////////////////////////////////
/// <summary>
/// Thigh.Retract retracts the thigh.
/// </summary>
////////////////////////////////////////////
public void Retract()
{
   //TODO: ADD CODE HERE
```

```
        }
        #endregion
    }
}


/////////////////////////////////////////////////////////////////////
// Knee.cs
//
// Last Modified: 9/26/09
//
/////////////////////////////////////////////////////////////////////

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using NationalInstruments.DAQmx;

namespace Jaywalker
{
    class Knee
    {
        #region "Object Variables"
        //***********************//
        //    CLASS VARIABLES    //
        //***********************//
        private long _leg;
        private bool _isExtended;
        private Task solenoidTask;
        private DigitalSingleChannelWriter solenoidChannel;
        #endregion
```

```
#region "Constructor"
//**********************//
//      CONSTRUCTOR      //
//**********************//
public Knee(long leg)
{
   string port = "";
   _leg = leg;

   switch (leg)
   {
      case Constants.X_LEG:
         port = "Dev1/port3/line0, Dev1/port3/line2";
         break;
      case Constants.Y_LEG:
         port = "Dev1/port3/line1";
         break;
      case Constants.Z_LEG:
         port = "Dev1/port3/line0, Dev1/port3/line2";
         break;
   }

   solenoidTask = new Task();
   solenoidTask.DOChannels.CreateChannel(
      port,
      "solenoidChannel",
      ChannelLineGrouping.OneChannelForAllLines);
   solenoidChannel = new DigitalSingleChannelWriter(solenoidTask.Stream);
}
#endregion

#region "Accessor Methods"
//**********************//
//   ACCESSOR METHODS    //
```

```
//**********************//
public long Leg
{
    get { return _leg; }
    set { _leg = value; }
}

public bool IsExtended
{
    get { return _isExtended; }
    // no setter method
}
#endregion

#region "Movement Methods"
//**********************//
//   Movement Methods    //
//**********************//
public void FlexKnee()
{
    bool[] onArray = { Constants.HIGH, Constants.HIGH };

    switch (_leg)
    {
        case Constants.X_LEG:
            solenoidChannel.WriteSingleSampleMultiLine(true, onArray);
            break;
        case Constants.Y_LEG:
            solenoidChannel.WriteSingleSampleSingleLine(true, Constants.HIGH);
            break;
        case Constants.Z_LEG:
            solenoidChannel.WriteSingleSampleMultiLine(true, onArray);
            break;
    }
```

```
            _isExtended = true;
        }

        public void ExtendKnee()
        {
            bool[] onArray = { Constants.LOW, Constants.LOW };

            switch (_leg)
            {
                case Constants.X_LEG:
                    solenoidChannel.WriteSingleSampleMultiLine(true, onArray);
                    break;
                case Constants.Y_LEG:
                    solenoidChannel.WriteSingleSampleSingleLine(true, Constants.LOW);
                    break;
                /*case Constants.Z_LEG:
                    solenoidChannel.WriteSingleSampleMultiLine(true, onArray);
                    break;*/
            }
            _isExtended = false;
        }
        #endregion
    }
}

////////////////////////////////////////////////////////////////////
// Shank.cs
//
// Last Modified: 3/14/10
//
////////////////////////////////////////////////////////////////////

using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Text;

namespace Jaywalker
{
    class Shank
    {
        #region "Object Variables"
        //**********************//
        //    CLASS VARIABLES    //
        //**********************//
        private long _leg;
        private double _angle;
        private double _previousAngle;
        Form1 _frm;
        #endregion

        #region "Constructor"
        //**********************//
        //      CONSTRUCTOR      //
        //**********************//
        public Shank(Form1 frm, long leg)
        {
            _leg = leg;
            _angle = 0.0;
            _previousAngle = Constants.MAX_ACCEL + 100.0;
            _frm = frm;
        }
        #endregion

        #region "Accessor Methods"
        //**********************//
        //   ACCESSOR METHODS    //
        //**********************//
```

```
public long Leg
{
    get { return _leg; }
    set { _leg = value; }
}

public void SetAngle(double currentAngle)
{
    /*double average;
    long threshold = Constants.THRESHOLD;

    // Check for first reading
    if (_previousAngle > Constants.MAX_ACCEL)
    {
        _previousAngle = currentAngle;
        average = Constants.MAX_ACCEL + 100.0;
    }
    else
    {
        // check for a change in position beyond the threshold
        if ((_previousAngle - currentAngle) > threshold)
        {
            currentAngle = _previousAngle - threshold;
        }
        else if ((currentAngle - _previousAngle) > threshold)
        {
            currentAngle = _previousAngle + threshold;
        }

        // average the previous angle with the current
        average = (currentAngle + _previousAngle) / 2;

        // store the new previous angle
        _previousAngle = currentAngle;
```

```
            }

            _angle = average;*/

            _angle = currentAngle;
        }
        #endregion

        #region "Sensing Methods"
        //**********************//
        //    SENSING METHODS    //
        //**********************//
    public double GetAngle()
        {
            return _angle;
        }
        #endregion
    }
}

//////////////////////////////////////////////////////////////////
// Ankle.cs
//
// Last Modified: 12/06/09
//
//////////////////////////////////////////////////////////////////

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Diagnostics;

using NationalInstruments.DAQmx;
```

```
namespace Jaywalker
{
    class Ankle
    {
        #region "Object Variables"
        //**********************//
        //    OBJECT VARIABLES   //
        //**********************//
        private long _leg;
        private string _legString;
        private long _velocity;
        private bool _moveToFlag;
        private bool _plantarflexFlag;
        private bool _dorsiflexFlag;
        Task solenoidTask;
        DigitalSingleChannelWriter solenoidChannel;
        #endregion

        #region "Constructor"
        //**********************//
        //      CONSTRUCTOR      //
        //**********************//
        public Ankle(long leg)
        {
            string port = "";

            _moveToFlag = false;
            _plantarflexFlag = false;
            _dorsiflexFlag = false;
            _leg = leg;

            switch (leg)
            {
```

```
        case Constants.X_LEG:
            port = "Dev1/port3/line3, Dev1/port3/line5";
            _legString = "5 ";
            break;
        case Constants.Y_LEG:
            port = "Dev1/port3/line4";
            _legString = "2 ";
            break;
        case Constants.Z_LEG:
            port = "Dev1/port3/line3, Dev1/port3/line5";
            _legString = "5 ";
            break;
    }

    solenoidTask = new Task();
    solenoidTask.DOChannels.CreateChannel(
        port,
        "solenoidChannel",
        ChannelLineGrouping.OneChannelForAllLines);
    solenoidChannel = new DigitalSingleChannelWriter(solenoidTask.Stream);
}
#endregion

#region "Accessor Methods"
//**********************//
//   ACCESSOR METHODS    //
//**********************//
public long Leg
{
    get { return _leg; }
    set { _leg = value; }
}

public long Velocity
```

```
{
   get { return _velocity; }
   set { _velocity = value; }
}
#endregion

#region "Sensing Methods"
//**********************//
//    SENSING METHODS    //
//**********************//
#endregion

#region "Movement Methods"
//**********************//
//   Movement Methods    //
//**********************//
public void MoveTo(long pos, long vel)
{
   bool[] onArray = { Constants.LOW, Constants.LOW };

   // Disable Ankle Air Cylinders
   switch (_leg)
   {
      case Constants.X_LEG:
         solenoidChannel.WriteSingleSampleMultiLine(true, onArray);
         break;
      case Constants.Y_LEG:
         solenoidChannel.WriteSingleSampleSingleLine(true, Constants.LOW);
         break;
   }

   // Generate arguement string
   string args = "m " + _legString + vel.ToString() + " " + pos.ToString();
```

```
    // Start OmsMove and pass the arguement string
    if (_moveToFlag == false)
    {
        Process.Start("C:\\Jaywalker_Data\\OmsMove.exe", args);
        _moveToFlag = true;
    }

    // Reset all flags
    _plantarflexFlag = false;
    _dorsiflexFlag = false;
}

public void Dorsiflex(long vel)
{
    bool[] onArray = { Constants.LOW, Constants.LOW };

    // Generate arguement string
    string args = "d " + _legString + vel.ToString();

    // Start OmsMove and pass the arguement string
    if (_dorsiflexFlag == false)
    {
        Process.Start("C:\\Jaywalker_Data\\OmsMove.exe", args);
        _dorsiflexFlag = true;
    }

    // Disable Ankle Air Cylinders
    switch (_leg)
    {
        case Constants.X_LEG:
            solenoidChannel.WriteSingleSampleMultiLine(true, onArray);
            break;
        case Constants.Y_LEG:
            solenoidChannel.WriteSingleSampleSingleLine(true, Constants.LOW);
```

```
            break;
    }

    // Reset all flags
    _moveToFlag = false;
    _plantarflexFlag = false;
}

public void Plantarflex(long vel)
{
    bool[] onArray = { Constants.HIGH, Constants.HIGH };

    // Generate arguement string
    string args = "p " + _legString + vel.ToString();

    // Start OmsMove and pass the arguement string
    if (_plantarflexFlag == false)
    {
        Process.Start("C:\\Jaywalker_Data\\OmsMove.exe", args);
        _plantarflexFlag = true;
    }

    // Actuate Ankle Air Cylinders
    switch (_leg)
    {
        case Constants.X_LEG:
            solenoidChannel.WriteSingleSampleMultiLine(true, onArray);
            break;
        case Constants.Y_LEG:
            solenoidChannel.WriteSingleSampleSingleLine(true, Constants.HIGH);
            break;
    }

    // Reset all flags
```

```
    _moveToFlag = false;
    _dorsiflexFlag = false;
}

public void StepReset(long vel)
{
    if (_leg == Constants.X_LEG)
    {
        Dorsiflex(vel);

        // Generate arguement string
        //string args = "m 2 " + vel.ToString() + " -1800";

        // Start OmsMove and pass the arguement string
        //Process.Start("C:\\Jaywalker_Data\\OmsMove.exe", args);
    }
}

public void Stop()
{
    bool[] onArray = { Constants.LOW, Constants.LOW };

    // Generate arguement string
    string args = "s " + _legString;

    // Start OmsMove and pass the arguement string
    Process.Start("C:\\Jaywalker_Data\\OmsMove.exe", args);

    // Disable Ankle Air Cylinders
    switch (_leg)
    {
        case Constants.X_LEG:
            solenoidChannel.WriteSingleSampleMultiLine(true, onArray);
            break;
```

```
                case Constants.Y_LEG:
                    solenoidChannel.WriteSingleSampleSingleLine(true, Constants.LOW);
                    break;
            }

            // Reset all flags
            _moveToFlag = false;
            _plantarflexFlag = false;
            _dorsiflexFlag = false;
        }
        #endregion
    }
}

/////////////////////////////////////////////////////////////////////
// Foot.cs
//
// Last Modified: 3/14/10
//
/////////////////////////////////////////////////////////////////////

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using NationalInstruments.DAQmx;

namespace Jaywalker
{
    class Foot
    {
        #region "Object Variables"
        //***********************//
```

```
//    CLASS VARIABLES    //
//**********************//
private long _leg;
private double _previousAngle;
private double _angle;
Form1 _frm;
private string port;
private Task sensorTactileTask;
private DigitalSingleChannelReader sensorTactileChannel;
//bool[] data;
#endregion

#region "Constructor"
//**********************//
//     CONSTRUCTOR      //
//**********************//
public Foot(Form1 frm, long leg)
{
   _leg = leg;
   _angle = 0.0;
   _previousAngle = Constants.MAX_ACCEL + 100.0;
   _frm = frm;

   switch (leg)
   {
      case Constants.X_LEG:
         port = "Dev1/port6/line0:1";
         break;
      case Constants.Y_LEG:
         port = "Dev1/port6/line2:3";
         break;
   }

   sensorTactileTask = new Task();
```

```
    sensorTactileTask.DIChannels.CreateChannel(
        port,
        "sensorTactileChannel",
        ChannelLineGrouping.OneChannelForAllLines);
    sensorTactileChannel = new DigitalSingleChannelReader(sensorTactileTask.Stream);

}
#endregion

#region "Accessor Methods"
//**********************//
//   ACCESSOR METHODS    //
//**********************//
public long Leg
{
    get { return _leg; }
    set { _leg = value; }
}

public void SetAngle(double currentAngle)
{
    /*double average;
    long threshold = Constants.THRESHOLD;

    // Check for first reading
    if (_previousAngle > Constants.MAX_ACCEL)
    {
        _previousAngle = currentAngle;
        average = Constants.MAX_ACCEL + 100.0;
    }
    else
    {
        // check for a change in position beyond the threshold
        if ((_previousAngle - currentAngle) > threshold)
```

```
            {
               currentAngle = _previousAngle - threshold;
            }
            else if ((currentAngle - _previousAngle) > threshold)
            {
               currentAngle = _previousAngle + threshold;
            }

            // average the previous angle with the current
            average = (currentAngle + _previousAngle) / 2;

            // store the new previous angle
            _previousAngle = currentAngle;
         }

         _angle = average;*/

         _angle = currentAngle;
      }
      #endregion

      #region "Sensing Methods"
      //**********************//
      //    SENSING METHODS    //
      //**********************//
   public double GetAngle()
      {
         return _angle;
      }

      public bool GetHeelStrike()
      {
         bool[] data;
         data = sensorTactileChannel.ReadSingleSampleMultiLine();
```

```
            return data[0];
        }

        public bool GetToeOff()
        {
            bool[] data;
            data = sensorTactileChannel.ReadSingleSampleMultiLine();
            return data[1];
        }
        #endregion
    }
}

/////////////////////////////////////////////////////////////////////
// Toe.cs
//
// Last Modified: 3/14/10
//
/////////////////////////////////////////////////////////////////////

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Jaywalker
{
    class Toe
    {
        //**********************//
        //    CLASS VARIABLES    //
        //**********************//
        private long _leg;
    private bool _isExtended;
```

```
    //**********************//
    //      CONSTRUCTOR      //
    //**********************//
    public Toe(long leg)
    {
       _leg = leg;
    }

    //**********************//
    //   ACCESSOR METHODS    //
    //**********************//
    ////////////////////////////////////////
    public long Leg
    {
       get { return _leg; }
       set { _leg = value; }
    }

public bool IsExtended
    {
       get { return _isExtended; }
       // no setter method
    }

    //**********************//
    //    SENSING METHODS    //
    //**********************//
    ////////////////////////////////////////

    #region "Movement Methods"
    //**********************//
    //   Movement Methods    //
    //**********************//
```

```
        public void FlexToe()
        {
            //TODO: ADD CODE HERE

            _isExtended = true;
        }

        public void ExtendToe()
        {
            //TODO: ADD CODE HERE

            _isExtended = false;
        }
        #endregion
    }
}

////////////////////////////////////////////////////////////////////////
// MPU.cs
//
// Last Modified: 10/3/09
//
////////////////////////////////////////////////////////////////////////

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Jaywalker
{
    class MPU
    {
        //**********************//
```

```
//    CLASS VARIABLES    //
//**********************//
private Robot _jaywalker;
private Form1 _frm;
private Accelerometer _accel;
private Position[] _accelData;
private bool _readFlag;

//**********************//
//      CONSTRUCTOR      //
//**********************//
public MPU(Robot robot, Form1 frm)
{
   _jaywalker = robot;
   _frm = frm;
   _accel = new Accelerometer(_frm);
   _accelData = new Position[10];
   _readFlag = false;
}

//**********************//
//     READ METHODS      //
//**********************//
public void ReadData()
{
   //int writeCycle = 0;
   //long start, runTime;
   Position blankPos = new Position();

   blankPos.SetAngle(100, 100, 100);
   _readFlag = true;

   do
   {
```

```
            //start = DateTime.Now.Ticks;
            // Read in data from accelerometers
            for (int i = 0; i < 10; i++)
            {
                lock (_accelData)
                {
                    if (i == Constants.X_THIGH_ACCEL || i == Constants.Y_THIGH_ACCEL)
                    {
                        _accelData[i] = _accel.Read(i);
                    }
                    else
                    {
                        _accelData[i] = blankPos;
                    }
                }
            }
            //runTime = DateTime.Now.Ticks - start;
            //Console.WriteLine(runTime.ToString() + " Ticks");


            lock (_accelData)
            {
                // Write accelerometer data to form
                _frm.AccelDataUpdate(_accelData);
                // Send accelerometer data to robot parts
                _jaywalker.DitributeData(_accelData);
            }
        } while (_readFlag);
    }

    public void Stop()
    {
        _readFlag = false;
        _accel.Stop();
```

```
        }
    }
}

////////////////////////////////////////////////////////////////////
// Accelerometer.cs
//
// Last Modified: 11/22/09
//
////////////////////////////////////////////////////////////////////

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

using NationalInstruments.DAQmx;

namespace Jaywalker
{
    class Accelerometer
    {
        #region "Object Variables"
        //***********************//
        //    OBJECT VARIABLES   //
        //***********************//
        Task readTask;
        DigitalSingleChannelReader readChannel;
        Task selectorTask;
        DigitalSingleChannelWriter selectorChannel;
        Task sxTalkTask;
        DigitalSingleChannelReader sxTalkChannel;
        Task usbCommTask;
```

```
DigitalSingleChannelWriter usbCommChannel;

Form1 _frm;
#endregion

#region "Constructor"
//**********************//
//     CONSTRUCTOR      //
//**********************//
public Accelerometer(Form1 frm)
{
   readTask = new Task();
   readTask.DIChannels.CreateChannel(
      "Dev1/port0/line0:7, Dev1/port1/line0:3",
      "readChannel",
      ChannelLineGrouping.OneChannelForAllLines);
   readChannel = new DigitalSingleChannelReader(readTask.Stream);

   selectorTask = new Task();
   selectorTask.DOChannels.CreateChannel(
      "Dev1/port4/line0:7, Dev1/port5/line0:1",
      "selectorChannel",
      ChannelLineGrouping.OneChannelForAllLines);
   selectorChannel = new DigitalSingleChannelWriter(selectorTask.Stream);

   sxTalkTask = new Task();
   sxTalkTask.DIChannels.CreateChannel(
      "Dev1/port5/line2",
      "sxTalkChannel",
      ChannelLineGrouping.OneChannelForAllLines);
   sxTalkChannel = new DigitalSingleChannelReader(sxTalkTask.Stream);

   usbCommTask = new Task();
   usbCommTask.DOChannels.CreateChannel(
```

```
        "Dev1/port5/line3:5",
        "usbCommChannel",
        ChannelLineGrouping.OneChannelForAllLines);
    usbCommChannel = new DigitalSingleChannelWriter(usbCommTask.Stream);

    _frm = frm;
}
#endregion

#region "Read Methods"
//***********************//
//     READ METHODS      //
//***********************//
public Position Read(int accelSelector)
{
    // Declare Variables
    bool[] selector = { true, true, true, true, true, true, true, true, true, true };
    Position pos = new Position();

    //Choose the accelometer to read
    selector[accelSelector] = false;
    selectorChannel.WriteSingleSampleMultiLine(true, selector);

    //Request data from the MPU
    SendUsbCommunications(Constants.REQUEST_DATA);
    WaitForDataRequestComplete();

    //Request Z-Data from MPU
    SendUsbCommunications(Constants.REQUEST_Z_AXIS);
    WaitForSxTalkToBe(Constants.HIGH);
    pos.ZAngle = ReadData();

    //Send Complete Signal
    SendUsbCommunications(Constants.REQUEST_COMPLETE);
```

```
    //Clear accelerometer selector
    selector[accelSelector] = true;
    selectorChannel.WriteSingleSampleMultiLine(true, selector);

    return pos;
}

public void Stop()
{
    SendUsbCommunications(Constants.REQUEST_COMPLETE);
    Thread.Sleep(10);
}
#endregion

#region "Private Methods"
//**********************//
//    PRIVATE METHODS    //
//**********************//
private int ReadData()
{
    // Declare variables
    int data;

    data = readChannel.ReadSingleSamplePortInt32();

    // Adjust for 2's Compliment
    if (data < 2048)
    {
        return data;
    }
    else
    {
        return (data - 4096);
```

```
    }
}

private void WaitForSxTalkToBe(bool logic)
{
    while (sxTalkChannel.ReadSingleSampleSingleLine() != logic)
    {
        // Do Nothing
    }
}

private void WaitForDataRequestComplete()
{
    int data;

    while ((data = ReadData()) != -1366)
    {
        // Do nothing
    }
}

private void SendUsbCommunications(int command)
{
    bool[] comm = { false, false, false };

    _frm.StatusUpdate("Send USB: " + command.ToString());

    switch (command)
    {
        case Constants.REQUEST_DATA:
            //Set to %100
            comm[0] = true;
            comm[1] = false;
            comm[2] = false;
```

```
        break;
    case Constants.REQUEST_X_AXIS:
        //Set to %101
        comm[0] = true;
        comm[1] = false;
        comm[2] = true;
        break;
    case Constants.REQUEST_Y_AXIS:
        //Set to %110
        comm[0] = true;
        comm[1] = true;
        comm[2] = false;
        break;
    case Constants.REQUEST_Z_AXIS:
        //Set to %111
        comm[0] = true;
        comm[1] = true;
        comm[2] = true;
        break;
    case Constants.REQUEST_COMPLETE:
        //Set to %011
        comm[0] = false;
        comm[1] = true;
        comm[2] = true;
        break;
    default:
        //Set to %011
        comm[0] = false;
        comm[1] = true;
        comm[2] = true;
        break;
}

try
```

```
        {
            usbCommChannel.WriteSingleSampleMultiLine(true, comm);
        }
        catch(System.Exception msg)
        {
            // DO NOTHING
        }
    }
    #endregion
  }
}

/////////////////////////////////////////////////////////////////////
// Constants.cs
//
// Last Modified: 8/22/09
//
/////////////////////////////////////////////////////////////////////

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Jaywalker
{
  /// <summary>
  /// Constants holds all constants for Jaywalker.
  /// </summary>
  static class Constants
  {
    // Logic Constants
    public const bool HIGH = true;
    public const bool LOW = false;
```

```
        // Communication Constants
        public const int REQUEST_DATA = 0;
        public const int REQUEST_X_AXIS = 1;
        public const int REQUEST_Y_AXIS = 2;
        public const int REQUEST_Z_AXIS = 3;
        public const int REQUEST_COMPLETE = 4;

        // Axis Constants
        public const int X_LEG = 1;
        public const int Y_LEG = 2;
        public const int Z_LEG = 4;

        // Accelerometer Constants
        public const int X_THIGH_ACCEL = 0;
        public const int X_SHANK_ACCEL = 1;
        public const int X_FOOT_ACCEL = 2;
        public const int Y_THIGH_ACCEL = 3;
        public const int Y_SHANK_ACCEL = 4;
        public const int Y_FOOT_ACCEL = 5;
        public const int Z_THIGH_ACCEL = 6;
        public const int Z_SHANK_ACCEL = 7;
        public const int Z_FOOT_ACCEL = 8;
        public const int HIP_ACCEL = 9;
        public const long THRESHOLD = 100;
        public const double MAX_ACCEL = 4096;
    }
}

////////////////////////////////////////////////////////////////////////
// Position.cs
//
// Last Modified: 8/29/09
//
```

```
//////////////////////////////////////////////////////////////////////

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Jaywalker
{
    /// <summary>
    /// Position encapsulates a cartesian spatial point of an object and its orientation.
    /// </summary>
    public class Position
    {
        //***********************//
        //    CLASS VARIABLES    //
        //***********************//
        private double _x;
        private double _y;
        private double _z;
        private double _xAngle;
        private double _yAngle;
        private double _zAngle;

        //***********************//
        //      CONSTRUCTOR      //
        //***********************//
        // Using the default constructor

        //***********************//
        //   ACCESSOR METHODS    //
        //***********************//
        public double X
        {
```

```
    get { return _x; }
    set { _x = value; }
}

public double Y
{
    get { return _y; }
    set { _y = value; }
}

public double Z
{
    get { return _z; }
    set { _z = value; }
}

public double XAngle
{
    get { return _xAngle; }
    set { _xAngle = value; }
}

public double YAngle
{
    get { return _yAngle; }
    set { _yAngle = value; }
}

public double ZAngle
{
    get { return _zAngle; }
    set { _zAngle = value; }
}
```

```
        //***********************//
        //     INPUT METHODS     //
        //***********************//
        /// <summary>
        /// Position.SetCoordinate sets the coordinates of the object.
        /// </summary>
        /// <param name="x">The x coordinate of the object.</param>
        /// <param name="y">The y coordinate of the object.</param>
        /// <param name="z">The z coordinate of the object.</param>
        public void SetCoordinate(double x, double y, double z)
        {
            _x = x;
            _y = y;
            _z = z;
        }

        /// <summary>
        /// Position.SetAngle sets the angles of the object.
        /// </summary>
        /// <param name="xAngle">The x angle of the object.</param>
        /// <param name="yAngle">The y angle of the object.</param>
        /// <param name="zAngle">The z angle of the object.</param>
        public void SetAngle(double xAngle, double yAngle, double zAngle)
        {
            _xAngle = xAngle;
            _yAngle = yAngle;
            _zAngle = zAngle;
        }
    }
}
```

A.2 C++ INTERFACING CODE

```
//////////////////////////////////////////////////////////////////////
```

```
// OmsMove.cpp
//
// Version: 0.5
//
// Date: 11/22/2009
//
// Description:
// Console program for interfacing with the OmsMaxpMC card
//
//
// Warning:
// The foot must be set to dorsiflex before the computer is started
//
// Command Line Arguments:
// Command Axis Velocity Position State
//
// Command: (required for all)
// Plantarflex Foot - p
// Dorsiflex Foot - d
// Move Foot To - m
// Stop Foot - s
// Reset Card - r
// Hip - h
// Stop Hip - k
//
// Axis: (required for p, d, m, s, and l)
// X Leg - 1
// Inside Leg - 2
// Z Leg - 4
// Outside Legs - 5
//
// Velocity: (required for p, d, m, and h)
// #
//
```

```
// Position: (required for m, and h)
// #
//
// Examples:
//     p 2 1500
//     d 5 1500
//     m 2 1500 1000
//     s 5
//     r
//     h 1500 1000
//     k
//
////////////////////////////////////////////////////////////////////////

//Included Files
#include <iostream>

#include "OmsMAXpMC.h"

// Set Namespace
using namespace std;

// Declare Functions
long getMtrPosValue(long legAxis);
long getAxis(char axis);

// Declare Constants
// DEFINE SOFT LIMITS
const long MAX_PLANT_FLEX = -3500;
const long MAX_DORSI_FLEX = 0;

//////////////////////////////////////
// Main Function
int main (int argc, const char *argv[])
```

```
{
    // Declare Variables
    long axis;
    long velocity;
    long position;
    AXES_DATA pos;

    // Open Handle to controller card
    const HANDLE omsHandle = GetOmsHandle("OmsMaxp1");

    // Check for minimum arguements
    if (argc < 2)
    {
            cout << "Not enough parameters." << endl;
            return -1;
    }

    // Read in the command
    switch (argv[1][0])
    {
    // Move the Foot to a position
    case 'm':
            cout << "Move Foot To Position command received." << endl;
            // Check for proper number of arguments
            if (argc < 5)
            {
                    cout << "Not enough parameters." << endl;
                    return -1;
            }
            else
            {
                    // Read in the axis
                    axis = getAxis(argv[1][2]);
                    // Read in velocity and position
```

```
            velocity = strtol (argv[3], NULL, 10);
            position = strtol (argv[4], NULL, 10);
            cout << "Velocity: " << velocity << endl;
            cout << "Position: " << position << endl;
        }

        // Move the foot
        switch(axis)
        {
            case OMS_X_AXIS + OMS_Z_AXIS:
                //Set Axii X and Z Velocities
                SetOmsAxisVelocity(omsHandle, OMS_X_AXIS, velocity);
                SetOmsAxisVelocity(omsHandle, OMS_Z_AXIS, velocity);

                //Set Position for Axii X and Z
                pos.X = position;
                pos.Z = position;
                break;

            case OMS_Y_AXIS:
                //Set Axes Y Velocity
                SetOmsAxisVelocity(omsHandle, OMS_Y_AXIS, velocity);

                //Set Position for Axes Y
                pos.Y = position;
                break;
        }

        // Move foot to the limit
        MoveOmsLinearAbsMt(omsHandle, axis, &pos);
        break;
    // Move the foot to dorsiflex
    case 'd':
        cout << "Move Foot To Dorsiflex command received." << endl;
```

```
// Check for proper number of arguments
if (argc < 4)
{
        cout << "Not enough parameters." << endl;
        return -1;
}
else
{
        // Read in the axis
        axis = getAxis(argv[1][2]);
        // Read in velocity
        velocity = strtol (argv[3], NULL, 10);
        cout << "Velocity: " << velocity << endl;
}

// Move the foot
switch(axis)
{
        case OMS_X_AXIS + OMS_Z_AXIS:
                //Set Axii X and Z Velocities
                SetOmsAxisVelocity(omsHandle, OMS_X_AXIS, velocity);
                SetOmsAxisVelocity(omsHandle, OMS_Z_AXIS, velocity);

                //Set Position for Axii X and Z
                pos.X = MAX_DORSI_FLEX;
                pos.Z = MAX_DORSI_FLEX;
                break;

        case OMS_Y_AXIS:
                //Set Axes Y Velocity
                SetOmsAxisVelocity(omsHandle, OMS_Y_AXIS, velocity);

                //Set Position for Axes Y
                pos.Y = MAX_DORSI_FLEX;
```

```
                break;
        }

        // Move foot to the limit
        MoveOmsLinearAbsMt(omsHandle, axis, &pos);
        break;
case 'p':
        cout << "Move Foot To Plantarflex command received." << endl;
        // Check for proper number of arguments
        if (argc < 4)
        {
                cout << "Not enough parameters." << endl;
                return -1;
        }
        else
        {
                // Read in the axis
                axis = getAxis(argv[1][2]);
                // Read in velocity
                velocity = strtol (argv[3], NULL, 10);
                cout << "Velocity: " << velocity << endl;
        }

        // Move the foot
        switch(axis)
        {
                case OMS_X_AXIS + OMS_Z_AXIS:
                        //Set Axii X and Z Velocities
                        SetOmsAxisVelocity(omsHandle, OMS_X_AXIS, velocity);
                        SetOmsAxisVelocity(omsHandle, OMS_Z_AXIS, velocity);

                        //Set Position for Axii X and Z
                        pos.X = MAX_PLANT_FLEX;
                        pos.Z = MAX_PLANT_FLEX;
```

```
                break;

            case OMS_Y_AXIS:
                //Set Axes Y Velocity
                SetOmsAxisVelocity(omsHandle, OMS_Y_AXIS, velocity);

                //Set Position for Axes Y
                pos.Y = MAX_PLANT_FLEX;
                break;
        }

        // Move foot to the limit
        MoveOmsLinearAbsMt(omsHandle, axis, &pos);
        break;
    case 's':
        cout << "Stop Foot command received." << endl;
        // Check for proper number of arguments
        if (argc < 3)
        {
            cout << "Not enough parameters." << endl;
            return -1;
        }
        else
        {
            // Read in the axis
            axis = getAxis(argv[1][2]);

            // Stop the foot
            switch(axis)
            {
                case OMS_X_AXIS + OMS_Z_AXIS:
                    // Stop Axii
                    StopOmsAxis(omsHandle, OMS_X_AXIS);
                    StopOmsAxis(omsHandle, OMS_Z_AXIS);
```

```
                    break;

                case OMS_Y_AXIS:
                    StopOmsAxis(omsHandle, OMS_Y_AXIS);
                    break;
            }
        }
        break;
    case 'r':
        cout << "Reset controller command received." << endl;

        // Reset the controller
        ResetOmsController(omsHandle);
        break;
    case 'h':
        cout << "Move Hip command received." << endl;
        // Check for proper number of arguments
        if (argc < 4)
        {
            cout << "Not enough parameters." << endl;
            return -1;
        }
        else
        {
            // Read in velocity and position
            velocity = strtol (argv[2], NULL, 10);
            position = strtol (argv[3], NULL, 10);
            cout << "Velocity: " << velocity << endl;
            cout << "Position: " << position << endl;
        }

        // Move the hip
        //Set Axes T Velocity
        SetOmsAxisVelocity(omsHandle, OMS_T_AXIS, velocity);
```

```
        //Set Position for Axes T
        pos.T = position;

        // Move hip to the limit
        MoveOmsLinearAbsMt(omsHandle, OMS_T_AXIS, &pos);
        break;
    case 'k':
        cout << "Stop Hip command received." << endl;

        // Stop the hip
        StopOmsAxis(omsHandle, OMS_T_AXIS);
        break;
    default:
        cout << "Not a valid command." << endl;
        return -3;
    }

    // Close Handle
    CloseOmsHandle(omsHandle);

    return 0;
}

//////////////////////////////////////
// Get Axis
long getAxis(char axis)
{
    switch (axis)
    {
    case '1':
        cout << "Axis: " << axis << endl;
        return 1;
        break;
```

```
   case '2':
         cout << "Axis: " << axis << endl;
         return 2;
         break;
   case '4':
         cout << "Axis: " << axis << endl;
         return 4;
         break;
   case '5':
         cout << "Axis: " << axis << endl;
         return 5;
         break;
   default:
         cout << "Not a valid axis." << endl;
         return -2;
   }
}

/////////////////////////////////////
// Get Motor Position
long getMtrPosValue(long legAxis)
{
   // Declare Constant Variables
   const HANDLE omsHandle = GetOmsHandle("OmsMaxp1");

   // Declare variables
   long motorPosition;
   long status;

   // Read motor position
   status = GetOmsAxisMotorPosition(omsHandle, legAxis, &motorPosition);

   // Close Handle
   CloseOmsHandle(omsHandle);
```

```
  return motorPosition;
}
```