Work Space Analysis and Walking Algorithm Development for A Radially Symmetric Hexapod Robot

Mark H. Showalter

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Mechanical Engineering

Dennis H. Hong, Chair
Robert H. Sturges
Alfred L. Wicks

August 1, 2008
Blacksburg, Virginia

Keywords: Workspace, Walking Algorithm, Gait Planning, Hexapod,
Robot Locomotion, Kinematics

Work Space Analysis and Walking Algorithm Development for A Radially Symmetric Hexapod Robot

Mark H. Showalter

(ABSTRACT)

The Multi-Appendage Robotic System (MARS) built for this research is a hexapod robotic platform capable of walking and performing manipulation tasks. Each of the six limbs of MARS incorporates a three-degree of freedom (DOF), kinematically spherical proximal joint, similar to a shoulder or hip joint; and a 1-DOF distal joint, similar to an elbow or knee joint. Designing walking gaits for such multi-limb robots requires a thorough understanding of the kinematics of the limbs, including their workspace. The specific abilities of a walking algorithm dictate the usable workspace for the limbs. Generally speaking, the more general the walking algorithm is, the less constricted the workspace becomes. However, the entire limb workspace cannot be used in a continuous, statically stable, alternating tripedal gait for such a robot; therefore a subset of the limb workspace is defined for walking algorithms. This thesis develops MARS limb workspaces in the knee up configuration, and analyzes its limitations for walking on planar surfaces. The workspaces range from simple 2D geometry to complex 3D volumes.

While MARS is a hexapedal robot, the tasks of defining the workspace and walking agorthm for all six limbs can be abstracted to a single limb using the constraint of a tripedal, statically stable gait. Based on understanding the behavior of an individual limb, a walking algorithm was developed to allow MARS to walk on level terrain. The algorithm is adaptive in that it continously updates based on control inputs. Open Tech developed a similar algorithm, based on a 2D workspace. This simpler algorithm developed resulted in smooth gait generation, with near-instantaneous response to control input. This accomplishment demonstrated the feasibility of implementing a more sophisticated algorithm, allowing for inputs of all six DOF: x and y velocity, z velocity or walking height, yaw, pitch and roll. This latter algorithm uses a 3D workspace developed to afford near-maximum step length. The workspace analysis and walking algorithm development in this thesis can be applied to the further advancement of walking gait generation algorithms.

Support Information

This work received support from Joint Unmanned Systems Test, Experimentation, and Research (JOUSTER). Thanks is also due to NASA JPL for their generous support of this project.

Dedication

This thesis is dedicated to all those who dread academia.

Acknowledgments

I thank Jesus Christ for getting me into graduate school and laughing me through it. I extend my sincere gratitude to Dr. Hong who has exemplified the accomplished engineer, professor, mentor, and teacher. Dr. Hong's dedication to his students is obvious and admirable. I appreciate Dr. Sturges' and Dr. Wicks' commitment to serve on my committee. I am confident in your assessment of my work. I would like to thank Karl, Derek, and Robert for all your patient help with MARS. To the rest of my lab mates, thank you for helping me to think and laugh; I greatly appreciate all of you. I would like to thank Dan for all your programing help. And I would like the thank my family, friends, and church for your solid support in this undertaking.

Contents

1 Introduction **1**
1.1 Motivation . . . 3
1.2 A Survey of Non-Biped Walking Machines . . . 3
1.3 Hexapod Walking Algorithms . . . 5
1.4 Discussion . . . 7

2 Robot Design **9**
2.1 Limb Design . . . 9
2.2 Coordinate Frame Definitions . . . 10
2.3 Proximal Joint . . . 11
2.4 Distal Joint . . . 11
2.5 Proximal Limb Section . . . 14
2.6 Distal Limb Section . . . 15
2.7 Body . . . 15

3 Kinematics and Singularity Analysis **18**
3.1 Forward Kinematics . . . 18
3.2 Inverse Kinematics . . . 22
3.3 Singularity Analysis . . . 23

4 Workspace Analysis **26**
4.1 The General Workspace . . . 26
4.2 The General Knee Up Workspace . . . 27
4.3 Similarities to Insect Limbs . . . 31
4.4 Geometric Delineation of the General Knee Up Workspace . . 32
4.5 General Knee Up Workspace Limitations . . . 36
4.6 The Buffer Cylinder . . . 38
4.7 MARS Specific Workspace Limitations . . . 42

4.8 The 2D Common Workspace 42
4.9 3D Common Workspaces 49

5 Walking Algorithms 52
5.1 The Abstracted Walking Algorithm 53
5.2 The General Walking Algorithm 53
5.2.1 Defining the Next Limb Tip Location 54
5.2.2 Buffer Cylinder Tangency 58
5.2.3 Contact/Non-Contact Limb Switch 61
5.2.4 Stride-Line Workspace Intersection 62
5.3 Attempted Implementation of the General Walking Algorithm 68
5.4 Implemented Walking Algorithm 73

6 Future Work 74
6.1 New Approach to Replace Inverse Kinematics 74
6.2 Workspace Analysis Using Spherical Coordinates 75
6.3 Kinematics Redesign 75
6.4 Alternative Walking Algorithms 75
6.5 Walking Algorithm Additions 76

A Programming Developed for the General Walking Algorithm 77
A.1 Mathematica Gait Simulation Script 77
A.2 LabVIEW VI 84
A.3 Function Call 89
A.4 Inverse Kinematics 91
A.5 MARSlimbCONTROL 92
A.6 MARStotalTIPtranslation 94
A.7 MARSlimbFRAMEdirection 96
A.8 MARSfindSTRIDEline 97
A.9 MARSdirectionSTRIDEline 99
A.10 MARSdirectionCHECK 101
A.11 MARSnonContactSTRIDEline 103
A.12 MARSnoncontactSTEPvector 104
A.13 MARSnextTIPpos 106
A.14 MARSlimbSWITCHtest 106
A.15 MARStipPOSoutput 108
A.16 MARSContactLine 110
A.17 MARSNonContactLine 115

A.18 MARSCylInt 119
A.19 MARSTanPoint 120
A.20 MARSTanPointAlt 121
A.21 MARSShell1Int 121
A.22 MARSShell2Int 122
A.23 MARSShell3Int 124
A.24 MARSShell4Int 125
A.25 MARSPlane1Int 126
A.26 MARSPlane2Int 127
A.27 MARSRoots 129
A.28 MARSCircLinInt 130
A.29 MARSPlaneInt 130

B Mathematica script used to find the polynomial terms for stride-line torus intersection roots 132

C Walking Algorithm Code Developed by Open Tech 134

List of Figures

1.1 MARS, developed at RoMeLa. Courtesy Virginia Tech. Photograph taken by Josh Armstrong. 2
1.2 LEMUR IIb (Legged Excursion Mechanical Utility Rover) from NASA JPL. Courtesy NASA\JPL-CalTech 2

2.1 The y-axis of the body coordinate frame points toward the center of the proximal joint of Limb 1. 10
2.2 Coordinate frames are assigned to each revolute joint on a limb. 11
2.3 In the proximal joint, Revolute Joints 1, 2, and 3 intersect at a single point. 12
2.4 The proximal joint design utilizes the entire range of the Revolute Joint 1 actuator while providing a well supported framework for all three revolute joints. Courtesy Virginia Tech. Photograph taken by Josh Armstrong. 13
2.5 The distal joint also provides support on both sides of the actuator, similar to the revolute joints two and three of the proximal joint. Courtesy Virginia Tech. Photograph taken by Josh Armstrong. 13
2.6 The proximal limb section, constructed primarily of single layer carbon fiber and composite/core sandwich end nodes, provides high strength and rigidity with low weight. 14
2.7 A clam-shell mold was used to cast the proximal limb sections. Granular salt pressed inside a carbon fiber sleeve forces the sleeve to conform to the inside of the mold. 15
2.8 A friction fit secures the polystyrene and carbon fiber distal limb section to an aluminum bracket and tube assembly attached to the distal joint actuator. 16
2.9 Two thin carbon-fiber plates, separated by limb supports, forms the body of the robot. 17

3.1 The inverse kinematics can be derived geometrically for the MARS limb with Revolute Joint 1 frozen. 24

4.1 In 2-D Revolute Joints 3 and 4 can sweep out this crescent shape. 28
4.2 Revolute Joints 2, 3 and 4 can sweep out this 3D volume–the general workspace. 29
4.3 Of the two possible configurations, the -90 to 0 degrees knee-up configuration is used for walking. 30
4.4 Knee-down limb configuration requires the limb to extend through the floor for certain walking heights. 31
4.5 An insect limb uses a similar joint configuration to the MARS limb. 32
4.6 In the knee-up configuration the 2D workspace in the z2-y2 plane is the area contained within four curves. 33
4.7 Six shells form the initial 3D knee-up workspace. 36
4.8 Planar sections which form part of the boundary for the workspace are encompassed by arc sections of circles. 37
4.9 Movement of the limb tip from the blue region of the workspace to the yellow region requires an instantaneous 180 degree rotation of Revolute Joint 2. 39
4.10 The workspace is limited due to constraints on continuous movement of the limb tip. 40
4.11 The workspace is limited due to constraints on continuous movement of the limb tip. 41
4.12 The required rotational velocity of Revolute Joint 2 goes to infinity as the limb tip path approaches the z_2 axis. 43
4.13 The buffer cylinder, about the z_2-axis, further limits the workspace. 44
4.14 All three contact limb stride-lines are limited by the shortest stride-line. 45
4.15 Overlaying the workspaces of the three contact limbs reveals the common workspace in the style of a Venn diagram. The circle is a simplification of the common workspace. 46
4.16 A 3D volume contains the set of 2D circular workspaces. . . . 48
4.17 A spherical workspace simplifies the walking algorithm while allowing robot body roll and pitch. 51

5.1 Iteratively generated stride-lines can form curved step paths. . 55

5.2 One iteration of the basic walking algorithm. 55
5.3 The Ideal stride-lines make a long path through the workspace, avoiding the buffer cylinder. 57
5.4 The next non-contact limb tip location is selected so as to optimize the gait for the current operator input. 59
5.5 A point tangent to the buffer cylinder, p, is used to help define non-contact stride-lines and pre-stride-lines; however, if this point is outside the workspace, the orthogonal intersection point between the height-line and stride-line is used. . . . 60
5.6 The intersection points of a stride-line with spherical Shells 3a, 4a, and 4b are found using this geometric approach. 64
5.7 The basic architecture of the general walking algorithm consists of a LabVIEW VI which calls a set of MATLAB m-files. . 70

A.1 A set of hierarchically-arranged MATLAB functions were developed to demonstrate the feasibility of using a computer to find the stride-line through the general workspace. 78

Unless otherwise stated, all images and figures are property of the author.

List of Tables

3.1 Denavit-Hartenburg Parameters 18

4.1 MARS Limb Range of Operation. 27

Chapter 1

Introduction

This thesis presents the design, construction, workspace analysis, and walking algorithm development for a mobile robot, MARS (Multi-Appendage Robotic System) shown in Figure 1.1. MARS is a hexapod mobile robotic research platform patterned after the LEMUR IIb (Legged Excursion Mechanical Utility Rover) [1–4]. The LEMUR IIb, shown in Figure 1.2, is the latest in a series of hexapedal robots developed at JPL for autonomous inspection and maintenance tasks on the exterior of space structures and vehicles in near-zero gravity. The robot performs maintenance tasks by exchanging, via a quick connect, a foot for a tool. After positioning with the remaining limbs, the robot would then perform necessary repairs. These scenarios evoke many research possibilities including: wrench space analysis, robot and work object coordination, hull navigation, and walking algorithms. The main focus of this thesis is to develop limb workspaces and basic walking algorithms applicable to robots kinematically similar to LEMUR IIb.

The design of LEMUR IIb and MARS differ from biologicaly-inspired hexapedal robots [5–9] in symmetry. Quinn, Espenchied, et al., have developed highly mobile hexapedal robots patterned after the stick insect and cockroach. These robots employ two rows of three bilaterally symmetric limbs. However, by employing radial symmetry the LEMUR IIb and MARS platforms do not possess a set front and back, and are therefore capable of walking in any direction without turning.

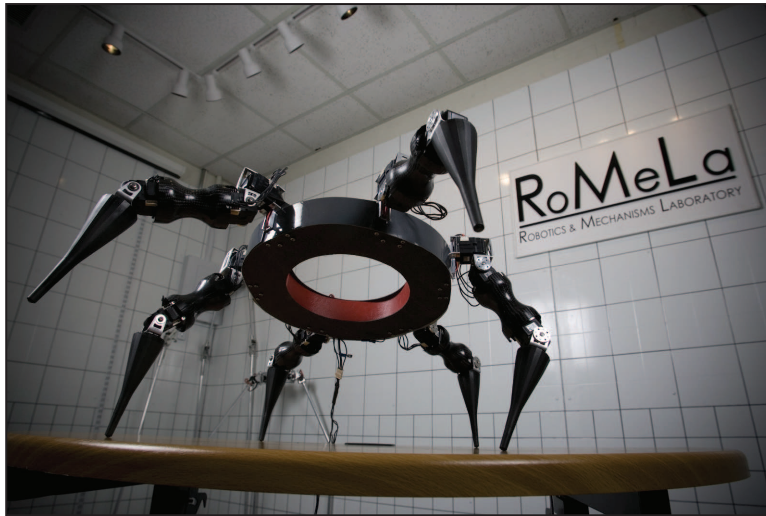

Figure 1.1: MARS, developed at RoMeLa. Courtesy Virginia Tech. Photograph taken by Josh Armstrong.

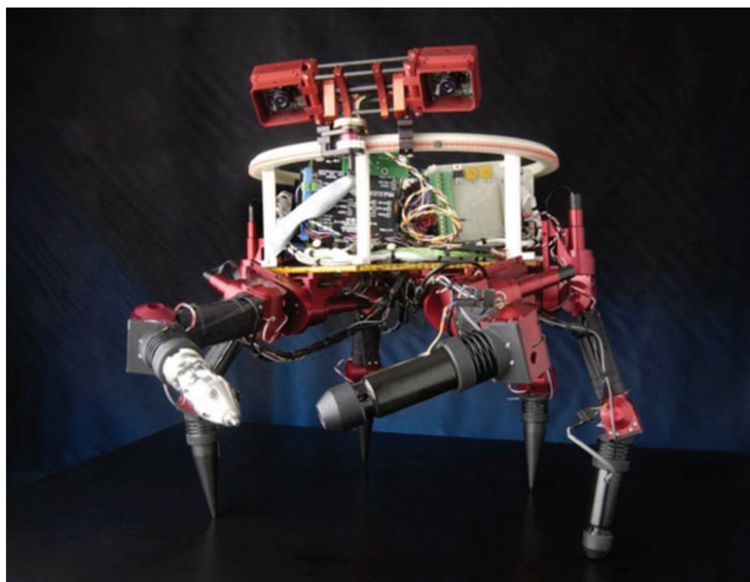

Figure 1.2: LEMUR IIb (Legged Excursion Mechanical Utility Rover) from NASA JPL. Courtesy NASA\JPL-CalTech

1.1 Motivation

This thesis documents the initial work on radially-symmetric hexapods at RoMeLa (Robotics and Mechanisms Laboratory at Virginia Tech, which began in collaboration with NASA(National Aeronautics and Space Administration) JPL(Jet Propulsion Laboratory). The goals of RoMeLa's research in this area are to develop robots of this type, to understand the capabilities and limitations of the robot, and to develop autonomous and semi-autonomous control strategies for gait generation and limb manipulation. These goals are based on the intended use of LEMUR robots for repair and maintenance in near-zero gravity on the surface of spacecraft. Building MARS, and defining a few of its workspaces and walking algorithms, laid a foundation for further research. The construction of the MARS robot lead to the use of innovative techniques in forming composite members and also lead to important considerations for redesign. The development of a walking algorithm lead to the discovery of key relationships between the walking algorithm and limb workspace. An understanding of these relationships will be key in the future work of expanding the walking algorithm for 3D terrain as well as limb manipulation for tool use.

In a broader sense this research serves as an addition to man's work in reproducing the limbed conveyance methods found in nature. It has been shown that on unstructured terrain, legged vehicles are superior to wheeled vehicles and tracked vehicles for speed, economy, and mobility [10]. For these reasons humans have made numerous attempts to replicate this natural mode of transportation. This thesis adds to previous and current work by developing one type of algorithm for hexapod gaits which are omnidirectional and statically stable.

1.2 A Survey of Non-Biped Walking Machines

The development of limbed machines for the purpose of walking is by no means a new endeavor. According to legend, the ancient Chinese inventor and master carpenter, Lu Ban (circa 507-444 BC), built a "wooden horse carriage" capable of automatic movement [11]. It is recorded that Lu Ban built the device, which required no manual intervention, for his aged mother. When she rode the device it sped away, and did not return. While Lu Ban was credited with many inventions, most notably a wooden bird which could

stay in the sky for three days, documentation of his achievements is limited to literary records. There are no surviving plans or hardware. Like Lu Ban, the ancient Chinese military leader Zhu-ge Liang is credited with the invention of a walking machine during the Era of the Three Kingdoms (AD 220-280). Liang's device, named "wooden ox and gliding horse," was evidently developed to transport heavy military cargo over rough terrain. Wooden ox and gliding horse is recorded to not have been powered by wind or water and did not require man's physical effort.

While records of ancient walking machines have survived from antiquity, documented designs for limbed devices have been much more abundant within the last several decades. In 1893 Rygg attained a patent for a design of a limbed mechanical horse [12]. In the early twentieth century several patents were attained for walking vehicles for moving load and crossing difficult terrain [13–15]. Interestingly, in 1945 Wallace patented a one legged hopping tank [16]. This concept was latter proven on a smaller scale in 1983 by Raibert et al. [17].

In the second half of the twentieth century walking machines became more reality than novelty. In 1961 the Space General Corporation developed two multi-legged walking vehicles, one with six legs and one with eight [10]. These vehicles were intended for lunar exploration. In 1968 the General Electric General Engineering Laboratory developed a quadruped walking truck [18]. The 1400 kg truck was controlled by the arm and leg movements of a human rider and actuated using hydraulics. At roughly the same time development of a load carrying device for the US Army and NASA began [18]. This Iron Mule Train was envisioned to be a train of legged vehicles which would trail a human guide. Versions with six and eight cam-controlled legs were developed which used alternating tripod or tetrapod gaits, respectively.

At this point limbed vehicle development shifted to using primarily computer control. The "Phoney Pony," developed by Frank and McGhee of the University of Southern California used computer control for its four electric motor-activated legs [10]. In 1972 Petternella and associates at the University of Rome built a computer-controlled robot similar to the "Phoney Pony," but with six legs [10]. In 1977 Ohio State University (OSU) demonstrated a computer-controlled hexapod with axial symetry [18]. The limb kinematics of this machine are similar to MARS in the 2-DOF hip joint and 1-DOF knee joint; however, the axes of the shoulder joints were slightly offset and did not intersect.

The 1980's saw the development of several hexapod robots. Most used ax-

ial symmetry; however, Odetics Inc. developed a radially symmetric hexapod which used an onboard computer to play back pre-programmed motions [19]. Using remote human control of the pre-recoded motions, the hexapod could climb obstacles such as stairs and a pickup truck. In 1983 Raibert and Southerland developed a hexapod walking machine capable of navigating rough terrain using different types of gaits [10]. This axi-symmetric machine used a combination of hydraulic feedback, computer control, and human control. In the late 1980's Quinn, et al. began work on biologically-inspired axi-symmetric hexapods.

The past two decades have seen a flourish in the development of hexapod robots. Two of the more notable examples are Quinn's work with robots based on the cockroach [5–9, 20], and JPL's work on a utility rover [1–4]. Quinn's work with biologically-inspired hexapod robots, has been very extensive from a purely mechanical/kinematic view. The robots developed by the Case Western University team has ranged from simple kinematics similar to the OSU hexapod, to a pneumatically-actuated robot with kinematics directly modeled after and almost as complex as the *Blaberus gigantius* cockroach. The teams "abstracted biological" approach lead to the development of the wheel/leg crossover Whegs, which provides exceptional maneuverability in hexapod form. JPL's development of utility rovers began with axi-symmetric hexapods; however, the recent LEMUR IIb has a radially-symmetric limb mounting. This arrangement provides for no set front or back to the robot. This approach has grown in popularity sence the early 1980's Odetics robot, and allows for walking algorithms with near-equal maneuverability in any direction.

1.3 Hexapod Walking Algorithms

Considerable work has been done in generating walking algorithms for hexapod robots. In 1983 Raibert and Southerland developed a hexapedal walking machine capable of navigating rough terrain using onboard computer-controlled gaits [10]. A few years later, Brooks, et al. developed distributed networks which used layered control for robot control as well as for hexapedal walking algorithm control [21, 22]. Since the late 1980s, Quinn, et al. have been controlling hexapedal robots using neural networks based on the cockroach [5–9]. This method of control uses interconnected neurons which use excitatory or inhibitory control over each other. The arrangement of these

neurons produces various predictable gait patterns in the robot legs, depending on the speed required. This neural network control also proved very robust and was able to produce working gaits even with damage to the robot.

Many walking algorithms have been developed for hexapedal robots with limbs arranged symmetrically on either side of a longitudinal body axis, similar to an insect. Gaits for bodies with limbs arranged axially symmetric, have been defined by Song and Waldron [10], as:

Periodic

- Wave gait: seeping motions run from the rear to the front and legs on opposite sides of the body are 180 degrees out of phase
- Equal phase gait: all leg movements are ordered so that power consumption is consistent, like the wave gait motions run from rear to front
- Backward wave gait: similar to the wave gait except that motions run from front to rear
- Backward equal phase gait: similar to the equal phase gait except that motions run from front to rear
- Dexterous periodic gait: a follow the leader gait with the ability to adjust the placement of the two front feet
- Continuous follow-the-leader gait: feet are placed in the foot print of the foot ahead

Non-Periodic

- Discontinuous follow-the-leader gait: feet are placed in the foot print of the foot ahead, only one foot at a time is moved for greater stability
- Large obstacle gait: leg and body motions coordinate to traverse large obstacles while maintaining stability
- Precision footing gait: the operator either controls an individual leg with 3 DOF or controls the body with 6 DOF
- Free gait: used for avoidance of areas not suitable for weight bearing

where periodic gaits are generally preferable because they are easily implemented and can provide smother motion.

Various periodic wave gaits have been used for hexapedal robots, combined with biologically inspired coordination mechanisms found in stick insects [5]. However, MARS limbs are arranged around the body with radial symmetry, not arranged in rows on either side of the body. This, combined with the kinematic design of the limbs, invites the possibility of omni-directional motion. For this reason a walking algorithm which builds on the maximized omni-directional step length described by Schmiedeler [23] was chosen. This algorithm would be a combination of a tripedal wave gait and a precision footing gait. That is, the algorithm would be based on an alternating tripod gait, but with the capability of precisely positioning each limb tip within the workspace.

While many walking algorithms [5, 10] would be suitable for such planar hexapedal locomotion, developing one sufficiently general enough to handle all navigable terrain and to utilize the kinematic structure of the robot adds to the problem complexity. The adaptable gait-planning algorithms under development are basic in the sense that they are currently only capable of planar locomotion, but general in that they could be used as the foundations for a more sophisticated algorithm capable of navigating complex terrain such as the surface of a spacecraft. It is also desirable that the basic elements of a walking algorithm be applicable in using the limbs to manipulate tools. For these reasons, suitable base walking algorithms, while currently only capable of planer locomotion, must be capable of precise, pre-determined limb tip positioning. Also, the kinematic structure of the robot allows for body translation in any 3-space direction, as well as for pitch, yaw, and roll, while walking. Therefore, in order not to exclude mechanical capabilities, the base algorithm will be capable of instantaneously and simultaneously executing any combination of translations and change of orientation of the body while walking.

1.4 Discussion

This thesis focuses on the development of walking algorithms based on a defined workspace. While there are other approaches to walking algorithm development, such as biologically inspired central pattern generators, these methods were not employed for the walking algorithms presented here. Rather,

the algorithms are generated within the constraints of the limb kinematics and desired walking functionality. This approach highlights how walking algorithms effect workspace and provides for precise body and limb positioning; however, a biologically inspired approach should prove less computationally intensive and may provide for faster development of more sophisticated walking algorithms.

The workspace analyses presented in this thesis not only provides the foundation for the kinematically based walking algorithm development, but also provides a useful design tool for design of a limbed robot. This analysis has shown how walking algorithm requirements can effect shape of the workspace while providing a geometric approach for mathematically defining the workspace. The approach used to define the MARS workspace is applicable to limbs with differing kinematics. Further, understanding workspace reduction based on pre-defined gait characteristics, such as how constraining a gait to be continuos reduces the usable workspace, can be used during the design phase of limbs.

Chapter 2

Robot Design

MARS was designed to be light weight while remaining relatively simple and inexpensive to build. The design makes use of carbon fiber composite as well as machined aluminum components and off-the-shelf molded polystyrene components. The resulting hexapod frame is capable of walking while supporting more than the weight of the actuators and structure. While future plans include an onboard computer, a battery pack and sensors, at present the power source and gait generation computer are both external to the robot–power and actuator commands are transfered umbilically.

2.1 Limb Design

MARS is kinematically and dimensionally similar to the JPL LEMUR IIb robot [2]. Each of the six limbs has four revolute actuators and therefore four degrees of freedom. The limb attaches to the body of the robot with a 3DOF proximal joint. In this joint the axes of three revolute actuators intersect orthogonally at a single point. The result is a kinematically spherical joint which can be equated with a ball and socket joint. The remaining 1-DOF distal joint uses a single revolute actuator located between the inner and outer limb sections.

As with the LEMUR class robots, the MARS limb design simplifies the kinematics, resulting in a large workspace [1]. The use of the spherical proximal joint simplifies the limb kinematics.

Carbon-fiber composite, aluminum, and polystyrene were used to form the structural limb and body components of MARS in order to reduce weight

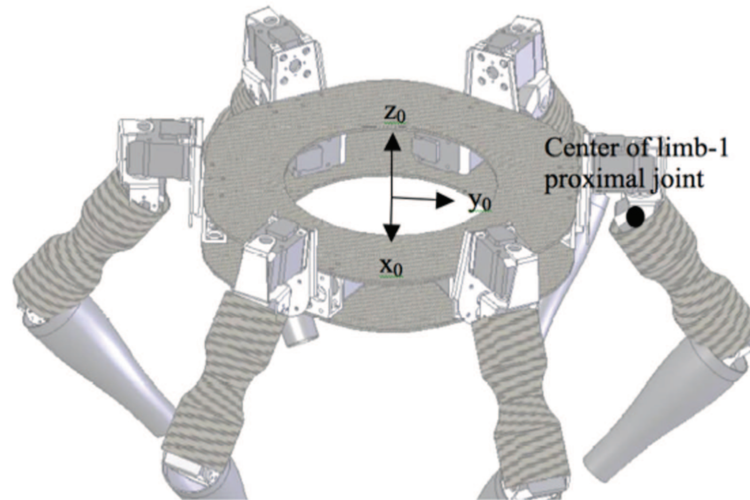

Figure 2.1: The y-axis of the body coordinate frame points toward the center of the proximal joint of Limb 1.

and maximize stiffness. This lightweight design allowed for the use of compact Dynamixel DX-117 actuators for all 24 revolute joints. The actuators provide sufficient torque for the robot to be fully supported by three limbs in any statically stable position. While these actuators are capable of 300 degrees of rotation, only one of the actuators on each limb is free to use this range. The other three actuators are physically limited by the structure of the limb.

the joint rotations are structurally limited for three of the four degrees of freedom.

2.2 Coordinate Frame Definitions

The workspace analysis covered is based on Schmledeler, Bradley, and Kennedy [23]. For that reason their coordinate frame definitions will be followed. The coordinate frame of the body of the robot (x_0, y_0, z_0) is positioned at the center of the body with the y-axis pointing directly at the center of the Limb 1 proximal joint and the z-axis extending upward away from the body, as shown in Fig. **??**. The revolute joints are then assigned coordinate frames in accordance with the Denavit-Hartenburg convention, as shown in Fig. 2.2.

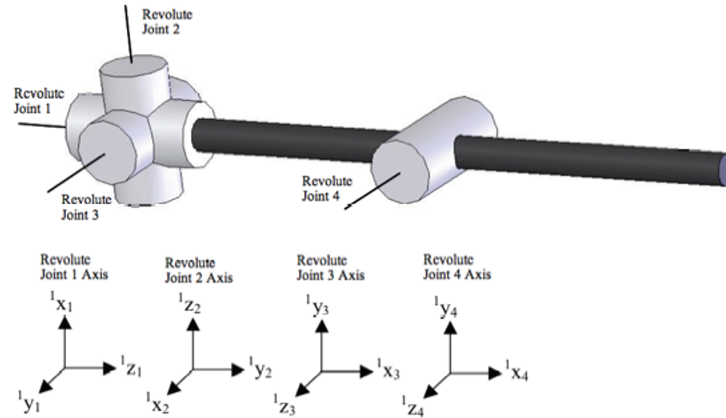

Figure 2.2: Coordinate frames are assigned to each revolute joint on a limb.

2.3 Proximal Joint

The proximal joint was constructed so that Revolute Joints 1, 2, and 3 would intersect at a single point arranged as shown in Figure 2.3. While a U-shaped support structure attaching at either side of the actuator was used for revolute joints two through four, this type of construction was not desired for Revolute Joint 1 because it would have limited maneuverability. One of the main concepts of the MARS design is the ability to use limbs for both walking and selecting tools from the back of the robot. To obtain this functionality, Revolute Joint 1 was not limited beyond the maximum 300 degrees of rotation of the DX-117 through the use of a U-shaped support. Rather the shaft-bracket, shown in Figure 2.4 was used. In this design the shaft-bracket is supported at the edge of the robot body by a bearing and within the robot body by the actuator shaft. The shaft-bracket and actuator-1 are held by the limb-support, and the shaft-bracket supports actuator-2. Revolute Joints 2 and 3 are connected by a single proximal frame. The proximal frames were laser-cut from sheet aluminum and bent to shape.

2.4 Distal Joint

The single revolute joint of the distal joint is supported by the distal frame shown in Figure 2.5. The distal frame was manufactured in the same way as the proximal frame.

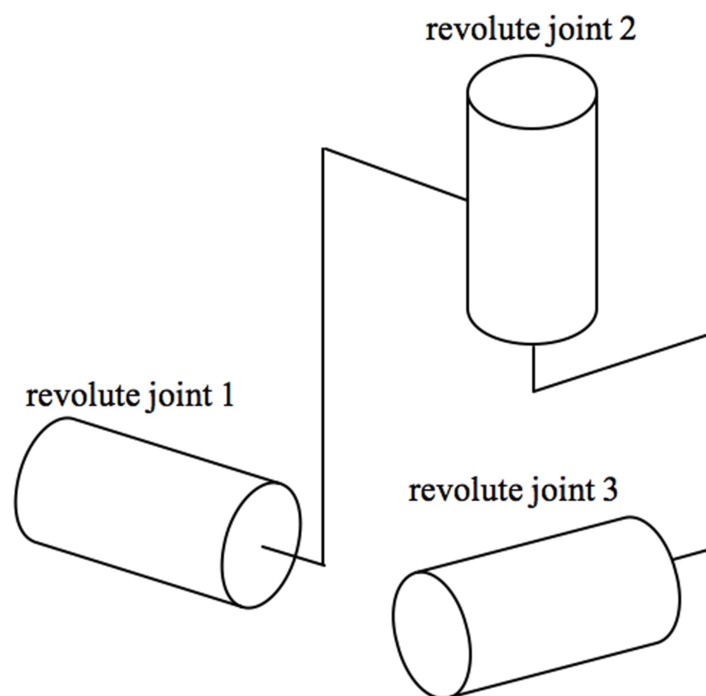

Figure 2.3: In the proximal joint, Revolute Joints 1, 2, and 3 intersect at a single point.

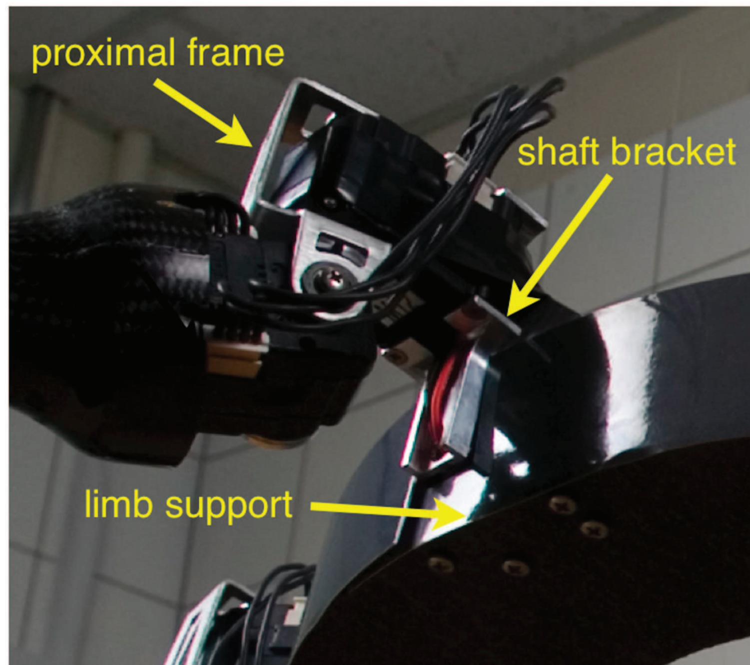

Figure 2.4: The proximal joint design utilizes the entire range of the Revolute Joint 1 actuator while providing a well supported framework for all three revolute joints. Courtesy Virginia Tech. Photograph taken by Josh Armstrong.

Figure 2.5: The distal joint also provides support on both sides of the actuator, similar to the revolute joints two and three of the proximal joint. Courtesy Virginia Tech. Photograph taken by Josh Armstrong.

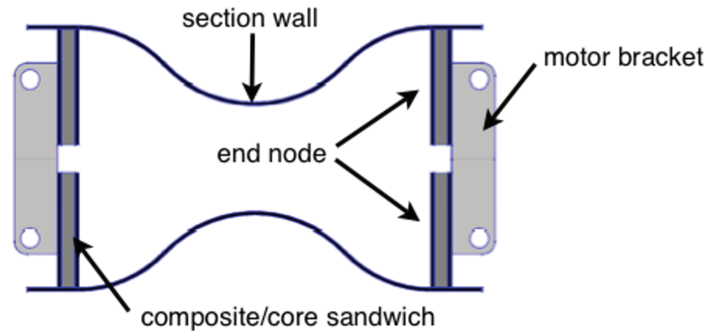

Figure 2.6: The proximal limb section, constructed primarily of single layer carbon fiber and composite/core sandwich end nodes, provides high strength and rigidity with low weight.

2.5 Proximal Limb Section

The proximal limb section, which serves as the physical link between actuators 3 and 4, was designed to be light weight as well as rigid under compressive and bending loads. As shown Figures 2.6 and 2.7, this section is made primarily of carbon fiber composite. While the end nodes use a composite-core sandwich, the use of a core is not necessary on the section wall. This is due to the complex curvature of the wall, where bending in any area requires stretching in another. While the thin section wall can easily bend under load, the carbon fiber greatly resists stretching.

A clam shell process was used to mold the thin section wall. First a glass tube was crafted into the desired shape of the proximal limb section. This tube was used to form a clam shell plastic mold. The glass tube, ends plugged with clay and surface coated with mold release, was set in a lined container. The container was filled half way with two-part plastic, which solidified to form one half of the clam shell mold. Once solidified, the exposed surface of the plastic was coated with mold release and the container was filled with two-part plastic. Once solidified the resulting clam shell mold was opened and the glass tube removed. The clam shell was then used to cast the proximal limb section wall. A plastic bag was inserted into one layer of woven carbon fiber sleeve. The sleeve was coated with epoxy and inserted into the closed clam-shell mold. With the mold set vertically, as shown in Figure 2.7, the plastic bag was packed with salt to press the carbon fiber sleeve outward

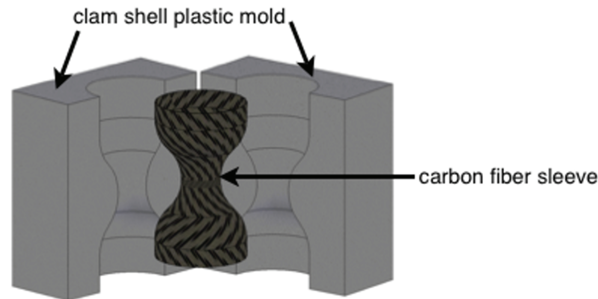

Figure 2.7: A clam-shell mold was used to cast the proximal limb sections. Granular salt pressed inside a carbon fiber sleeve forces the sleeve to conform to the inside of the mold.

against the mold. Weights were used to compress the salt, while the epoxy cured.

2.6 Distal Limb Section

The distal limb section is composed of an outer cone, a reinforcing disk, a distal motor bracket, a flange, and an aluminum tube. These components are assembled as shown in Figure 2.8. The outer cone and reinforcing disk form a foot which attaches with a friction fit to the distal bracket assembly. The distal bracket assembly–composed of the distal motor bracket, flange, and aluminum tube–allow for quick conversion between the foot and other possible distal attachments, such as tools. The outer cone is a thin polystyrene shell painted black to match the robots other carbon components. The reinforcing disk is a carbon fiber sandwich attached with epoxy into the large end of the cone. The aluminum tube is attached with epoxy inside the flange, which is in turn fastened with screws to the distal motor bracket. The distal motor bracket attaches via motor flange and bearing to the distal actuator.

2.7 Body

The body of the robot was designed to be light weight and rigid while providing space and flexibility for installing batteries and a computer. The body

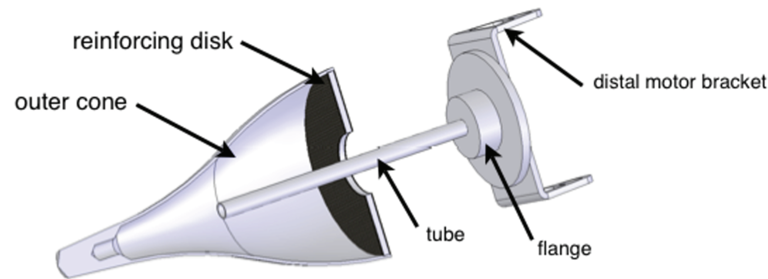

Figure 2.8: A friction fit secures the polystyrene and carbon fiber distal limb section to an aluminum bracket and tube assembly attached to the distal joint actuator.

design is structurally based on a typical carbon fiber composite core sandwich, in which a core is sandwiched between two laminates of carbon fiber as shown in Figure 2.9. In such a structure, the core prevents change in distance and orientation between the laminates, while the carbon resists stretching of the laminates. The combination results in a rigid, high strength, low weight composite. For the MARS body, however, the limb supports take the place of the core. Attached to the limb supports with machine screws, the structure provides a rigidity similar to a traditional core carbon sandwich. Weight is further reduced by removing the center of the carbon layers.

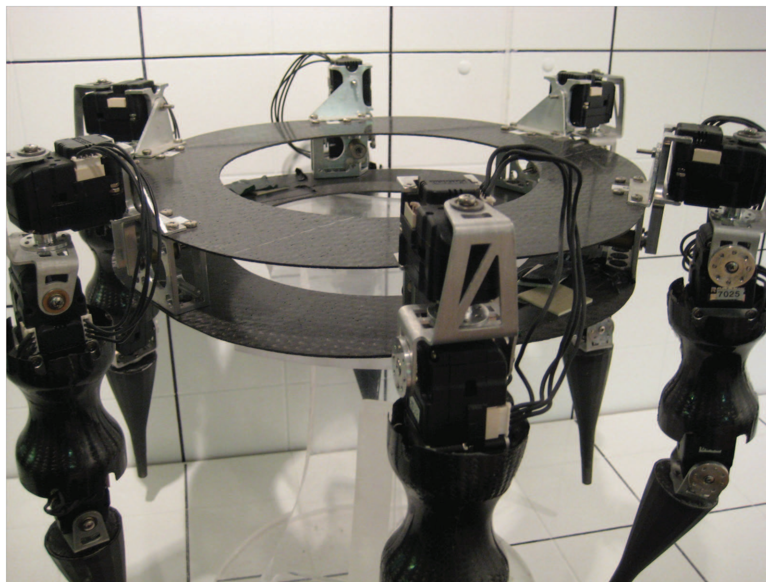

Figure 2.9: Two thin carbon-fiber plates, separated by limb supports, forms the body of the robot.

Chapter 3

Kinematics and Singularity Analysis

3.1 Forward Kinematics

The forward kinematics for the MARS limbs are found using the Denavit-Hartenburg convention. In this section the forward kinematics are found for a limb with all four DOF, as well as for the three DOF situation where Revolute Joint 1 is fixed. The Denavit-Hartenburg parameters for the full limb are presented in Table 3.1.

Table 3.1: Denavit-Hartenburg Parameters

Link	a_i	α_i	d_i	θ_i
1	0	$\frac{\pi}{2}$	0	$\frac{\pi}{2}+\theta_1$
2	0	$\frac{\pi}{2}$	0	$\frac{\pi}{2}+\theta_2$
2	L_1	0	0	θ_3
4	d_2	0	0	θ_4

The homogeneous transformations A_1 through A_4 are given in Equations 3.1, 3.2, 3.3, and 3.4.

$$A_1 = \begin{bmatrix} C\theta_1 & -S\theta_1 & 0 & 0 \\ S\theta_1 & C\theta_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & C\frac{\pi}{2} & -S\frac{\pi}{2} & 0 \\ 0 & S\frac{\pi}{2} & C\frac{\pi}{2} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A_1 = \begin{bmatrix} C\theta_1 & -S\theta_1 & 0 & 0 \\ S\theta_1 & C\theta_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.1}$$

$$A_1 = \begin{bmatrix} C\theta_1 & 0 & S\theta_1 & 0 \\ S\theta_1 & 0 & -C\theta_1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A_2 = \begin{bmatrix} C\theta_2 & -S\theta_2 & 0 & 0 \\ S\theta_2 & C\theta_2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & C\frac{\pi}{2} & -S\frac{\pi}{2} & 0 \\ 0 & S\frac{\pi}{2} & C\frac{\pi}{2} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A_2 = \begin{bmatrix} C\theta_2 & -S\theta_2 & 0 & 0 \\ S\theta_2 & C\theta_2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.2}$$

$$A_2 = \begin{bmatrix} C\theta_2 & 0 & S\theta_2 & 0 \\ S\theta_2 & 0 & -C\theta_2 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A_3 = \begin{bmatrix} C\theta_3 & -S\theta_3 & 0 & 0 \\ S\theta_3 & C\theta_3 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & d_1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.3}$$

$$A_3 = \begin{bmatrix} C\theta_3 & -S\theta_3 & 0 & C\theta_3 d_1 \\ S\theta_3 & C\theta_3 & 0 & S\theta_3 d_1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$
\begin{aligned}
A_4 &= \begin{bmatrix} C\theta_4 & -S\theta_4 & 0 & 0 \\ S\theta_4 & C\theta_4 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & d_2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
A_4 &= \begin{bmatrix} C\theta_4 & -S\theta_4 & 0 & C\theta_4 d_2 \\ S\theta_4 & C\theta_4 & 0 & S\theta_4 d_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\end{aligned}
\tag{3.4}
$$

To find the Transformation Matrix T_0^4, which includes all four degrees of freedom:

$$T_0^4 = A_1 A_2 A_3 A_4 \tag{3.5}$$

$$T_0^4 = \begin{bmatrix} r_{11} & r_{12} & r_{13} & d_x \\ r_{21} & r_{22} & r_{23} & d_y \\ r_{31} & r_{32} & r_{33} & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.6}$$

where

$$r_{11} = C\theta_4 \left(C\theta_1 C\theta_2 C\theta_3 + S\theta_1 S\theta_3 \right) + \left(C\theta_3 S\theta_1 - C\theta_1 C\theta_2 S\theta_3 \right) S\theta_4 \tag{3.7}$$

$$r_{21} = C\theta_4 \left(C\theta_2 C\theta_3 C\theta_1 - C\theta_1 S\theta_3 \right) + \left(-C\theta_1 S\theta_3 - C\theta_2 S\theta_1 S\theta_3 \right) S\theta_4 \tag{3.8}$$

$$r_{31} = C\theta_3 C\theta_4 S\theta_2 - S\theta_2 S\theta_3 S\theta_4 \tag{3.9}$$

$$r_{12} = C\theta_4 \left(C\theta_3 S\theta_1 - C\theta_1 C\theta_2 S\theta_3 \right) - \left(C\theta_1 C\theta_2 C\theta_3 + S\theta_1 S\theta_3 \right) S\theta_4 \tag{3.10}$$

$$r_{22} = C\theta_4 \left(-C\theta_1 C\theta_3 - C\theta_2 S\theta_1 S\theta_3 \right) - \left(C\theta_2 C\theta_3 S\theta_1 - C\theta_1 S\theta_3 \right) S\theta_4 \tag{3.11}$$

$$r_{32} = -C\theta_4 S\theta_2 S\theta_3 - C\theta_3 S\theta_2 S\theta_4 \tag{3.12}$$

$$r_{13} = C\theta_1 S\theta_2 \tag{3.13}$$

$$r_{23} = S\theta_1 S\theta_2 \tag{3.14}$$

$$r_{33} = -C\theta_2 \tag{3.15}$$

$$\begin{aligned} d_x = \; & C\theta_1 C\theta_2 C\theta_3 d_1 + S\theta_1 S\theta_3 d_1 + \\ & C\theta_4 \left(C\theta_1 C\theta_2 C\theta_3 + S\theta_1 S\theta_3\right) d_2 + \\ & \left(S\theta_3 S\theta_1 - C\theta_1 C\theta_2 S\theta_3\right) S\theta_4 d_2 \end{aligned} \tag{3.16}$$

$$\begin{aligned} d_y = \; & C\theta_2 C\theta_3 S\theta_1 d_1 - C\theta_1 S\theta_3 d_1 + \\ & C\theta_4 \left(C\theta_2 C\theta_3 S\theta_1 - C\theta_1 S\theta_3\right) d_2 + \\ & \left(-C\theta_1 C\theta_3 - C\theta_2 S\theta_1 S\theta_3\right) S\theta_4 d_2 \end{aligned} \tag{3.17}$$

$$d_z = C\theta_3 S\theta_2 d_1 + C\theta_3 C\theta_4 S\theta_2 d_2 - S\theta_2 S\theta_3 S\theta 4 d_2 \tag{3.18}$$

However, as discussed in Section 4.1, Revolute Joint 1 is not used for walking. Therefore the transformation matrix T_1^4 is used for the resulting 3DOF limb:

$$T_1^4 = A_2 A_3 A_4 \tag{3.19}$$

$$T_1^4 = \begin{bmatrix} R_{11} & R_{12} & R_{13} & D_x \\ R_{21} & R_{22} & R_{23} & D_y \\ R_{31} & R_{32} & 0 & D_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.20}$$

where

$$R_{11} = C\theta_2 C\theta_3 C\theta_4 - C\theta_2 S\theta_3 S\theta_4 \tag{3.21}$$

$$R_{31} = C\theta_3 C\theta_4 S\theta_2 - S\theta_2 S\theta_3 S\theta_4 \tag{3.22}$$

$$R_{31} = C\theta_4 S\theta_3 + C\theta_3 S\theta_4 \tag{3.23}$$

$$R_{12} = -C\theta_2 C\theta_4 S\theta_3 - C\theta_2 C\theta_3 S\theta_4 \tag{3.24}$$

$$R_{22} = -C\theta_4 S\theta_2 S\theta_3 - C\theta_3 S\theta_2 S\theta_4 \tag{3.25}$$

$$R_{32} = C\theta_3 C\theta_4 - S\theta_3 S\theta_4 \tag{3.26}$$

$$R_{13} = S\theta_2 \tag{3.27}$$

$$R_{23} = -C\theta_2 \tag{3.28}$$

$$D_x = C\theta_2 C\theta_3 d_1 + C\theta_2 C\theta_3 C\theta_4 d_2 - C\theta_2 S\theta_3 S\theta_4 d_2 \tag{3.29}$$

$$D_x = C\theta_3 S\theta_2 d_1 + C\theta_3 C\theta_4 S\theta_2 d_2 - S\theta_2 S\theta_3 S\theta_4 d_2 \tag{3.30}$$

$$D_z = S\theta_3 d_1 + C\theta_4 S\theta_3 d_2 + C\theta_3 S\theta_4 d_2 \tag{3.31}$$

3.2 Inverse Kinematics

The inverse kinematics are derived geometrically for the MARS limb with Revolute Joint 1 frozen. With this constraint the limb has a total of 3DOF and a finite number of solutions exist when solving for the joint angles given only the (x,y,z) limb tip position. Figure 3.1 illustrates this derivation where the angles θ, ϕ, and α for Revolute Joints 2, 3, and 4 are given respectively by Equations 3.32, 3.36, and 3.40:

$$\theta = \arctan(y, x) \tag{3.32}$$

$$c = \sqrt{x^2 + y^2 + z^2} \tag{3.33}$$

The assumption is made:

$$B \leq \frac{\pi}{2} \tag{3.34}$$

This is a reasonable assumption as the segment lengths and joint limits of a MARS limb prohibit B greater than $\frac{\pi}{2}$. With this assumption the derivation continues:

$$B = \arccos\left(\frac{b^2 - a^2 + c^2}{2ac}\right) \tag{3.35}$$

The values a, b, and c are shown in Figure 3.1.

$$\phi = B - \arcsin\left(\frac{z}{c}\right) \tag{3.36}$$

$$x = \arcsin B \tag{3.37}$$

$$C_1 = \frac{\pi}{2} - B \tag{3.38}$$

$$C_2 = \arccos\frac{x}{b} \tag{3.39}$$

$$\alpha = \pi - C_1 - C_2 \tag{3.40}$$

3.3 Singularity Analysis

Singularities occur whenever one or more DOFs are lost [24]. For the MARS limbs an *elbow singularity* [25] exists whenever the proximal limb section is in line with the distal limb section–when x_3 is collinear with x_4 and $\theta_4 = 0$. This singularity occurs because in this position the DOF provided by Revolute Joint 4 is lost. In this position the limb tip reaches the outer boundary of the workspace. To avoid this singularity, in a walking algorithm, a buffer can be applied to the boundary of the workspace so that the algorithm will not position the limb tip beyond the buffer and therefore near the singularity. For this reason the variable “buffer” is included in the programming presented in Appendix A, and discussed in Section 5.3. Furthermore, a singularity could exist when $\theta_4 = \pi$; however, this is not possible on the MARS limbs due to the physical joint limits.

Another singularity within the MARS limb workspace occurs when the limb tip intersects the z_2-axis [26]. This *shoulder singularity* [25] occurs because the DOF provided by Revolute Joint 2 is lost [24]. In this orientation the limb tip position is independent of θ_2. This singularity can be defined by:

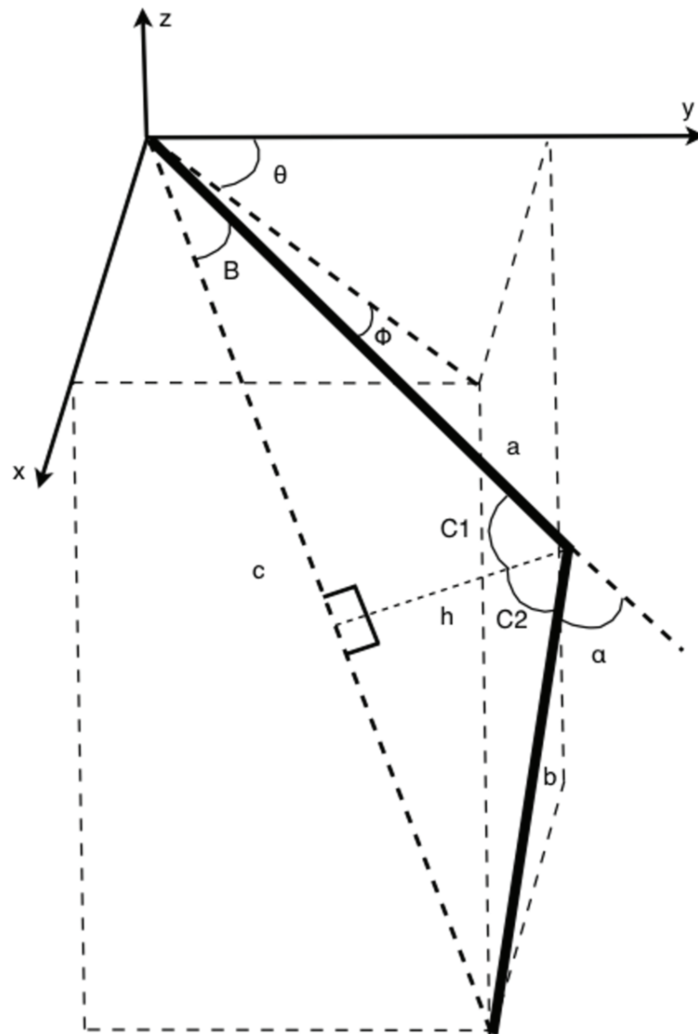

Figure 3.1: The inverse kinematics can be derived geometrically for the MARS limb with Revolute Joint 1 frozen.

$$\text{while} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} x \\ y \\ z \end{bmatrix} \text{ is independent of } \theta_2 \tag{3.41}$$

This is significant with relation to the walking algorithm because the algorithm operates in 3-space rather than joint space. Therefore, moving the limb tip in a line which passes infinitesimally near to the z_2-axis requires Revolute Joint 2 to rotate at near-infinite velocity. For the general walking algorithm developed in Section 5.2, the limb tip is prevented from nearing the z_2-axis by a buffer cylinder, as discussed in Section 4.6.

There are further singularities which do not come about due to the loss of a degree of freedom. These singularities, discussed in Section 4.5, occur when the limb tip is attempting to move continuously from one part of the workspace to another. These limitations are due to the requirements of the walking algorithm discussed in Chapter 5.

Chapter 4

Workspace Analysis

Understanding how the MARS limbs can move is key to forming walking algorithms. Interestingly, the workspace and walking algorithms are interdependent. Setting restrictions on the workspace constrain the possible walking algorithms and placing requirements on the walking algorithms restrict the workspace. While many different workspace and walking algorithm combinations are possible, this thesis only covers a few. However, understanding the workspace is key to understanding the relationship between workspace and walking algorithm. These relationships will be covered in more depth in Chapter 5. This chapter defines the workspaces used for the MARS limb in this thesis.

4.1 The General Workspace

The most inclusive or general workspace presented in this thesis is termed the "general workspace." This workspace is the volume containing all points a limb tip can reach with Revolute Joint 1 fixed at zero degrees–with the x_1 axis parallel to the z_0 axis. It would be possible to include Revolute Joint 1 unfixed, and use the even broader "total workspace." However, doing so would require a walking algorithm which provides more information than just the next limb tip position. For example, by specifying the limb tip position as well as the distal limb section orientation in 3-space, positioning the 4-DOF limb would be possible. However, the assumption was made that it would be simpler to develop a walking algorithm which need only generate the next limb tip position. Therefore, Revolute Joint 1 is not used for walking.

Table 4.1: MARS Limb Range of Operation.

Revolute Joint Angle	Joint Operable Range in Degrees	Axis of Rotation
θ_1	-60 to 240	z_1
θ_2	-10 to 190	z_2
θ_3	-110 to 0	z_3
θ_4	-90 to 90	z_4

Defining the limb workspace is contingent upon limits placed on the unfixed revolute joints. The physical limits of θ_1, θ_2, θ_3, and θ_4 are given in Table 4.1. While the workspace exists as a complex 3D shape, visualization of the work space is eased by examining the limits of θ_3 and θ_4 in 2-D. With Revolute Joint 2 fixed at zero degrees the limb extends away from the robot body in the y_2-z_2 plane. Within this plane the limb is simplified to a two link planar manipulator as only Revolute Joints 3 and 4 are used. Employing the full range of motion of Revolute Joint 3 and Revolute Joint 4, the limb tip is capable of any point within the crescent shown in Figure 4.1. Revolving this 2-D shape about the z_2-axis through the full range of motion of Revolute Joint 3, from -10 to 190 degrees, results in the full 3D workspace with Revolute Joint 1 fixed, as shown in Figure 4.2.

4.2 The General Knee Up Workspace

By constraining Revolute Joint 1, only two configurations are available for a given tip position; "knee-up" and "knee-down." While it is possible for the limb tip to touch any point within the general workspace with Revolute Joint 1 fixed, the entire workspace is not used for walking. This is due to the limitations of tracing a line through the workspace with the limb tip. While the limb tip is in contact with the floor it must move in a continuous line with respect to the robot body. This is equivalent to moving along a continuous line within the 3D workspace. However, the general workspace contains regions between which a continuous line cannot be traced, except in certain situations. These regions are defined by the position of Revolute Joint 4. The operable range of Revolute Joint 4 is divided into two ranges:

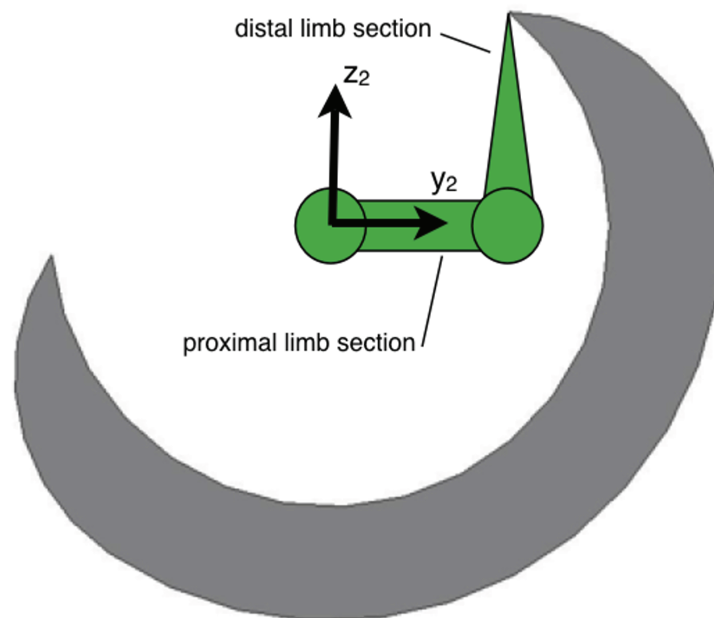

Figure 4.1: In 2-D Revolute Joints 3 and 4 can sweep out this crescent shape.

- -90 degrees to 0 degrees
- 0 degrees to 90 degrees

Figure 4.3 shows the 2-D and 3D work spaces associated with the two operable ranges or configurations for Revolute Joint 4. Tracing a continuous line with the limb tip while transitioning from one configuration to the other requires at least one of the following constraints:

- Fully extending the limb (passing Revolute Joint 4 through 0 degrees)
- Instantaneously rotating Revolute Joint 2 through 180 degrees, or pausing the step motion while Revolute Joint 2 rotates 180 degrees (this transition only works when the limb tip is directly beneath the center of the proximal joint on the negative z_2 axis).
- Making Revolute Joint 1 unfixed, which would result in infinite solutions to any tip position.

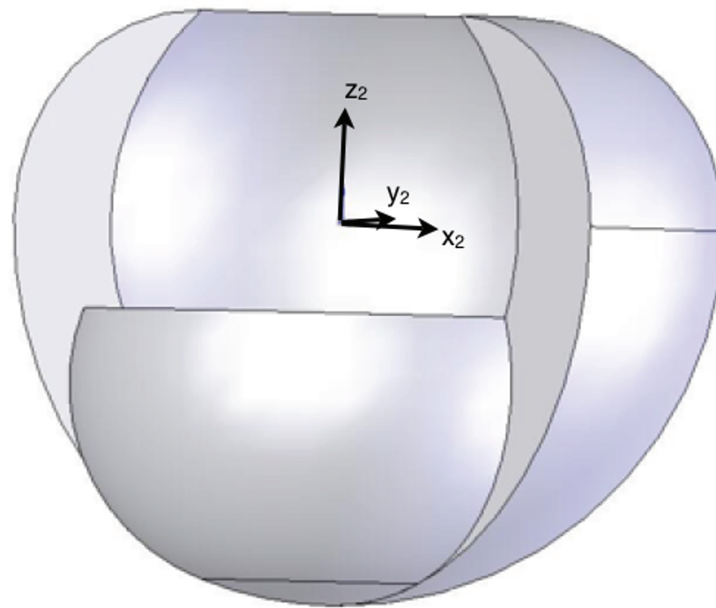

Figure 4.2: Revolute Joints 2, 3 and 4 can sweep out this 3D volume–the general workspace.

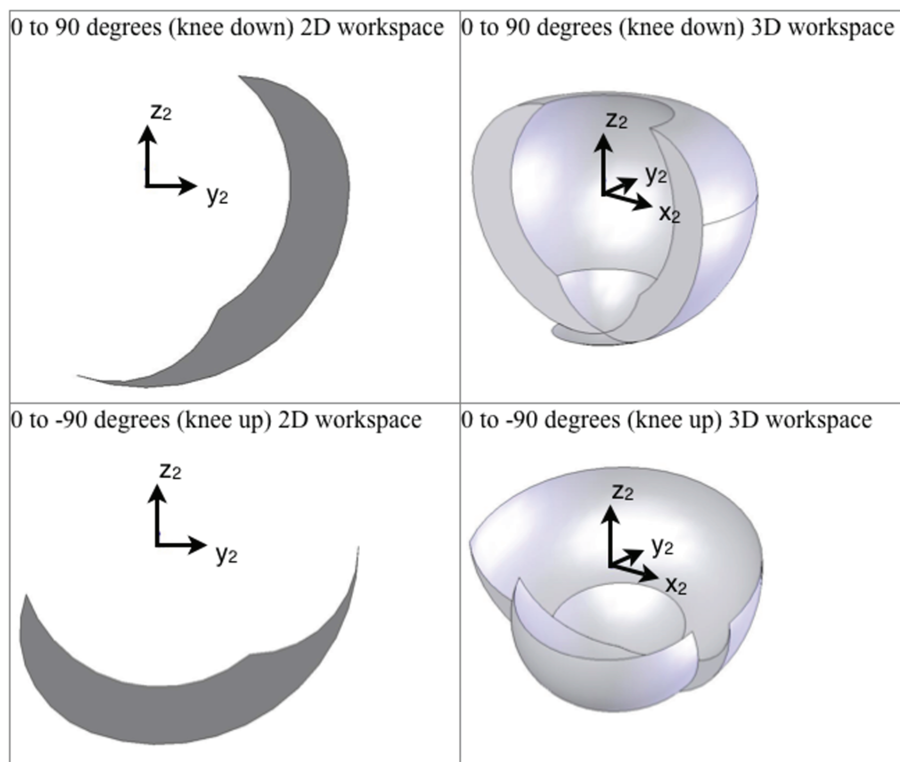

Figure 4.3: Of the two possible configurations, the -90 to 0 degrees knee-up configuration is used for walking.

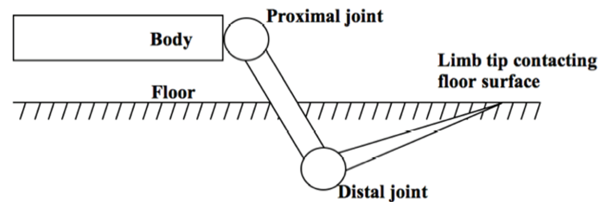

Figure 4.4: Knee-down limb configuration requires the limb to extend through the floor for certain walking heights.

Because the proposed walking algorithm should remain independent of all these constraints, operation within only one of the two configurations must be considered. Operation within the 0 to 90 degrees configuration (knee-down) is excluded and the -90 to 0 degrees configuration (knee-up) is selected. This is mainly based on the assumption that the walking surface will be located, for the most part, beneath the robot body. In such an orientation, the intersection of the walking surface with the workspace will be more likely to yield a larger 2D workspace for knee-up than for knee-down. Further, the 0 to 90 degrees configuration (knee-down) would result in scenarios where the limb would have to enter the walking surface for the limb tip to contact the walking surface. This impossible scenario is shown in Figure 4.4.

4.3 Similarities to Insect Limbs

It is interesting to note that insects also exhibit the knee-up configuration. The insect limb [27] shown in Figure 4.5 is more complex than a MARS limb; however, analogies can be made between the two:

- The proximal joint on MARS is analogous to the joints between the body, coxa, trochaner, and femur sections of an insect.
- The distal joint on MARS is analogous to the joint between the femur and tibia sections of an insect.
- The remaining tarsomere sections of the insect serve as a foot for which MARS uses the tip of the outer limb section.

Generally in insects, the tibia section angles downward from the femur section. This configuration increases the workspace for walking and generally

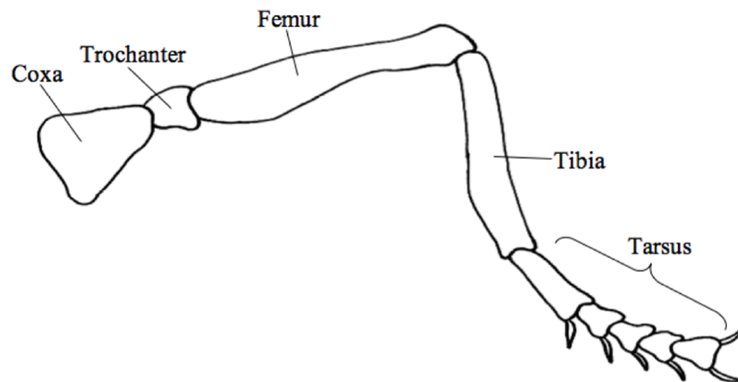

Figure 4.5: An insect limb uses a similar joint configuration to the MARS limb.

orients the tibia towards the ground, which results in minimizing the torque needed at the joints. While the emulation of nature was not the main concern in the design or limb configuration of MARS, insect kinematic similarities can prove desirable. Biological kinematic similarities allow for more direct application of often-superior biologically-inspired locomotion techniques [7].

4.4 Geometric Delineation of the General Knee Up Workspace

For the walking algorithms discussed in this thesis, only the knee-up workspace is used. The geometry of the general knee-up workspace can be completely defined mathematically. This mathematical definition of the workspace boundary is necessary for the walking algorithm. The walking algorithm operates in the space domain rather than the actuator domain. For this reason, defining the boundaries of the workspace in the space domain is a necessary preliminary to formulating the walking algorithm.

The 2D workspace is examined in the z_2-y_2 plane as shown in Figure 4.6. Sweeping Revolute Joints 3 and 4 through their respective ranges causes the limb tip to reach all points within the area bounded by curves 1 through 4. Curves 1 and 2 are the result of sweeping Revolute Joint 3 though its range while Revolute Joint 4 is at the two extremes of its range. Curves 3 and 4 are the result of sweeping Revolute Joint 4 through its range while Revolute

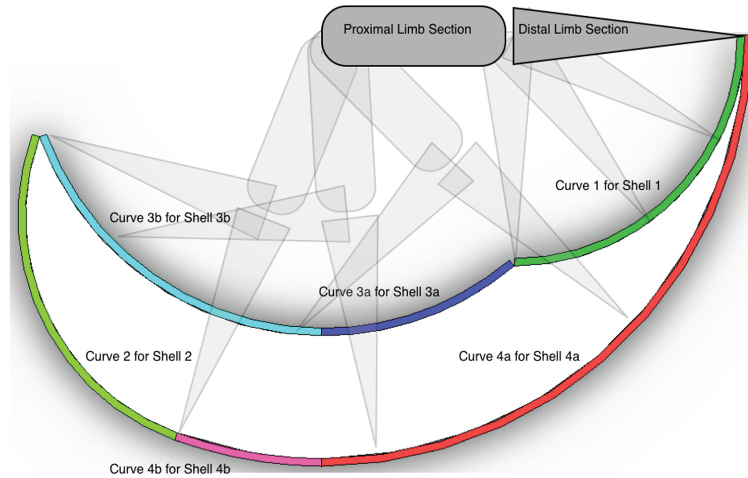

Figure 4.6: In the knee-up configuration the 2D workspace in the z2-y2 plane is the area contained within four curves.

Joint 3 is at the two extremes of its range. Curves 1, 2, 3, and 4 are defined by Equations 4.1, 4.3, 4.5, and 4.7 respectively.

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \sqrt{L_2^2}\cos\theta + L_1 \\ \sqrt{L_2^2}\sin\theta \end{bmatrix} \tag{4.1}$$

where

$$\theta = \frac{-90\pi}{180}...0 \tag{4.2}$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \sqrt{L_2^2}\cos\theta - L_1\sin\frac{20\pi}{280} \\ \sqrt{L_2^2}\sin\theta - L_1\cos\frac{20\pi}{180} \end{bmatrix} \tag{4.3}$$

where

$$\theta = \frac{-200\pi}{180}...\frac{-110\pi}{180} \tag{4.4}$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \sqrt{L_1^2 + L_2^2}\cos\theta \\ \sqrt{L_1^2 + L_2^2}\sin\theta \end{bmatrix} \tag{4.5}$$

where

$$\theta = \left(\frac{-110\pi}{180} - \arctan\frac{L_2}{L_1}\right)... - \arctan\frac{L_2}{L_1} \tag{4.6}$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} (L_1 + L_2)\cos\theta \\ (L_1 + L_2)\sin\theta \end{bmatrix} \tag{4.7}$$

where

$$\theta = \frac{-110\pi}{180}...0 \tag{4.8}$$

Rotating the 2D workspace about the z_2 axis through the range of Revolute Joint 2 generates the 3D workspace shown in Figure 4.7. The shells which comprise the 3D workspace are formed by sweeping the 2D curves about the z_2 axis through the range of Revolute Joint 2. However, as 2D curves 3 and 4 cross the z_2 axis, they are separated into Shells 3a, 3b, 4a, and 4b. Shells 1 and 2 are torus sections, while Shells 3a, 3b, 4a, and 4b are of sections of spheres centered at origin 2. Shells 1, 2, 3a, 3b, 4a, and 4b are mathematically defined in Equations 4.9, 4.11, 4.13, 4.15, 4.17, and 4.19 respectively:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} (L_1 + L_2\cos v)\cos u \\ (L_1 + L_2\cos v)\sin u \\ L_2\sin v \end{bmatrix} \tag{4.9}$$

where

$$u = \tfrac{-10\pi}{180}...\tfrac{190\pi}{180}, v = \tfrac{-90\pi}{180}...\tfrac{0\pi}{180} \tag{4.10}$$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \left(L_1\sin\left(\tfrac{20\pi}{180}\right) + L_2\cos v\right)\cos u \\ \left(L_1\sin\left(\tfrac{20\pi}{180}\right) + L_2\cos v\right)\sin u \\ L_2\sin v - L_1\cos\left(\tfrac{20\pi}{180}\right) \end{bmatrix} \tag{4.11}$$

where

$$u = \tfrac{190\pi}{180}...\tfrac{350\pi}{180}, v = \tfrac{20\pi}{180}...\tfrac{-70\pi}{180} \tag{4.12}$$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \sqrt{L_1^2 - L_2^2 - u^2}\cos\theta \\ \sqrt{L_1^2 - L_2^2 - u^2}\sin\theta \\ u \end{bmatrix} \tag{4.13}$$

where

$$\theta = \tfrac{-10\pi}{180}...\tfrac{190\pi}{180}, u = L_2... - \sqrt{L_1^2 - L_2^2} \tag{4.14}$$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \sqrt{L_1^2 - L_2^2 - u^2}\cos\theta \\ \sqrt{L_1^2 - L_2^2 - u^2}\sin\theta \\ u \end{bmatrix} \tag{4.15}$$

where

$$\theta = \frac{190\pi}{180} \ldots \frac{350\pi}{180}, \\ u = \sqrt{L_1 + L_2}\cos\left(\frac{20\pi}{180} + \arctan\frac{L_2}{L_1}\right) \ldots - \sqrt{L_1^2 - L_2^2} \tag{4.16}$$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \sqrt{(L_1 - L_2)^2 - u^2}\cos\theta \\ \sqrt{(L_1 - L_2)^2 - u^2}\sin\theta \\ u \end{bmatrix} \tag{4.17}$$

where

$$\theta = \frac{-10\pi}{180} \ldots \frac{190\pi}{180}, u = 0 \ldots - (L_1 + L_2) \tag{4.18}$$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \sqrt{(L_1 - L_2)^2 - u^2}\cos\theta \\ \sqrt{(L_1 - L_2)^2 - u^2}\sin\theta \\ u \end{bmatrix} \tag{4.19}$$

where

$$\theta = \frac{190\pi}{180} \ldots \frac{350\pi}{180}, \\ u = -11\cos\left(\frac{20\pi}{180}\right) \ldots - (L_1 + L_2) \tag{4.20}$$

However, the shells in Figure 4.7 do not fully contain the workspace. In addition to these six shells, two planar section also bound the workspace. These planar sections lie on two planes, both of which contain the z_2-axis. These two planes intersect the x_2-y_2 plane at -10 degrees and 190 degrees. As the workspace is symmetric on either side of the y_2-z_2 plane, the two planar sections are identical. The form of these sections is shown in Figure 4.8. Each planar section is formed from two areas bounded by arcs. The equations of the circles which form the arcs A, B, C, and D are given by Equations 4.21, 4.22, 4.23, and 4.24 respectively:

$$x^2 + y^2 = L_1^2 + L_2^2 \tag{4.21}$$

$$(x - L_1)^2 + y^2 = L_2^2 \tag{4.22}$$

$$\left(x - \sin\left(\frac{20\pi}{180}\right)\sqrt{L_1^2 + L_2^2}\right)^2 + \left(y - \cos\left(\frac{20\pi}{180}\right)\sqrt{L_1^2 + L_2^2}\right)^2 = L_2^2 \tag{4.23}$$

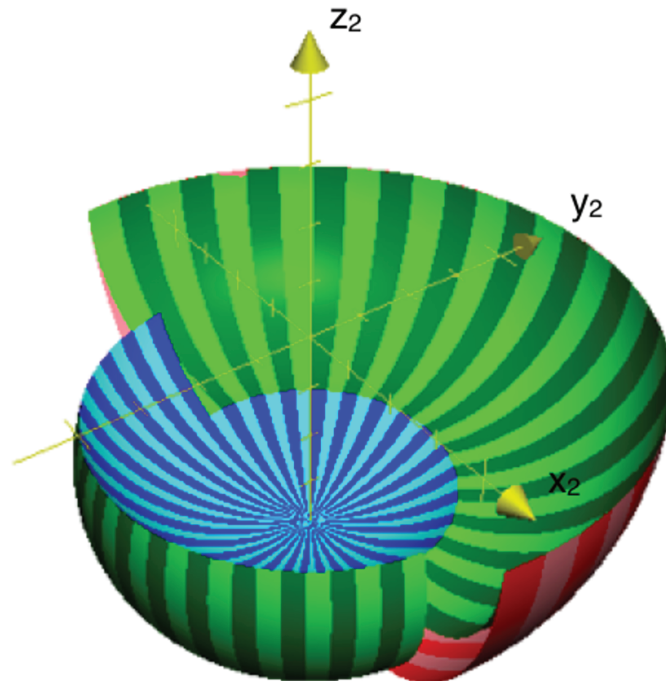

Figure 4.7: Six shells form the initial 3D knee-up workspace.

$$x^2 + y^2 = (L_1 + L_2)^2 \tag{4.24}$$

It should be noted that the equations for the arcs are based on a separate coordinate system specific to the plane containing each section.

4.5 General Knee Up Workspace Limitations

While the limb tip can reach any point within the workspace in the knee-up configuration, it is not necessarily possible to travel continuously from one point in the workspace to another. Figure 4.9 shows a top view of the workspace. The limb tip can travel continuously within the blue and green regions or the limb tip can travel continuously within the yellow and green regions. However, it is not possible for the limb tip to travel continuously from the blue region to the yellow region. Such a motion would require a

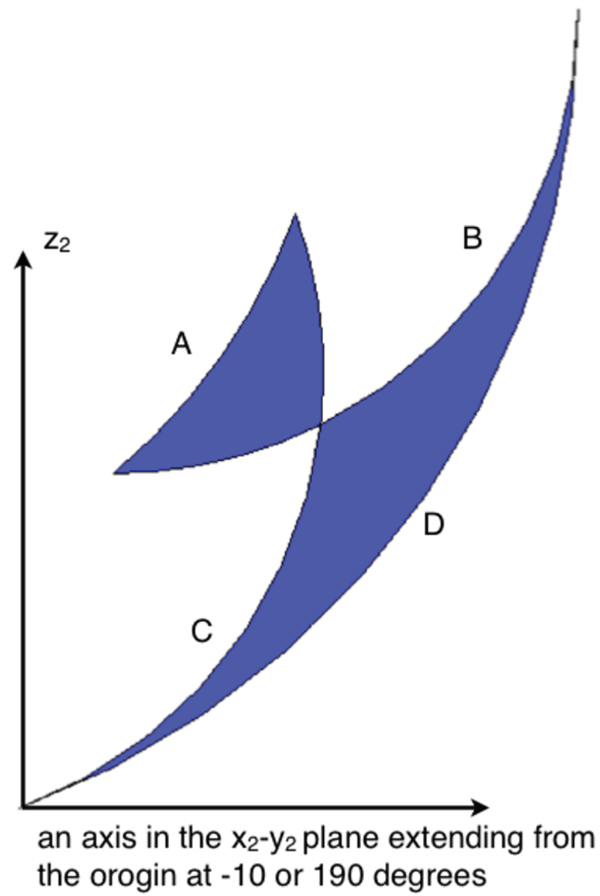

Figure 4.8: Planar sections which form part of the boundary for the workspace are encompassed by arc sections of circles.

rotation of 180 degrees of Revolute Joint 2 in the green region. Due to the complexity of incorporating this requirement into a continuously updating walking algorithm, the yellow region was removed from the workspace. As a result, only Shells 1, 3a, and 4a are used to bound the workspace. The workspace assumes the form in Figure 4.10, and the planar sections assume the form in Figure 4.11, where arcs A, B, D, and E are defined by Equations 4.21, 4.22, 4.24, and 4.25:

$$x = 0 \tag{4.25}$$

Again, the equations for the arcs are based on a separate coordinate system specific to the plane containing each section.

4.6 The Buffer Cylinder

While it is possible for the limb tip to reach a range of points on the z_2-axis, it is not necessarily possible for the limb tip to move near the z_2-axis. Limb tip motions near the z_2-axis which do not require the movement of Revolute Joint 2 are possible. For example: by freezing Revolute Joint 2, Revolute Joints 3 and 4 can still be used to trace a line from the z_2-axis radially out. Limb tip motions near the z_2-axis which require the movement of Revolute Joint 2 may not be possible due to the velocity limit of Revolute Joint 2. For example: tracing a line, at a finite speed, which passes infinitesimally close to the z_2-axis would require near-infinite rotational velocity of Revolute Joint 2 as it rotates nearly 180 degrees. The farther this traced line is from the z_2-axis, the slower the required rotational velocity of Revolute Joint 2 for a given tracing speed. Therefore, for a given walking speed and maximum rotational velocity of Revolute Joint 2, the required minimum distance a stride-line must pass from the z_2-axis can be calculate by:

$$r_B = \frac{U_t}{\dot{\theta}_2} \tag{4.26}$$

where r_B is the radius of the buffer cylinder and U_t is the component of the limb tip velocity tangent to the buffer cylinder and orthogonal to the z_2-axis. The rotational velocity $\dot{\theta}_2$ is plotted in Figure 4.12 for a limb tip velocity of 1 unit, perpendicular to a radius r given by:

$$r = L_1 cos\theta_3 + L_2 cos(\theta_3 + \theta_4) \tag{4.27}$$

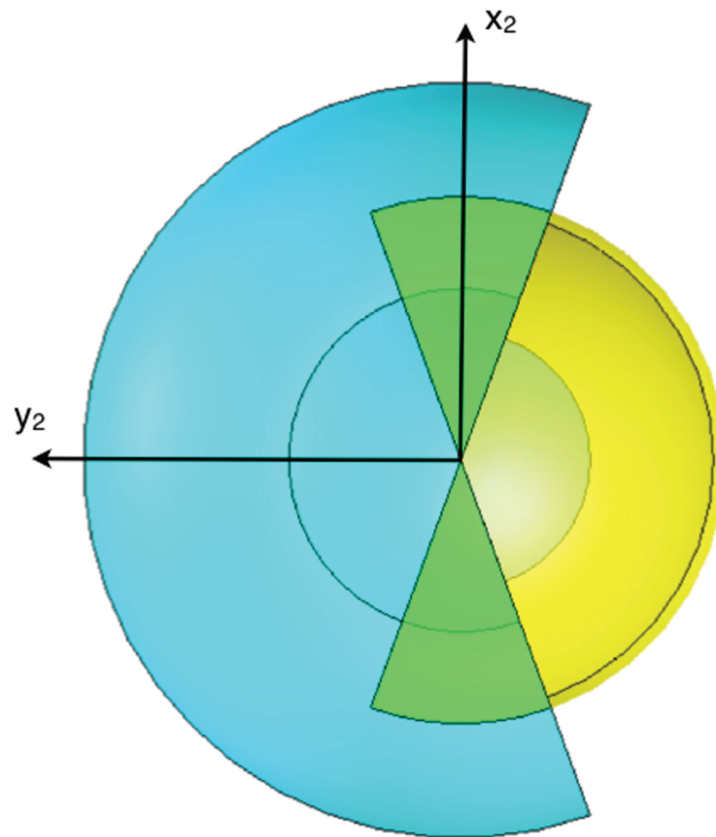

Figure 4.9: Movement of the limb tip from the blue region of the workspace to the yellow region requires an instantaneous 180 degree rotation of Revolute Joint 2.

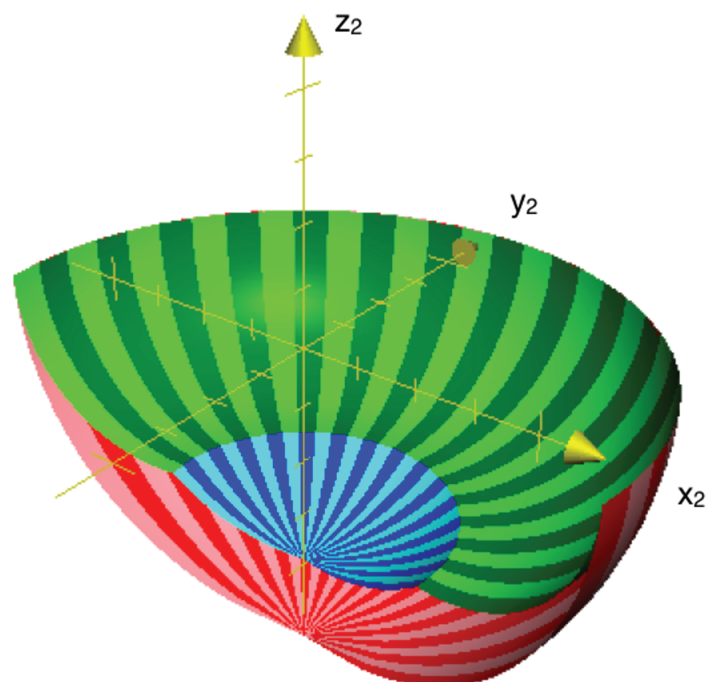

Figure 4.10: The workspace is limited due to constraints on continuous movement of the limb tip.

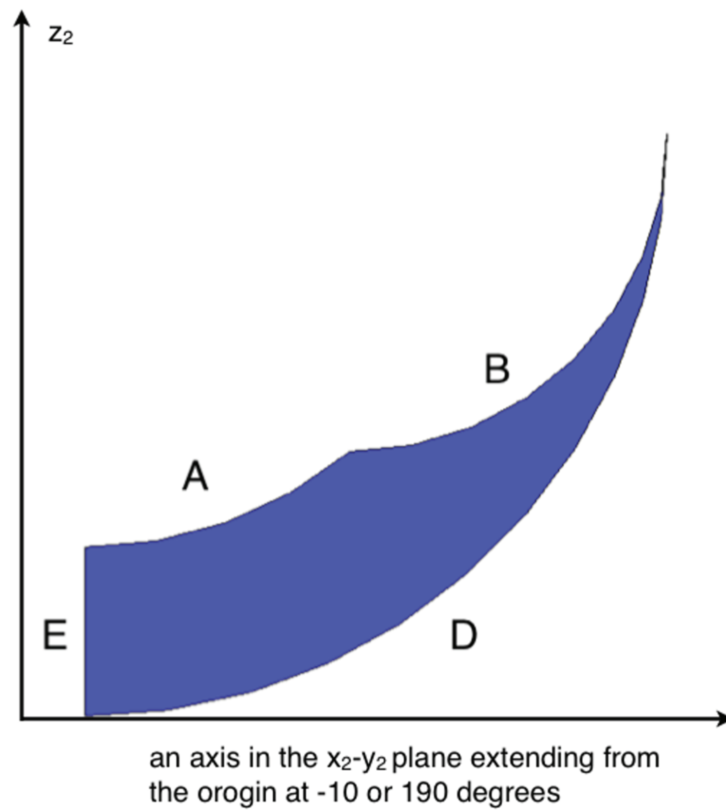

Figure 4.11: The workspace is limited due to constraints on continuous movement of the limb tip.

To deal with the existence of the z_2-axis singularity, a cylinder about the z_2-axis with radius r_B can be removed from the workspace. Because, $\dot{\theta}_2$ is inversely proportional to r_B, the cylinder is of constant radius along its length. Furthermore, because r is independent of θ_2, the cylinder has a circular cross-section in the x_2-y_2 plane. This cylinder is referred to as the "buffer cylinder." The buffer cylinder changes the workspace, as shown in Figure 4.13. Also the vertical boundary of the planar sections is moved away from the y-axis by the radius of the buffer cylinder, such that Equation 4.25 becomes:

$$x = r_B \tag{4.28}$$

where r_B is the radius of the buffer cylinder.

4.7 MARS Specific Workspace Limitations

Limb arrangement on the body determines further limitations on the work space. Because the walking algorithm requires a limb switch when the first contact limb reaches its workspace boundary, the work space of the other two contact limbs is essentially limited by the workspace of the one contact limb. In other words, for a given stride, the longest stride all three contact limbs can make is limited to the shortest of the three individual stride-lines. This concept is illustrated in Figure 4.14. Notice that though Limbs 3 and 5 have not reached the boundary of their respective workspace, they are cut short by limb 1, which has reached its workspace boundary. Because all strides are limited to the shortest stride, the usable workspace of all three contact limbs is limited. The resulting workspace is found by overlaying the three contact limb workspaces as shown in Figure 4.15 for 2D. The workspaces are overlain so that the largest common workspace will result. The largest common workspace roughly resembles a circle. For ease of programming, the common workspace was limited to the circle shown in Figure 4.15.

4.8 The 2D Common Workspace

A 2D representation of the workspace is sufficient for walking level. When the body of MARS is parallel to the walking surface then the walking surface is parallel to the x_2-y_2 plane of each limb. While the x_2-y_2 planes for the

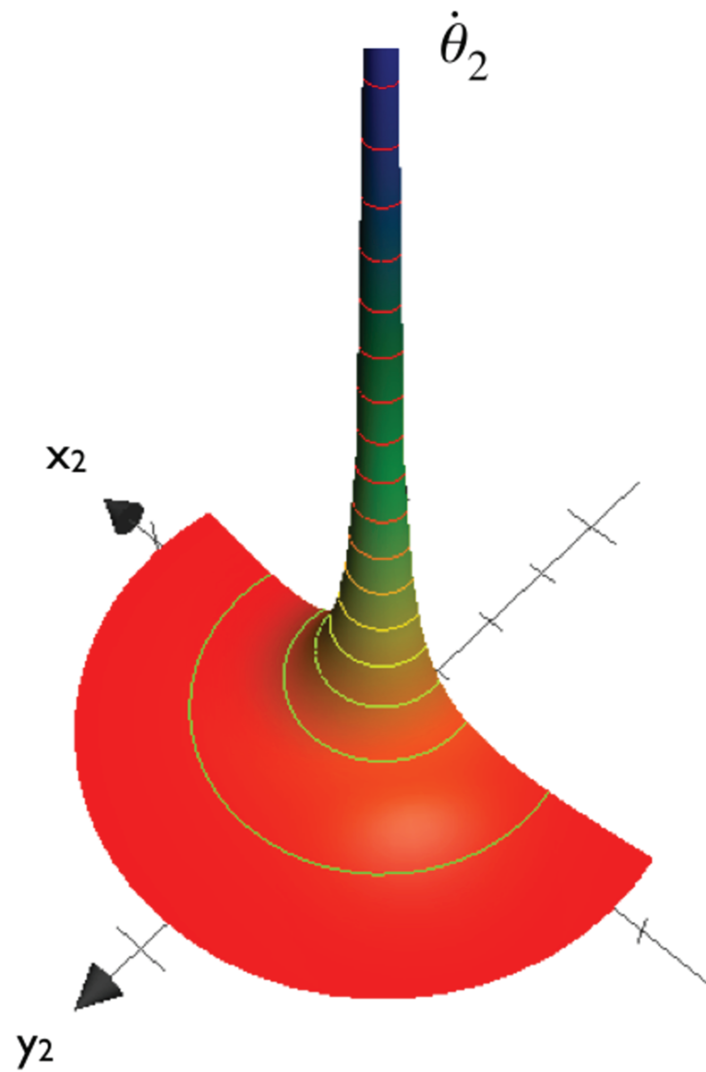

Figure 4.12: The required rotational velocity of Revolute Joint 2 goes to infinity as the limb tip path approaches the z_2 axis.

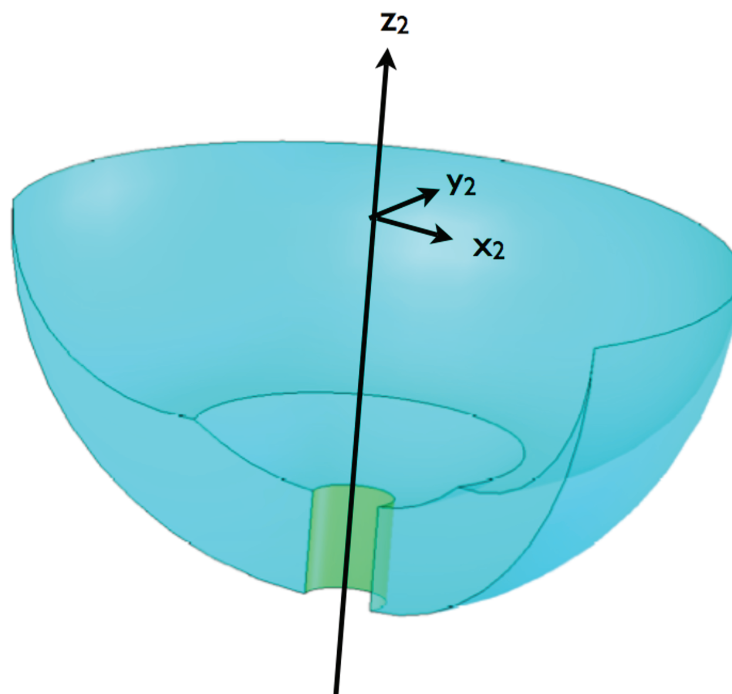

Figure 4.13: The buffer cylinder, about the z_2-axis, further limits the workspace.

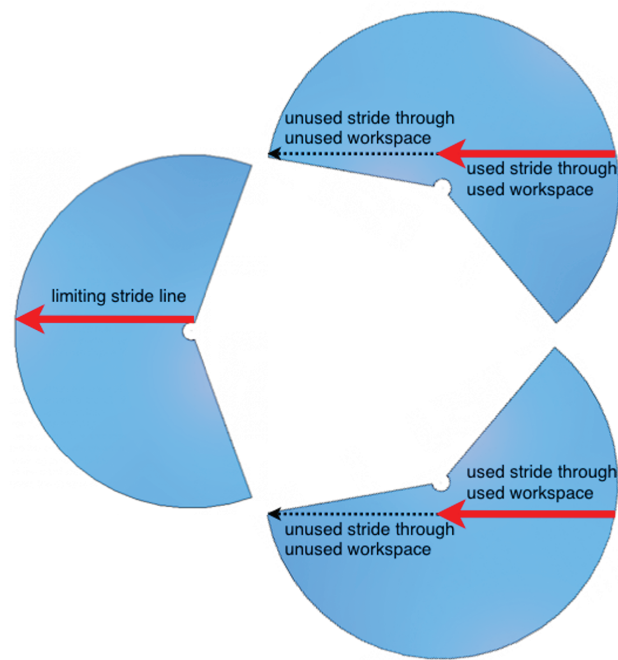

Figure 4.14: All three contact limb stride-lines are limited by the shortest stride-line.

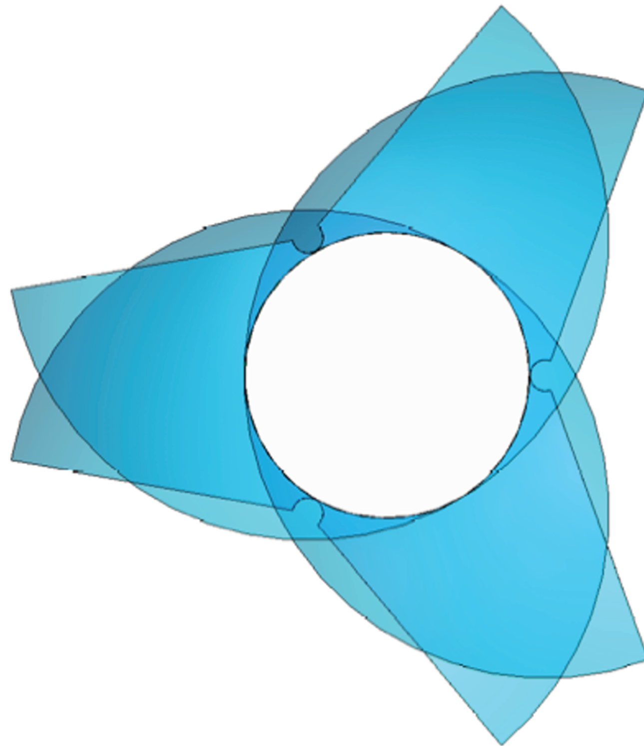

Figure 4.15: Overlaying the workspaces of the three contact limbs reveals the common workspace in the style of a Venn diagram. The circle is a simplification of the common workspace.

limbs are parallel to the walking surface, only a 2D slice of the workspace needs to be considered for the walking algorithm. For this condition the workspace for each limb will be a circle. The piecewise Equation 4.29 defines the circle diameters:

$$D = \begin{cases} D_1 & \text{if} \quad \text{Condition}_1 \\ D_2 & \text{if} \quad \text{Condition}_2 \\ D_3 & \text{if} \quad \text{Condition}_3 \end{cases} \tag{4.29}$$

where

$$D_1 = \sqrt{\left((L_1 + L_2)^2 - z_2^2\right)} - \left(\sqrt{L_2^2 - z_2^2} + L_1\right) \tag{4.30}$$

$$\text{Condition}_1 = 0 \geq z_2 > -L_2 \tag{4.31}$$

$$D_2 = \sqrt{\left((L_1 + L_2)^2 - z_2^2\right)} - \sqrt{L_1^2 + L_2^2 - z^2} \tag{4.32}$$

$$\text{Condition}_2 = -L_2 \geq z_2 > -\sqrt{L_1^2 + L_2^2 - r_B} \tag{4.33}$$

$$D_3 = \sqrt{\left((L_1 + L_2)^2 - z_2^2\right)} - r_B^2 \tag{4.34}$$

$$\text{Condition}_3 = -\sqrt{L_1^2 + L_2^2 - r_B} \geq z_2 \geq -\sqrt{(L_1 + L_2)^2 - r_B^2} \tag{4.35}$$

where L_1 and L_2 are the lengths of the proximal and distal limb sections, respectively; D is diameter of the circular workspace; and r_B is the radius of the buffer cylinder. The location of the center of the circle is given by the piecewise Equation 4.36 :

$$D = \begin{cases} \text{Center}_1 & \text{if} \quad \text{Condition}_1 \\ \text{Center}_2 & \text{if} \quad \text{Condition}_2 \\ \text{Center}_3 & \text{if} \quad \text{Condition}_3 \end{cases} \tag{4.36}$$

where

$$\text{Center}_1 = \frac{\sqrt{\left((L_1 + L_2)^2 - z_2^2\right)} + \left(\sqrt{L_2^2 - z_2^2} + L_1\right)}{2} \tag{4.37}$$

$$\text{Center}_2 = \frac{\sqrt{\left((L_1 + L_2)^2 - z_2^2\right)} + \sqrt{L_1^2 + L_2^2 - z^2}}{2} \tag{4.38}$$

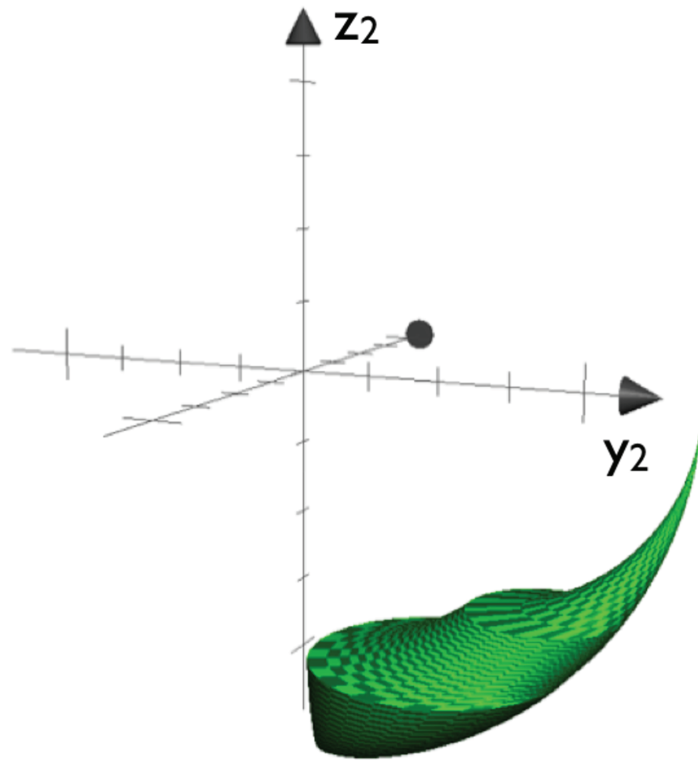

Figure 4.16: A 3D volume contains the set of 2D circular workspaces.

$$\text{Center}_3 = \frac{\sqrt{\left((L_1 + L_2)^2 - z_2^2\right)} + r_B^2}{2} \tag{4.39}$$

where the center point is located on the y_2-z_2 plane. The 3D workspace for each limb, provided the robot is walking parallel to the walking surface, is shown in Figure 4.16.

At this point it is possible to generate a walking algorithm which uses this workspace to walk parallel to the walking surface. Due to the limited size of the workspace, there is no risk of the limbs colliding with each other while walking. With proximal and distal section lengths of 5 and 6 units, respectively, the robot can theoretically walk at any height between 0 and 11 units. However, the dimensions of the limbs limit this to roughly 2 to 11 units.

4.9 3D Common Workspaces

Walking with the robot body not parallel to the walking surface complicates the use of the 2D common workspace. If 2D slices of the workspace are used, as with level walking, the slices could be of different shapes and sizes for each of the three contact limbs. With this approach the three 2D shapes would need to be calculated for each limb, for each iteration of the walking algorithm. However, the same results can be achieved by finding points of intersection of the stride-line with the workspace boundary. This approach is not examined in this thesis. However, two other approaches are examined:

- Mathematically define the workspace, as with the shell method. These definitions could then be used to find the intersection points of stride-lines with the workspace boundary.
- Select a spherical workspace which fits within an overlay of the 3D workspaces of all three contact limbs.

The 3D volume–composed of 2D circular workspaces shown in Figure 4.16–can be mathematically defined by dividing it into three sections as specified by $Condition_1$, $Condition_2$, and $Condition_3$ of Equations 4.31, 4.33, and 4.35. For $Condition_1$ the workspace is defined by Equation 4.40:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} Diameter_{Condition_1} + Center_{Condition_1} \cos\theta \\ Center_{Condition_1} \sin\theta \\ u \end{bmatrix} \tag{4.40}$$

where

$$Diameter_{Condition_1} = \left(\frac{\sqrt{(L_1 + L_2)^2 - u^2} + \sqrt{L_2^2 - u^2} + L_1}{2} \right) \tag{4.41}$$

$$Center_{Condition_1} = \left(\frac{\sqrt{(L_1 + L_2)^2 - u^2} - \sqrt{L_2^2 - u^2} + L_1}{2} \right) \tag{4.42}$$

$$u = -L_2...0, \theta = 0...2\pi \tag{4.43}$$

For $Condition_2$ the workspace is defined by Equation 4.44:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} Diameter_{Condition_2} + Center_{Condition_2} \cos\theta \\ Center_{Condition_2} \sin\theta \\ u \end{bmatrix} \tag{4.44}$$

where

$$Diameter_{Condition_2} = \left(\frac{\sqrt{(L_1 + L_2)^2 - u^2} + \sqrt{L_1^2 + L_2^2 - u^2} + L_1}{2} \right) \tag{4.45}$$

$$Center_{Condition_2} = \left(\frac{\sqrt{(L_1 + L_2)^2 - u^2} - \sqrt{L_1^2 + L_2^2 - u^2} + L_1}{2} \right) \tag{4.46}$$

$$u = -\sqrt{L_1^2 + L_2^2 - r_B^2} ... - L_2, \theta = 0...2\pi \tag{4.47}$$

For $Condition_3$ the workspace is defined by Equation 4.48:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} Diameter_{Condition_3} + Center_{Condition_3} \cos\theta \\ Center_{Condition_3} \sin\theta \\ u \end{bmatrix} \tag{4.48}$$

where

$$Diameter_{Condition_3} = \left(\frac{\sqrt{(L_1 + L_2)^2 - u^2} + r_B}{2} \right) \tag{4.49}$$

$$Center_{Condition_3} = \left(\frac{\sqrt{(L_1 + L_2)^2 - u^2} - r_B}{2} \right) \tag{4.50}$$

$$u = -\sqrt{(L_1 + L_2)^2 - r_B^2} ... - \sqrt{L_1^2 + L_2^2 - r_B^2} - L_2, \theta = 0...2\pi \tag{4.51}$$

Using the spherical workspace as the basis for a walking algorithm requires that the sphere be mathematically defined. The sphere is defined, as shown in Figure 4.17, as tangent to Shell 3a, Shell 4a, and centered on the y_2-z_2 plane. These constraints for the spherical workspace define its size, but there is still a range of locations for the workspace: tangent to Shell 1 (farthest

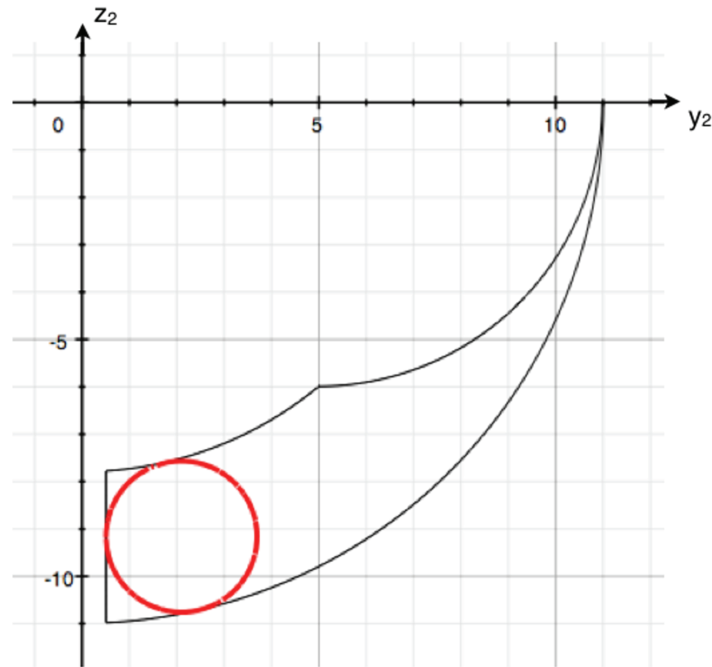

Figure 4.17: A spherical workspace simplifies the walking algorithm while allowing robot body roll and pitch.

possible from the body), tangent to the buffer cylinder (closest to the body), and on a continuum between these two extremes. The condition, tangent to the buffer cylinder, was selected to reduce motor torques. A sphere in this location is mathematically defined by Equation 4.52:

$$x_2^2 + (y_2 - (r_B + r_c))^2 + \left(z_2 + \sqrt{((L_1 + L_2) - r_c)^2 - (r_B + r_c)^2}\right)^2 = r_c^2 \tag{4.52}$$

where

$$r_c = \frac{(L_1 + L_2) - \sqrt{L_1^2 + L_2^2}}{2} \tag{4.53}$$

Chapter 5

Walking Algorithms

A range of workspaces are possible with MARS limbs. However, for walking, there is a balance between the size of the workspace and the simplicity of the adaptive walking algorithm. Specifically, a workspace with very simple geometry results in a computationally simpler walking algorithm. Conversely, a workspace with very complex geometry requires more computation to define a stride-line across the workspace. Walking algorithms presented in this thesis use both 2D and 3D workspaces. Generally speaking the larger workspaces are geometrically complex; however, the larger workspaces can be geometrically simplified by only using a simple geometry within the large workspace.

The walking algorithms presented in this paper meet several criteria which serve to limit the possible ways they could be achieved. These criteria also define desirable attributes which the developed walking algorithms possess. The main focus in developing the walking algorithms was to make use of as much functionality as possible. The MARS limbs are capable of continous motion within a 3D workspace, therefore the robot should be capable of walking in any direction, within a range of heights, within a range of speeds, and within a range of roll, pitch, and yaw. Further, the actuators are capable of operating at a range of speeds from max to still, therefore the robot should be able to continuously change its motion in response to input. Because the development of this algorithm is part of a foundation of research for MARS, it was decided to limit the algorithm to walking on planar surfaces for the sake of simplicity. One of the most significant decisions in the development of these walking algorithms was to have them execute in an iterative digital fashion. There are several other approaches, such as

analog control, or a neural network operating either as analog or digitall but not sinked. However, digital iteration was chosen as a conceptually simple basis for the algorithm, and also because the approach worked well with the Dynamixel actuators. These actuators are designed to operate with daisy-chained control wiring. Each actuator is assigned a specific ID. The position commands for all actuators are given, each command tied to an ID. Then a go command is given, and all actuators move to the next position. Wile such an iterative walking algorithm is not truly continuos, the effect is satisfactory with high iteration rates. As the iterations cycle the algorithms appear to provide seamless instantaneous response to changes in direction, speed, and any other allowed input.

5.1 The Abstracted Walking Algorithm

The complexity of how six limbs move together to form a walking gait can be reduced by abstracting the problem to one limb. As each limb performs the same role while walking, namely taking a step, a walking algorithm for any number of limbs can be constructed by simply applying the same algorithm to each limb. Walking is achieved by making some of the inputs to such an algorithm dependent on the limb location. For example: from the standpoint of the limb coordinate system, the direction that the robot is walking could be the positive y-direction for one limb and the negative y-direction for a limb on the opposite side of the body. Also, for statically stable gaits, the limbs are separated into two groups which act together. One group is in contact with the walking surface, while the other is not.

This chapter explains the algorithms developed from this abstracted approach. The two walking algorithms discussed in this chapter use the tripodal statically-stable gait, much like an insect. Given the two possible walking limb states of ("contact" and "non-contact"), as well as an even number of limbs, any limb is in the state different from the limbs next to it.

5.2 The General Walking Algorithm

The general walking algorithm is based on the general workspace. This algorithem can be used for any robot with six or more limbs kinematically similar to MARS limbs. The limbs can be attached to the robot body at any

point and orientation, provided all the limbs can reach the walking surface simultaneously. The limbs can be dimentionaly dissimilar from each other and the robot body can be of any shape, provided that a statically-stable gait is possible. furthermore, this algorithem can except as inputs:

- direction
- speed
- walking height
- roll
- pitch
- yaw

In this algorithm an entire step is not planned at once. Rather, only a section of the step is planned for each iteration. For each iteration, a limb tip is moved from its current point in space to a new point along what will be called the "stride-line." Over the course of several iterations, the limb tip moves along the path of the step. Though each stride-line is straight, the direction of successive stride lines can be continuously adjusted throughout the step motion, resulting in a curved step path. To do this, the walking algorithm must compute intersection points of the stride-line with the shells which make up the workspace boundary. Note that finding the equation of the stride-line is only necessary in order to solve for intersection points with the workspace boundary shells. Otherwise, it would only be necessary to define the next limb tip position by adding the limb tip direction vector to the current limb tip position. Figure 5.1 shows the stride-lines and limb tip positions associated with a curved step path through a circular 2D workspace.

5.2.1 Defining the Next Limb Tip Location

The basic components of the general walking algorithm are shown in Figure 5.2. For each iteration, a new stride-line, dependent on the direction and orientation of the robot body, must be found. The next limb tip location will be on this stride-line. However, the method used to find the stride-line will depend on whether the limb tip is in contact or not. The method for defining each follows:

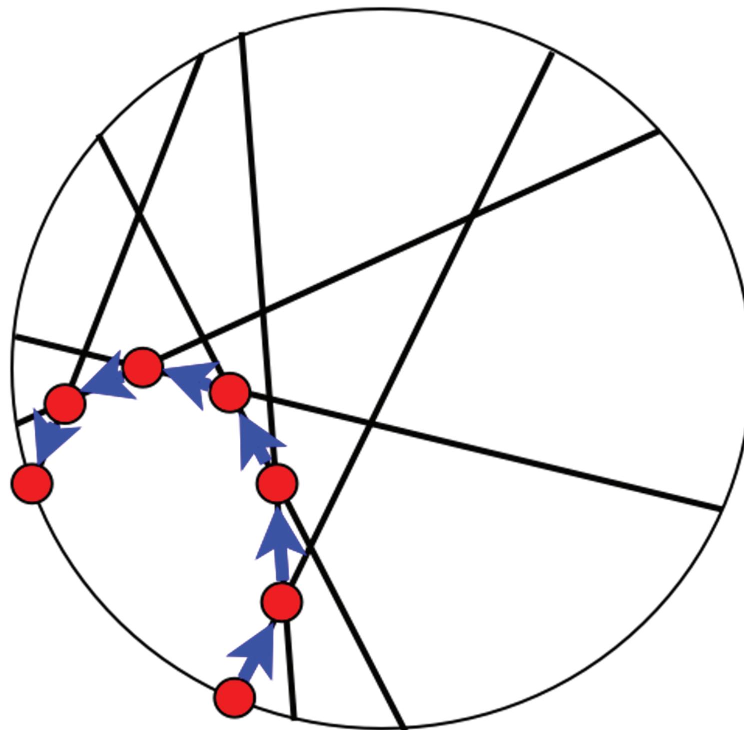

Figure 5.1: Iteratively generated stride-lines can form curved step paths.

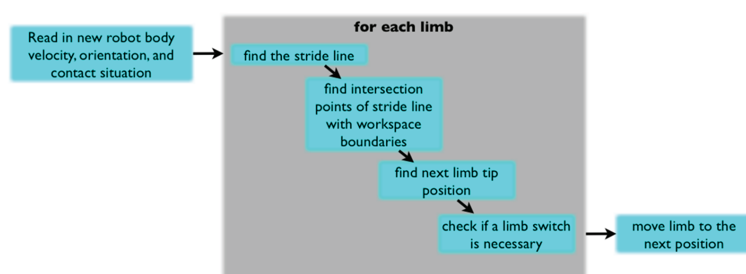

Figure 5.2: One iteration of the basic walking algorithm.

For the contact limb, the stride-line passes through the current limb tip position and is parallel to the velocity vector of the robot body. The direction of the velocity vector of the limb tip is opposite the direction of the velocity vector of the robot body. The next limb tip position is found by adding the limb tip velocity vector to the current limb tip position.

For non-contact limbs, the process for finding the stride-line is more involved. If the robot were directed to walk with the current velocity, the stride-lines could be optimized for long strides. This somewhat arbitrary long stride optimization does not necessarily improve the walking ability, but does provide for a more evenly timed alternating gate and reduces the number of contact/non-contact limb switches. Optimization cannot be effected mid-stride for the contact limbs, because they are fully constrained by the robot body velocity and their current contact point. However, in this algorithm both contact and non-contact stride-lines would ideally meet the form in Figure 5.3. The depicted stride-line is tangent to the buffer cylinder. This stride-line situation ensures a long stride-line and reasonable body stability. Most importantly, however, it is easy to define by the constraints:

- parallel to the body velocity vector
- tangent to the z-axis buffer cylinder at a given height

This walking algorithm is not designed to predict the future robot direction, speed, height, and orientation specified by the operator. Rather, the algorithm assumes that the current inputs will remain constant, and the algorithm attempts to optimize for them by moving the non-contact limbs to the starting point of their future contact stride-lines. In practice, this method works well because the non-contact limbs position faster than the contact limbs and because of the high iteration rate of the algorithm. However, for the non-contact limbs to move to an optimal position in anticipation of the limbs' switching, first a "pre-stride-line" and then a stride-line must be found. The pre-stride-line is essentially the predicted future contact stride-line, elevated from the walking surface to the non-contact height. The pre-stride-line is defined using the following constraints:

- parallel to the body velocity vector
- tangent to the z-axis buffer cylinder
- a set distance above the walking surface

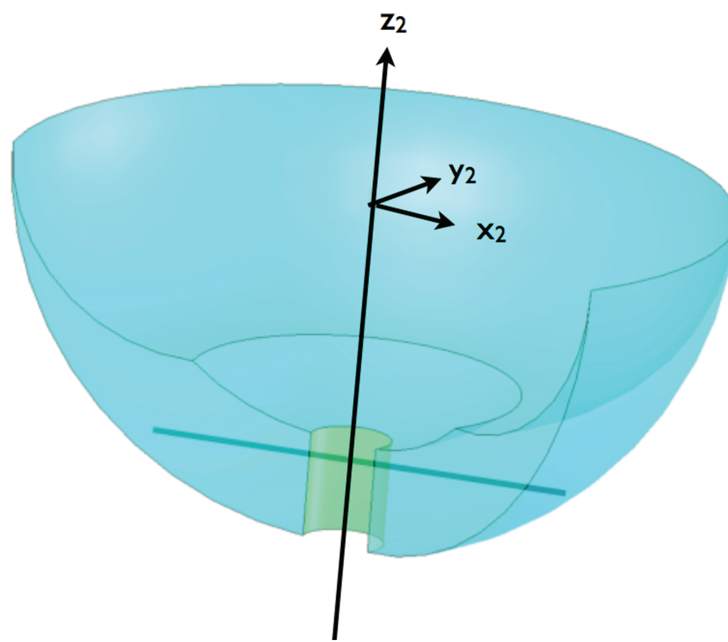

Figure 5.3: The Ideal stride-lines make a long path through the workspace, avoiding the buffer cylinder.

The immediate purpose of defining the pre-stride-line is to identify the forward point where the pre-stride-line intersects the workspace boundary–the forward endpoint. Having defined this pre-stride-line, its forward endpoint is then used to help define the non-contact stride-line–the actual line used to help specify the next non-contact limb tip position. The geometry of finding the next non-contact limb tip location is shown in Figure 5.4. A vector, $\vec{a}$, is constructed, originating at the current non-contact limb position and pointing toward the pre-stride-line forward endpoint. The magnitude of this vector is twice the magnitude of the body velocity vector. The doubled magnitude allows the non-contact limbs to reach the pre-stride-line endpoint before a contact/non-contact limb switch is necessary. However, if the distance from the current non-contact limb position to the pre-stride-line forward endpoint is less than $\vec{a}$, the remaining steps are skipped, and the next non-contact limb tip position is defined as the pre-stride-line forward endpoint. A second vector, $\vec{b}$, is formed, originating at the current non-contact limb tip position and pointing normal to the walking surface. The magnitude of $\vec{b}$ is equal to a percentage (e.g. 90%) of the difference between H and h, where:

- H is the length of a vector normal to the walking surface which stretches from the walking surface to any point on the pre-stride-line
- h is the length of a vector normal to the walking surface which stretches from the walking surface to the current non-contact limb tip position

The sum of $\vec{a}$ and $\vec{b}$ added to the current non-contact limb tip position gives the next non-contact limb tip position.

5.2.2 Buffer Cylinder Tangency

As discussed in Section 5.2.1, pre-stride-lines are defined as tangent to the buffer cylinder. When the limb tip is not in contact with the surface, the direction of the stride-line and stride height are the only inputs for finding the stride-line. These constraints are not sufficient to fully define the line. As the stride-line cannot intersect the z-axis singularity buffer cylinder, it is set to be tangent to this cylinder at a point, thus sufficiently defining the line. The point used is the tangent point of intersection between the stride-line and the z-axis singularity buffer cylinder. If this point is outside the 3D workspace, the point of orthogonal intersection between the height-line and

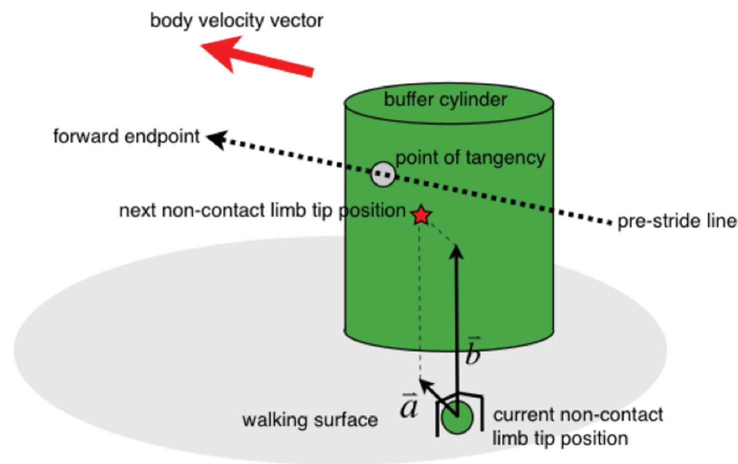

Figure 5.4: The next non-contact limb tip location is selected so as to optimize the gait for the current operator input.

the stride-line is found. Figure 5.5 displays the geometry of the non-contact stride-line.

We can find the stride-line buffer-cylinder tangent intersection point, p, given the direction of the stride-line (u_{xp}, u_{yp}, u_{zp}) and the stride-height h. First we find b:

$$h^2 = r^2 + b^2 \tag{5.1}$$

$$b = \sqrt{h^2 - r^2} \tag{5.2}$$

Next we find the angle θ:

$$\theta = \arctan\left(\sqrt{u_{px}^2 + u_{py}^2}, u_{pz}\right) \tag{5.3}$$

Then the law of sins is used to find z_p:

$$z_p = \frac{b}{\sin\theta} \tag{5.4}$$

This method is demonstrated in a MATLAB m-file in Appendix A.19.

If the point (x_p, y_p, z_p) is outside the 3D workspace, the point where the height-line orthogonally intersects the stride-line can be used in the algorithm. This point (x_q, y_q, z_q) can be found by:

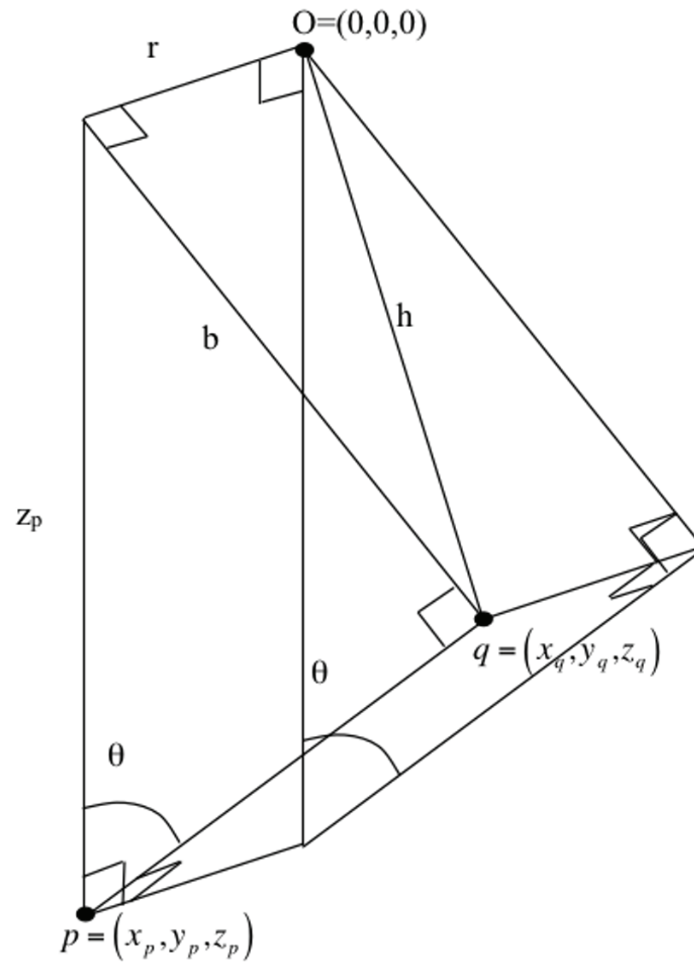

Figure 5.5: A point tangent to the buffer cylinder, p, is used to help define non-contact stride-lines and pre-stride-lines; however, if this point is outside the workspace, the orthogonal intersection point between the height-line and stride-line is used.

$$\vec{q} = \vec{r} + \vec{b} \tag{5.5}$$

where

$$\vec{r} = r\left(\frac{u_{py}}{\sqrt{-u_{px}^2 + u_{py}^2}}, \frac{-u_{py}}{\sqrt{-u_{px}^2 + u_{py}^2}}, 0\right) \tag{5.6}$$

$$\vec{b} = \left[b\cos\theta\left(\frac{-u_{ry}}{\sqrt{-u_{ry}^2 + u_{rx}^2}}, \frac{u_{ry}}{\sqrt{-u_{ry}^2 + u_{rx}^2}}\right), -b\sin\theta\right] \tag{5.7}$$

This method is demonstrated in a MATLAB m-file in Appendix A.20.

5.2.3 Contact/Non-Contact Limb Switch

If any contact limb "nears" the end of its workspace, all limbs switch their "contact/non-contact state." In this algorithm, "near" is defined as less than twice the distance travelled in one iteration. The contact/non-contact limb-switch requires two iterations:

Iteration 1

- The next point for the non-contact limbs is defined as the forward endpoint of a contact stride-line passing through the point directly below the current non-contact limb tip position. (The non-contact limbs "puts their tips down.")
- The contact limbs and non-contact limbs are moved to the next point. At this instant all limbs are in contact.

Iteration 2

- The original contact limbs become non-contact limbs, and their next limb tip position is found using the non-contact limb tip position algorithm. (The contact limbs "lift their tips.")
- The original non-contact limbs (now in contact) are moved to their new contact limb position.

5.2.4 Stride-Line Workspace Intersection

As mentioned in earlier, the workspace is constructed of portions of toruses, spheres, planes, and a cylinder. Thus finding stride-line intersections with the workspace boundary begins with finding stride-line intersections with each geometry. Next the algorithm selects the intersection points which lie on the shell, or boundary, discarding any not on the shell. Finally, the two intersection points closest to the "given point" (defined below) are designated as the endpoints of the stride-line (or the pre-stride line). The given point for contact limbs is the current limb tip position. The given point for non-contact limb pre-stride-lines is the point tangent to the buffer cylinder. After the end points are found, the points are screened to determine if they lie on the section of the base geometry which is used in the knee-up workspace. If the points pass the screening, their directional relation to the given point is used to designate the forward and rear endpoints in accordance with the body direction vector.

While there are many steps to finding the next limb tip position, they mostly deal with sorting and defining intersection points between the stride-line and the geometries which form the workspace. therefore the foundation of the walking algorithm can be viewed as the functions which find these intersection points. These functions are summarized below.

Stride-Line Sphere Intersection

The intersection of the stride-line with Shells 3a, 4a, and 4b require finding the points of intersection of a line and a sphere, as shown in Figure 5.6. This algorithm [28], specific to this situation, is given bellow:

First find the magnitude of the line OC given the $\left|\vec{OC}\right|$, as shown in Equation 5.9:

$$\begin{bmatrix} x_{OC} \\ y_{OC} \\ z_{OC} \end{bmatrix} = \begin{bmatrix} x_C \\ y_C \\ z_C \end{bmatrix} - \begin{bmatrix} x_R \\ y_R \\ z_R \end{bmatrix} \tag{5.8}$$

$$\left|\vec{OC}\right| = \sqrt{x_{OC}^2 + y_{OC}^2 + z_{OC}^2} \tag{5.9}$$

where the subscript $_{OC}$ denotes the vector $\vec{OC}$, the subscript C denotes the center point of the sphere, and the subscript R denotes the point position

of the limb tip. Assuming that the point R_0 is inside the sphere, that is $\left|\vec{OC}\right| > r$ is false, intersection points are found as follows:

$$L = \vec{OC} \cdot [u_x, u_y, u_z] \tag{5.10}$$

$$D^2 = \left|\vec{OC}\right|^2 = L^2 \tag{5.11}$$

$$HC^2 = r_s^2 - D^2 \tag{5.12}$$

$$t_i = L + HC \tag{5.13}$$

where L is the perpendicular distance from the limb tip to the center of the sphere, the vector u is the direction of the stride-line, D is the shortest distance from the center of the sphere to the stride-line, HC is the distance from an intersection point to the center of the sphere projected along the stride-line, r_s is the radius of the sphere, and t_i is the distance from the limb tip to an intersection point. Because the square root of HC yields both a positive and negative value, t_i yields two values, one for each intersection point. These intersection points can then be found as shown in Equation 5.14:

$$(x_i, y_i, z_i) = R_0 + t_i R_d \tag{5.14}$$

However, if the point R_0 is outside the sphere, the line may not intersect the sphere at all. If $HC^2 < 0$, the line does not intersect the sphere at any point. If $HC^2 = 0$ the line is tangent to the sphere and therefore intersects at one point. For this situation t_i is simply equal to L.

This function can be found as a MATLAB m-file in Appendix A.28.

Stride-Line Torus Intersection

The method used to find the points of intersection of a line with a torus shell is as follows: The parametric equation for a torus was algebraically manipulated using trigonometric identities to form an expression in terms of x, y, z, r, and α [29]. The parametric equation for a line was substituted into this expression to form a polynomial in terms of t. The roots of this polynomial are then used to find the points of intersection. The method of

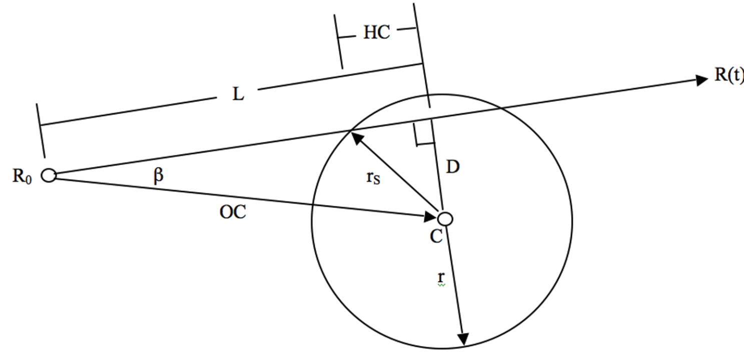

Figure 5.6: The intersection points of a stride-line with spherical Shells 3a, 4a, and 4b are found using this geometric approach.

manipulating and combining the equations for the line and torus to find the roots of t follows:

Parametric equation of a torus:

$$\begin{aligned} x &= (r + \alpha \cos v) \cos u \\ y &= (r + \alpha \cos v) \sin u \\ z &= \alpha \sin v \end{aligned} \tag{5.15}$$

where r is the radius from the center of the hole to the center of the torus tube, and α is the radius of the tube.

Parametric equation of a line:

$$\begin{aligned} x &= x_1 + tu_x \\ y &= y_1 + tu_y \\ z &= z_1 + tu_z \end{aligned} \tag{5.16}$$

where (x_1, y_1, z_1) are a point on the line, t is any real number, and (u_x, u_y, u_z) is the unit vector direction of the line.

By squaring the x and y terms of the parametric equation of the torus we have:

$$x^2 = (r + \alpha \cos v)^2 \cos^2 u \tag{5.17}$$

$$y^2 = (r + \alpha \cos v)^2 \sin^2 u \tag{5.18}$$

Adding these squared terms gives:

$$x^2 + y^2 = (r + \alpha \cos v)^2 \left(\cos^2 u + \sin^2 u\right) \tag{5.19}$$

$$x^2 + y^2 = (r + \alpha \cos v)^2 \tag{5.20}$$

Including the squared z term:

$$z^2 = (\alpha \sin v)^2 = \alpha^2 \sin^2 v \tag{5.21}$$

gives:

$$x^2 + y^2 + z^2 = (r + \alpha \cos v)^2 + \alpha^2 \sin^2 v \tag{5.22}$$

$$x^2 + y^2 + z^2 = r^2 + 2r\alpha \cos v + \alpha^2 \cos^2 v + \alpha^2 \sin^2 v \tag{5.23}$$

$$x^2 + y^2 + z^2 = r^2 + 2r\alpha \cos v + \alpha^2 \left(\cos^2 v + \sin^2 v\right) \tag{5.24}$$

$$x^2 + y^2 + z^2 = r^2 + 2r\alpha \cos v + \alpha^2 \tag{5.25}$$

By subtracting r^2 and α^2 from both sides and then dividing both sides by $2r\alpha$ the cos term is isolated:

$$x^2 + y^2 + z^2 - r^2 - \alpha^2 = 2r\alpha \cos v \tag{5.26}$$

$$\frac{x^2 + y^2 + z^2 - r^2 - \alpha^2}{2r\alpha} = \cos v \tag{5.27}$$

From the z term for the parametric equation of a torus we have:

$$z = \alpha \sin v \tag{5.28}$$

$$sinv = \frac{z}{\alpha} \tag{5.29}$$

Using this relation the cos term can be eliminated:

$$\left(\frac{x^2 + y^2 + z^2 = r^2 - \alpha^2}{2r\alpha}\right)^2 = \cos^2 v \tag{5.30}$$

$$\left(\frac{x^2 + y^2 + z^2 = r^2 - \alpha^2}{2r\alpha}\right)^2 + \sin^2 v = \cos^2 v + \sin^2 v \tag{5.31}$$

$$\left(\frac{x^2 + y^2 + z^2 = r^2 - \alpha^2}{2r\alpha}\right)^2 + \sin^2 v = 1 \tag{5.32}$$

$$\left(\frac{x^2+y^2+z^2=r^2-\alpha^2}{2r\alpha}\right)^2+\frac{z}{\alpha}=1 \tag{5.33}$$

Mathematica was then used to substitute in the (x, y, z) terms from the parametric equation for a line and re-order the expression as a polynomial in terms of t. This Mathematica script can be found in Appendix B. This can be rewritten as:

$$t^4\frac{a}{4f}+t^3\frac{b}{f}+t^2\frac{c}{2f}+t\frac{d}{f}+\frac{e}{4f}=o \tag{5.34}$$

where:

$$a=\left(u_x^2+u_y^2+u_z^2\right)^2 \tag{5.35}$$

$$b=\left(u_x^2+u_y^2+u_z^2\right)\left(u_x x_1+u_y y_1+u_z z_1\right) \tag{5.36}$$

$$\begin{aligned} c= \; & 4u_y u_z y_1 z_1+4u_x x_1\left(u_y y_1+u_z z_1\right)+ \\ & u_x^2\left(-x^2-\alpha^2+3x_1^2+y_1^2+z_1^2\right)+ \\ & u_y^2\left(-x^2-\alpha^2+x_1^2+3y_1^2+z_1^2\right)+ \\ & u_z^2\left(x^2-\alpha^2+x_1^2+y_1^2+3z_1^2\right) \end{aligned} \tag{5.37}$$

$$\begin{aligned} d= \; & u_x x_1\left(-x^2-\alpha^2+x_1^2+y_1^2+z_1^2\right)+ \\ & u_y y_1\left(-x^2-\alpha^2+x_1^2+y_1^2+z_1^2\right)+ \\ & u_z z_1\left(x^2-alpha^2+x_1^2+y_1^2+z_1^2\right)+ \end{aligned} \tag{5.38}$$

$$\begin{aligned} e= \; & x_1^4+y_1^4-2y_1^2\left(x^2+\alpha^2-z_1^2\right)- \\ & u_2 x_1^2\left(x^2+\alpha^2-y_1^2-z_1^2\right)+\left(x^2-\alpha^2+z_1^2\right)^2 \end{aligned} \tag{5.39}$$

$$f=x^2\alpha^2 \tag{5.40}$$

Once the roots are found, they are plugged back into the parametric equation for the line to solve for the points of intersection.

This function can be found as a MATLAB m-file in Appendix A.27.

Stride-Line Plane Intersection

The intersection point, between the stride-line and the sections of planes that partially bound the general workspace, is found using a special case of line-plane intersection. The plane will always contain the z_2-axis and therefore also the point $(0, 0, 0)$. The point of intersection is found by first finding the value t as shown in Equation 5.41:

$$t = \frac{-\left(xa + yb + zc\right)}{\begin{bmatrix} a & b & c \end{bmatrix} \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix}} \tag{5.41}$$

where (x, y, z) is a point on the line, (u_x, u_y, u_z) is the direction vector of the line, and (a, b, c) is a vector perpendicular to the plane. The value of t is then plugged back into the parametric equation for the line to find the intersection point $(x_{int}, y_{int}, z_{int})$:

$$\begin{aligned} x_{int} &= x + u_x t \\ y_{int} &= y + u_y t \\ z_{int} &= z + u_z t \end{aligned} \tag{5.42}$$

This method is demonstrated as a MATLAB m-file in Appendix A.29.

Stride-Line Buffer-Cylinder Intersection

The workspace is further bounded by a cylinder which surounds the z-axis. Due to the singularity along the z-axis, a buffer-cylinder is required to ensure that the motor velocities can be achieved. The points of intersection of the stride-line with this cylinder are found by first finding the intersection of the stride-line and a circle in 2D. Looking at the stride-line and the buffer cylinder in the x-y plane, intersection points are found between the line and the circle. These points are then used in the x and y components of the parametric equation of the line to find the value of t. The z value can then be found using t. Given a point on the line (x, y, z) and the direction vector of the line (u_x, u_y, u_z), some terms are defined:

$$d_r = \sqrt{u_x^2 + u_y^2} \tag{5.43}$$

$$D = x\left(y + u_y\right) - y\left(x + u_x\right) \tag{5.44}$$

Points of intersection are given by:

$$x_{int} = \frac{Dd_y \pm sgn\left(u_y\right) u_x \sqrt{r^2 d_r^2 - D^2}}{d_r^2} \tag{5.45}$$

$$y_{int} = \frac{-Dd_y \pm \left|u_y\right| \sqrt{r^2 d_r^2 - D^2}}{d_r^2} \tag{5.46}$$

where (x_{int}, y_{int}) is an intersection point, and sgn is defined as:

$$sgn\left(x\right) = \begin{cases} -1 & \text{for } x < 0 \\ 1 & \text{otherwise.} \end{cases} \tag{5.47}$$

Now, by solving the x or y component of the parametric equation of the line for t:

$$t = \frac{x_{int} - x}{u_x} \tag{5.48}$$

the z intersection points are found:

$$z_{int} = z + u_z t \tag{5.49}$$

This method for finding the points of intersection of the stride-line and buffer cylinder is demonstrated as a MATLAB m-file in Appendix A.18.

5.3 Attempted Implementation of the General Walking Algorithm

A set of functions was generated to demonstrate the feasibility of the general walking algorithm. This was implemented using a combination of LabVIEW and MATLAB to generate joint angles. A Mathematica script was then used to generate a simulation of MARS walking. These MATLAB m-files, Mathematica script, as well as a brief description of the LabVIEW virtual instrument (VI) are found in Appendix A.

Unfortunately this set of functions never produced satisfactory results. The set of MATLAB m-files combined with LabVIEW VI, was capable of generating sets of joint angles. However, when these angles were animated using the Mathematica script, the simulation would make a short step-like motion and then the limbs would quickly extend beyond the physical limits of MARS and make quick repetitive motions physically impossible for the robot. A description of the function set is included for the following reasons: while the complete program was never shown to produce satisfactory results; the Mathematica simulator as well as all but the top level m-files were shown to work fine; an understanding of how the algorithm was implemented may be useful for those continuing work on MARS; and the function set is the

most thorough recorded explanation of how the general walking algorithm could work.

The basic architecture of the general walking algorithm is shown in Appendix A. A LabVIEW VI is used to collect user control input and output joint angles. A VI was chosen instead of an m-file because of the ease of using input devices with LabVIEW as well as the fact that LabVIEW VI's had already been developed in RoMeLa for communicating with the Dynamixel actuators.

The VI runs two embedded MATLAB m-files within a timed loop. The first m-file, named Function Call in Figure A.1, serves to call a set of MATLAB functions which output the next limb positions. It should be noted that limbCONTROL could be used directly in place of Function Call. Function Call is used instead because limbCONTROL is a lengthy file which is difficult to change in the VI interface. In the event that a limb switch is necessary, Function Call outputs the next two sets of limb tip positions from limbCONTROL. If two sets of limb tip positions are called, the VI stores the second set. During the next iteration of the loop, the second set is used instead of running Function Call again. For each iteration of the VI loop, the imbedded m-file Inverse Kinematics converts the limb tip position for each limb to joint angles. The VI then saves the joint angles in a text file which can be read by Mathematica for the test simulation. If the test simulations had ever proved successful, the joint angles would then have been used as input to the existing VI used to control the Dynamixel motors. A more detailed description of the VI can be found in Appendix A.2. The m-file Function Call can be found in Appendix A.3. The m-file Inverse Kinematics can be found in Appendix A.4.

The m-file limbCONTROL simply calls a series of m-files which, given the end points of a stride-line, find the next limb tip position for each limb. These m-files are called in order, one after the other. The m-file names and functional description follow:

- totalTIPtranslation: This function finds the limb tip translation due to body rotation and translation.
- limbFRAMEdirection: This function translates the body cordinate direction vector (u_x, u_y, u_z) to each limb cordinate frame.
- findSTRIDEline: This function calls the functions MARScontactLINE and MARSnoncontactLINE to find the endponts of stride-lines.

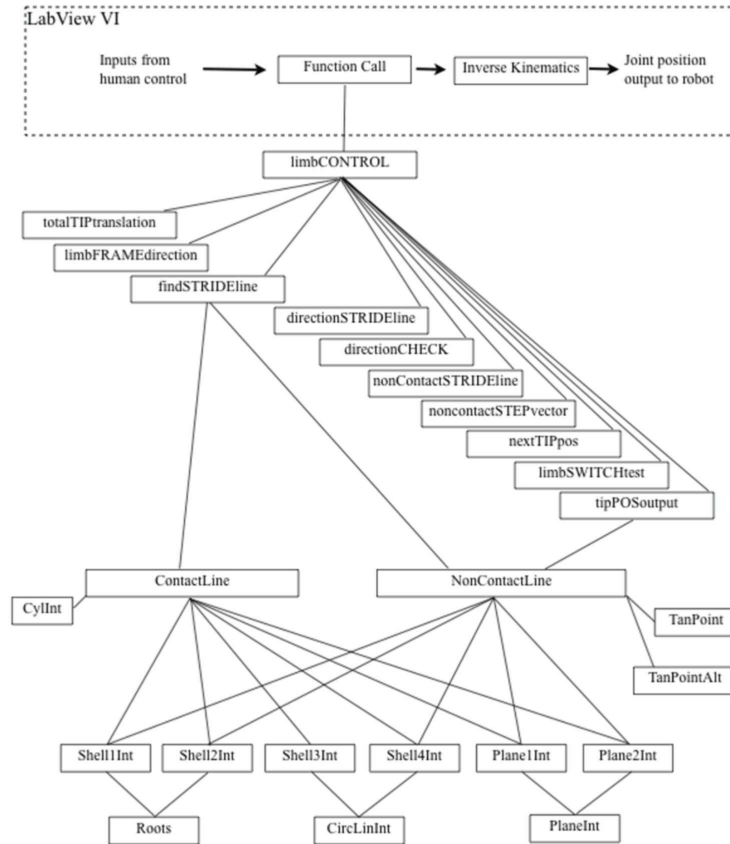

Figure 5.7: The basic architecture of the general walking algorithm consists of a LabVIEW VI which calls a set of MATLAB m-files.

- directionSTRIDEline: This function aranges the points which describe a stride-line so that the first point is in the forward direction of the limb tip motion.
- directionCHECK: Because "forward" is in opposite directions for contact and not-contact limbs, this function ensures that the forward endpoint is corrected where necessary.
- nonContactSTRIDEline: This function ensures that the non-contact stride-line includes the current limb tip position and is not just the pre-strideline.
- noncontactSTEPvector: This function finds the non-contact directional vector and multiplies it by the step size.
- nextTIPpos: This function finds the next limb tip position by adding the step vector to the current tip position.
- limbSWITCHtest: This function tests to see if a limb switch in necessary, based on how close the next limb tip position is to the workspace boundary.
- tipPOSoutput: This function outputs the next limb tip positions. If a limb switch is necessary, this function outputs two sets of limb tip positions, instead of one. To find the second set of limb tip positions, this function calls noncontactLINE. It does not call contactLINE because the noncontactLINE function is used to find the stride-line for noncontact limbs which are switching to contact limbs, as well as contact limbs which are switching to non-contact limbs.

These m-files can be found in Appendix A.6 through Appendix A.15.

The next level of functions find the endpoints of the stride-line. Two separate functions are used: contactLINE is used for limbs in contact with walking surface, and noncontactLINE is used for limbs which are not in contact with the walking surface. Each function calls the shell and plane functions in order to find all possible intersection points. However, if an intersection point is closer to the current limb tip position, in either direction along the stride-line, the stored point is replaced. In this manner these functions find the two closest intersection points to the current limb tip position. The m-file contactLINE checks for intersections on Shells 1 through

4 as well as the two bounding planar sections, by calling the appropriate functions. The m-file noncontactLINE calls all the same functions except for Shell3Int. This is because it is impossible for the non-contact line to intersect Shell 3, unless the walking height of the robot is less than the radius of Shell 3. However, as the walking height of the robot is kept between the radius of Shell 3 and the radius of Shell 4, this cannot happen. The m-file contactLINE also checks for intersection points with the buffer cylinder by calling the CylInt function. The m-file noncontactLINE calls the m-file tanpoint or tanpointALT in order to use a point other than the current limb tip position to find the closest pre-strideline endpoints. This is done because the current limb tip position is not necessarily on the pre-stride-line. The m-file tanpoint is used to find a point of tangent to the buffer cylinder on a line parallel to the robot body velocity. If the point returned by tanpoint is outside the general workspace, tanpointALT is used to find the orthogonal intersection point between the height-line and stride-line.

Each shell or plane which bounds the general workspace has a corresponding m-file. These m-files find all the points of intersection between the stride-line and the given surface. Intersection points with torus sections (Shells 1 and 2) are found by Shell1Int and Shell2Int, respectively. These functions each call the Roots m-file, which returns the values of t or roots to the polynomial, which represents the equating of the equation of a torus and the equation of a line. The m-files Shell1Int and Shell2Int then substitute these t values into the parametric equation of the stride-line or pre-stride-line to find the points of intersection. The points of intersection are screened so that points which do not lie on the section of the torus which bounds the general workspace are excluded.

Points of intersection with the two spherical shells, Shell 3 and Shell 4 are output by m-file Shell3Int and Shell4Int, respectively. Each of these functions call the CircLineInt m-file, which outputs points of intersection between a line and a circle. These points are then screened by Shell3Int and Shell4Int to ensure that they lie on the section of the sphere which bounds the general workspace.

Similarly, the points of intersection with the two planar sections are output by the m-files Plane1Int and Plane2Int. These functions call the m-file PlaneInt, which finds the points of intersection of a line and a plane which contains the z_2-axis. The points are then screened by Plane1Int and Plane2Int to ensure that they lie on the section of the plane which bounds the general workspace. The m-file Plane1Int finds points of intersection on

the plane which intersects the x_2-y_2 plane at -10 degrees, while Plane2Int finds point of intersection on the plane which intersects the x_2-y_2 plane at 190 degrees.

5.4 Implemented Walking Algorithm

A simplified version of the general walking algorithm was developed and implemented on MARS by Open Tech Inc. This adaptive iterative walking algorithm uses the 2D circular workspace. Therefore this walking algorithm is not fully capable of changing robot body roll and pitch while walking. However, the algorithm is computationally less intensive. Currently Open Tech's algorithm operates on a MACBook Pro laptop at 10 to 60 Hz.

There are two inputs to this algorithm:

- The translational vector in 3-space
- the angular velocity about the z_0-axis

The inputs are translated from body coordinates to limb coordinates. This translation allows for the limbs to be attached to the body at any position and orientation. The next limb tip position is found, and the inverse kinematics are used to generate the actuator positions. The contact limb tips move with each iteration within the circular workspace. However, if the next limb tip position is found to be outside the workspace, the limbs switch. The non-contact limb tips move at a height above the walking surface. The height of the non-contact limb tips is specified by Equation 5.50:

$$\text{Height} = \sqrt{1 - (r_T - r_c)} \tag{5.50}$$

where r_T is the distance from the limb tip to the center of the circular workspace, and r_c is the radius of the circular workspace. The non-contact limb tips must move in the opposite direction of the contact limb tips in order to reach the forward edge of the workspace before the limb switch. Further the non-contact limbs must move at a greater velocity than the contact limbs as the algorithm inputs may change in the middle of a stride. If the non-contact limb tips did not move faster than the contact limb tips, constant change in direction could result in a situation where the limbs would need to continuously switch and not be able to achieve the required velocity of the robot body.

Chapter 6

Future Work

The robot design, workspace analysis, and gait generation algorithms presented in this thesis for MARS are only the beginning of a body of research which could be explored on this project. In this chapter the most probable next steps are briefly discussed.

6.1 New Approach to Replace Inverse Kinematics

Revolute Joint 1 was not used for walking in the algorithms of this thesis. However, it is possible to use all four degrees of freedom of the limbs for walking. During the time frame of the work on this thesis, Open Tech developed an independent algorithm for positioning any multi-DOF robotic system. This algorithm is based on a lookup table and neural net combination which determines the optimal forward kinematics for a future position, given the current position. This algorithm is kinematics-independent in that it learns the kinematics for the device from device position and orientation feedback. This code could easily be used in the walking algorithms defined in this thesis in place of the inverse kinematics, thus allowing for the use of all four degrees of freedom while walking. However, such an application would require further workspace analysis, as the workspace has not been analyzed for the use of all four DOF.

6.2 Workspace Analysis Using Spherical Coordinates

Rectilinear coordinates were used in this thesis for workspace analysis. However, the analysis could be simplified with the use of spherical coordinates. Advantages to spherical coordinates include: a more direct relationship between actuator position and kinematic equations, simplification of the expressions defining the workspace, reduction in computational time due to the elimination of trigonometric functions.

6.3 Kinematics Redesign

The MARS limbs were designed to be kinematically similar to the JPL LEMUR IIb robot. While this design provides for the desirable characteristics of a kinematically-spherical proximal joint and a 1-DOF distal joint, other configurations can provide the same characteristics. One of the main challenges for the design of MARS was developing a rigid structure around the proximal joint. This was challenging primarily because of the location of Revolute Joint 1. Redesign of the proximal joint could provide a much stiffer structure and still have the three axes of rotation intersect at a single point. Aspects to consider in redesign include: how the workspace is effected, how the inverse kinematics is effected, how the Jacobian and therefore singularities are effected, and how the torque and stresses of the both the motors and structure are effected.

6.4 Alternative Walking Algorithms

This thesis presents only one of many possible walking algorithms for the MARS platform. This section discusses other algorithms which warrant consideration.

Insect-based walking algorithms have been studied and implemented over the past several years. These algorithms use reactionary gate formation, distributed control, or a combination of the two. The implementation platforms are primarily patterned after cockroaches or stick insects, with three legs in a row on either side of the body. However, such algorithms could also be implemented on the MARS platform. By segregating the MARS limbs into two

groups on either side of an arbitrary bisection line, the platform can function–in a limited capacity–similar to an insect with a fixed forward direction. By assigning a single stride to each limb, the limbs can function similarly to those used with insect-based algorithms. These two limitations produce a platform with functionality equal to platforms used for insect-based algorithm demonstration. The further expansion to full MARS capabilities with the use of insect-based control algorithms is a worthy candidate for future research.

Instead of using a continuously adaptive walking algorithm, one which adapts at the completion of each step could be used. Such an algorithm could operate similarly to the algorithm of this thesis, except that new stride-lines would be found at the completion of a full step rather than iteratively during a step.

6.5 Walking Algorithm Additions

Many useful functions already employed with other hexapoidal research platforms could be implemented on the MARS platform without the implementation of additional sensors. Such functions requiring force feedback could make use of the current/force feedback utility of the Dynamixel actuators. Autonomous navigation of rough terrain could be greatly enhanced by the use of searching algorithms and an elevator reflex. If a leg does not encounter a solid foothold, it searches in progressively higher and farther reaching motions until one is found. Similarly if a step motion is blocked by an obstacle, an elevator reflex causes the leg to step higher to avoid the obstacle. Load balancing, via actuator force feedback, could be employed to reduce body tilt and improve stability while traversing rough terrain.

Appendix A

Programming Developed for the General Walking Algorithm

A LabVIEW VI and a set of MATLAB m-files were developed to demonstrate the general workspace walking algorithm. A Mathematica script was used to simulate the walking gait of MARS using the output from the LabVIEW VI. These functions, program, and script are presented in this appendix.

The VI and m-files for finding a stride-line in the general workspace are architecturally arranged as shown in Figure A.1. It should be noted that for each iteration, stride-line intersections with all shells, planes, and the buffer cylinder are checked.

The programing presented in this appendix are ordered as follows:

- The Mathematica script used to simulate the MARS gait
- A description of the LabVIEW VI
- The MATLAB m-files arranged in accordance with Figure A.1: top to bottom, left to right

A.1 Mathematica Gait Simulation Script

Introduction

The purpose of this *Mathematica* notebook is to generate animations for visualizing the motion of the LEMUR IIa robot. It first defines a set of

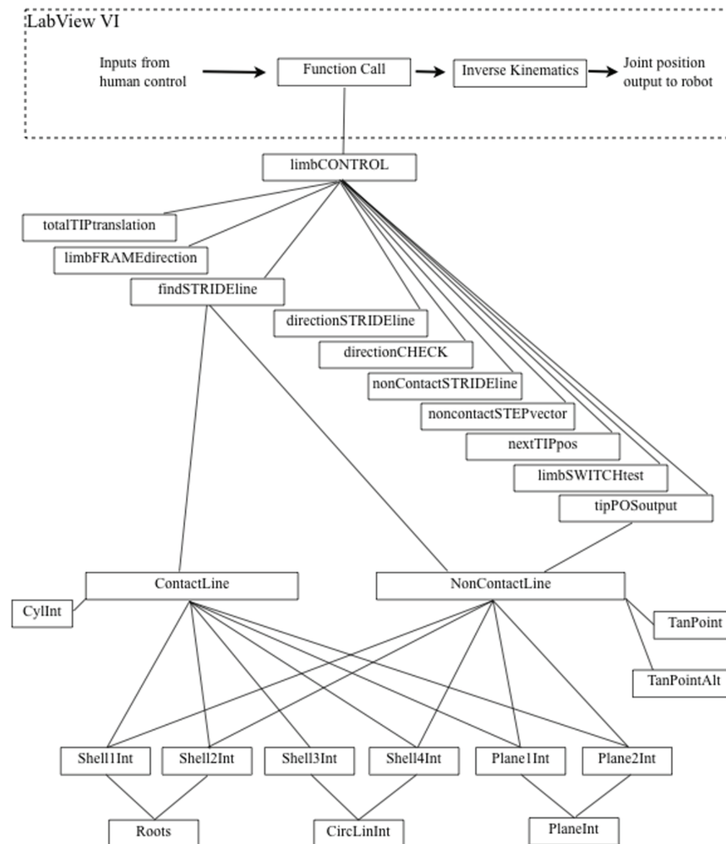

Figure A.1: A set of hierarchically-arranged MATLAB functions were developed to demonstrate the feasibility of using a computer to find the stride-line through the general workspace.

graphics functions for initializing the geometry and defines the configuration routine for the LEMUR IIa model. Then it reads in a data file (text file) and produces a series of figures to generate the animation (it can export a QuickTime movie file on a Macintosh system) The code is flexible and easy to modify to include many features. It can be used for checking the joint motion data before testing on real hardware and to explore new types of motion, or to generate animation movie files and figures for presentations, reports, or proposals. Our next task is to include a general gait planner, collision detection, static force & stability analysis code (with gravity and/or sticky feet), camera vision models which will produce movies with views from the camera which can be very helpful.
All work will be shared with the good people at JPL.

Turn off spelling error warnings

```
Off[General::spelll]
Off[General::spell]
```

Spatial Kinematics Functions

D-H Matrices

Define the D-H Matrix using Craig's notation (x then z)

```
DH[a_, α_, d_, θ_]:=
{{Cos[θ], −Sin[θ], 0, a}, {Cos[α]Sin[θ], Cos[α]Cos[θ], −Sin[α], −dSin[α]},
{Sin[α]Sin[θ], Cos[θ]Sin[α], Cos[α], dCos[α]}, {0, 0, 0, 1}}
```

Extract the rotation matrix/displacement vector from the D-H Matrix

```
GetR[H_]:={{H[[1, 1]], H[[1, 2]], H[[1, 3]]}, {H[[2, 1]], H[[2, 2]], H[[2, 3]]},
{H[[3, 1]], H[[3, 2]], H[[3, 3]]}}
Getd[H_]:={H[[1, 4]], H[[2, 4]], H[[3, 4]]}
```

Graphics Functions

Rotation & Translation Matrices

These commands set up the Rotation matrices and use the matrices to set up more complex rotations and translations

```
<< Graphics`Shapes`
```

General::obspkg : Graphics`Shapes` is now obsolete. The legacy version being loaded may conflict with

```
Rotz[t_]:={{Cos[t], −Sin[t], 0}, {Sin[t], Cos[t], 0}, {0, 0, 1}}
Rotx[t_]:={{1, 0, 0}, {0, Cos[t], Sin[t]}, {0, −Sin[t], Cos[t]}}
Roty[t_]:={{Cos[t], 0, Sin[t]}, {0, 1, 0}, {−Sin[t], 0, Cos[t]}}
Rotz2[t_]:={{Cos[t], −Sin[t], 0, 0}, {Sin[t], Cos[t], 0, 0}, {0, 0, 1, 0}, {0, 0, 0, 1}}
Rotx2[t_]:={{1, 0, 0, 0}, {0, Cos[t], Sin[t], 0}, {0, −Sin[t], Cos[t], 0}, {0, 0, 0, 1}}
Roty2[t_]:={{Cos[t], 0, Sin[t], 0}, {0, 1, 0, 0}, {−Sin[t], 0, Cos[t], 0}, {0, 0, 0, 1}}
TranzX[d_]:={{1, 0, 0, d}, {0, 1, 0, 0}, {0, 0, 1, 0}, {0, 0, 0, 1}}
TranzY[d_]:={{1, 0, 0, 0}, {0, 1, 0, d}, {0, 0, 1, 0}, {0, 0, 0, 1}}
TranzZ[d_]:={{1, 0, 0, 0}, {0, 1, 0, 0}, {0, 0, 1, d}, {0, 0, 0, 1}}
RotationMatrix3D[tx_, ty_, tz_]:=Rotx[tx].Roty[ty].Rotz[tz]
RotationMatrix3D2[tx_, ty_, tz_, dx_, dy_dz_]:=
Rotx2[tx].Roty2[ty].Rotz2[tz].TranzX[dx].TranzY[dy].TranzZ[dz]
RotateShapeR[shape_, R_]:=
Block[{rotmat = R},
shape/.{poly : Polygon[_] :→ Map[(rotmat.#)&, poly, {2}],
line : Line[_] :→ Map[(rotmat.#)&, line, {2}],
point : Point[_] :→ Map[(rotmat.#)&, point, {1}]}]
TransformShape[shape_, R_, d_]:=TranslateShape[RotateShapeR[shape, R], d]
```

Redering Options

ShowOptions sets up the rendering characteristics. Modify this to change carema view angles, lighting conditions, etc.

```
ShowOptions = {PlotRange → All, ViewPoint → {3, 1, 1}, BoxRatios → {22, 20, 6}};
```

Link Primitives

These commands set up the shapes of which the limbs (or any objects) are composed.

```
RJoint[r_, h_]:={Cylinder[r, h/2, 12], TranslateShape[Cone[r, 0, 12], {0, 0, h/2}],
TranslateShape[Cone[r, 0, 12], {0, 0, −h/2}]}
XBox[w_, h_]:=
{Polygon[{{0, w/2, w/2}, {0, −w/2, w/2}, {0, −w/2, −w/2}, {0, w/2, −w/2},
{0, w/2, w/2}}],
Polygon[{{h, w/2, w/2}, {h, −w/2, w/2}, {h, −w/2, −w/2}, {h, w/2, −w/2},
{h, w/2, w/2}}],
Polygon[{{0, w/2, w/2}, {0, −w/2, w/2}, {h, −w/2, w/2}, {h, w/2, w/2},
{0, w/2, w/2}}],
Polygon[{{0, −w/2, w/2}, {0, −w/2, −w/2}, {h, −w/2, −w/2}, {h, −w/2, w/2},
```

```
{0, -w/2, w/2}}],
Polygon[{{0, -w/2, -w/2}, {0, w/2, -w/2}, {h, w/2, -w/2}, {h, -w/2, -w/2},
{0, -w/2, -w/2}}],
Polygon[{{0, w/2, -w/2}, {0, w/2, w/2}, {h, w/2, w/2}, {h, w/2, -w/2},
{0, w/2, -w/2}}]}
ZBox[w_, h_]:=RotateShape[XBox[w, h], 0, -90Degree, 0]
LinkX[r_, h_, l_]:={RJoint[r, h], XBox[1.2r, l]}
LinkZ[r_, h_, l_]:={RJoint[r, h], ZBox[1.2r, l]}
```

LEMUR IIa

Kinematic Definition

The DefineLEMUR command is used before plotting to dimension the LEMUR

```
DefineLEMUR[l0_, l1_, l2_, r_, h_]:=(
Sides = Cylinder[188.340, 30, 6];
Link0 = LinkX[r, h, l0];
Link1 = LinkX[r, h, 0];
Link2 = LinkX[r, h, 0];
Link3 = LinkX[r, h, l1];
Link4 = LinkX[r, h, l2];
Link6 = LinkX[1, 1, 5];
H0[t0_]:=DH[0, 0, 0, t0];
H1[t1_]:=DH[l0, 0, 0, t1];
H2[t2_]:=DH[0, 90Degree, 0, t2];
H3[t3_]:=DH[0, -90Degree + t3, 0, 0];
H4[t4_]:=DH[l1, -90Degree, 0, t4]; )
```

The ConfigureLEMUR command is used during plotting to orient each limb individually

```
ConfigureLEMUR[t0_, t1_, t2_, t3_, t4_, t10_, t11_, t12_, d1_, d2_, d3_]:=(
Htemp = TranzX[d1].TranzY[d2].TranzZ[d3].Rotz2[t10].Rotx2[t11].Roty2[t12].H0[t0];
Rin = GetR[Htemp];
din = Getd[Htemp];
L0 = TransformShape[Link0, Rin, din];
Htemp = TranzX[d1].TranzY[d2].TranzZ[d3].Rotz2[t10].Rotx2[t11].Roty2[t12].H0[t0].H1[t1];
Rin = GetR[Htemp];
din = Getd[Htemp];
L1 = TransformShape[Link1, Rin, din];
Htemp = TranzX[d1].TranzY[d2].TranzZ[d3].Rotz2[t10].Rotx2[t11].Roty2[t12].
H0[t0].H1[t1].H2[t2];
```

Rin = GetR[Htemp];
din = Getd[Htemp];
L2 = TransformShape[Link2, Rin, din];
Htemp = TranzX[d1].TranzY[d2].TranzZ[d3].Rotz2[t10].Rotx2[t11].Roty2[t12].
H0[t0].H1[t1].H2[t2].H3[t3];
Rin = GetR[Htemp];
din = Getd[Htemp];
L3 = TransformShape[Link3, Rin, din];
Htemp = TranzX[d1].TranzY[d2].TranzZ[d3].Rotz2[t10].Rotx2[t11].Roty2[t12].
H0[t0].H1[t1].H2[t2].H3[t3].H4[t4];
Rin = GetR[Htemp];
din = Getd[Htemp];
L4 = TransformShape[Link4, Rin, din];
Return[{L1, L2, L3, L4}];)

The ConfigureLEMURBODY command is used during plotting to orient the LEMUR body

ConfigureLEMURBODY[t10_, t11_, t12_, d1_, d2_, d3_]:=(
Htemp = TranzX[d1].TranzY[d2].TranzZ[d3].Rotz2[t10].Rotx2[t11].Roty2[t12];
Rin = GetR[Htemp];
din = Getd[Htemp];
Body = TransformShape[Sides, Rin, din];
Return[{Body}];)

Read Data File

Data File Format & Definition

This section imports the movement data used to configure LEMUR with each plotting iteration. Table 1 shows the first ten rows of a sample table and the designation of each column. The first column is simply an iteration count which can be used for defining time steps. The next group of colums defines the LEMUR's position. This is followed by the rotation column group. The last 6 column groups defines the LEMUR's limb joints. The joint designation for each limb as well as the direction for positive rotaion is depicted in Figure 1. Note that with all joint angles et to zero, the limbs point straight out.

Table 1

Figure 1: The arows indicate the positive rotation direction for each joint

Read in the Data File

Note that you may need to define the directory for the location of the data file first such as:

```
(*SetDirectory["Motion Data"]; *)
```

These commands import the data file which includes the table of lengths and angles which define the movement of the LEMUR. The table is stored as MoDat for Movement Data.The value n, the number of rows in the table, is stored for use later in plotting LEMUR positions.

```
MoDat = Import["LEMUR_Motion.txt", "Table"];
n = Length[MoDat]
168
```

Define the Enviornment

LEMURenvirnoment sets up the objects in the environment and the plane about which the LEMUR moves.Modify this to include obstacles, terrain, etc.

```
XYZ = Graphics3D[{AbsoluteThickness[2], Line[{{0, 0, 0}, {100, 0, 0}}],
Line[{{0, 0, 0}, {0, 100, 0}}], Line[{{0, 0, 0}, {0, 0, 100}}], Text["X", {1, 0, 0}, {1, 0}],
Text["Y", {0, 1, 0}, {−1, 0}], Text["Z", {0, 0, 1}, {−1, −1}]}];
FloorTile = Graphics3D[Cuboid[{−600, −1200, 0}, {1400, 800, 0}]];
Obstacle1 = Graphics3D[Cuboid[{500, 50, 0}, {550, 300, 400}]];
Obstacle2 = TranslateShape[Graphics3D[Cylinder["188.34", 100, 8]], {1000, −800, 100}];
LEMURenvironment = {FloorTile, Obstacle1, Obstacle2};
```

Animation

This section defines the LEMUR, then uses a do loop to plot each subsequent LEMUR position from the MoDat table

```
DefineLEMUR[188.34, 126.97, 152.40, 15, 30]

Body = ConfigureLEMURBODY[0, 0, 0, 0, 0, 100];
limb1 = ConfigureLEMUR[30Degree, 0Degree, 45Degree, 0Degree, 100Degree, 0, 0, 0, 0, 0, 100];
limb2 = ConfigureLEMUR[90Degree, 0Degree, 50Degree, 10Degree, 120Degree, 0, 0, 0, 0, 0, 100];
limb3 = ConfigureLEMUR[150Degree, 0Degree, 80Degree, 0Degree, 100Degree, 0, 0, 0, 0, 0, 100];
limb4 = ConfigureLEMUR[210Degree, 10Degree, 70Degree, −10Degree, 70Degree, 0, 0,
0, 0, 0, 100];
limb5 = ConfigureLEMUR[270Degree, 20Degree, 20Degree, −5Degree, 20Degree, 0, 0, 0, 0, 0, 100];
limb6 = ConfigureLEMUR[330Degree, 5Degree, 50Degree, 10Degree, 90Degree, 0, 0, 0, 0, 0, 100];
```

```
Show[Graphics3D[{Body, limb1, limb2, limb3, limb4, limb5, limb6}]]

Do[Body = ConfigureLEMURBODY[MoDat[[i, 7]], MoDat[[i, 5]], MoDat[[i, 6]], MoDat[[i, 2]],
MoDat[[i, 3]], MoDat[[i, 4]]];
limb1 = ConfigureLEMUR[30Degree, MoDat[[i, 8]], MoDat[[i, 9]], MoDat[[i, 10]],
MoDat[[i, 11]], MoDat[[i, 7]], MoDat[[i, 5]], MoDat[[i, 6]], MoDat[[i, 2]],
MoDat[[i, 3]], MoDat[[i, 4]]];
limb2 = ConfigureLEMUR[90Degree, MoDat[[i, 12]], MoDat[[i, 13]], MoDat[[i, 14]],
MoDat[[i, 15]], MoDat[[i, 7]], MoDat[[i, 5]], MoDat[[i, 6]], MoDat[[i, 2]],
MoDat[[i, 3]], MoDat[[i, 4]]];
limb3 = ConfigureLEMUR[150Degree, MoDat[[i, 16]], MoDat[[i, 17]], MoDat[[i, 18]],
MoDat[[i, 19]], MoDat[[i, 7]], MoDat[[i, 5]], MoDat[[i, 6]], MoDat[[i, 2]],
MoDat[[i, 3]], MoDat[[i, 4]]];
limb4 = ConfigureLEMUR[210Degree, MoDat[[i, 20]], MoDat[[i, 21]], MoDat[[i, 22]],
MoDat[[i, 23]], MoDat[[i, 7]], MoDat[[i, 5]], MoDat[[i, 6]], MoDat[[i, 2]],
MoDat[[i, 3]], MoDat[[i, 4]]];
limb5 = ConfigureLEMUR[270Degree, MoDat[[i, 24]], MoDat[[i, 25]], MoDat[[i, 26]],
MoDat[[i, 27]], MoDat[[i, 7]], MoDat[[i, 5]], MoDat[[i, 6]], MoDat[[i, 2]],
MoDat[[i, 3]], MoDat[[i, 4]]];
limb6 = ConfigureLEMUR[330Degree, MoDat[[i, 28]], MoDat[[i, 29]], MoDat[[i, 30]],
MoDat[[i, 31]], MoDat[[i, 7]], MoDat[[i, 5]], MoDat[[i, 6]], MoDat[[i, 2]],
MoDat[[i, 3]], MoDat[[i, 4]]];
Print[Graphics3D[{Body, limb1, limb2, limb3, limb4, limb5, limb6}]], {i, n}]
```

A.2 LabVIEW VI

In this section details of the LabVIEW VI with embedded MATLAB code is described. The most noteable feature of the VI is that the MATLAB script node containing the Function Run m-file is within a true/false case structure. The setting of the structure is controlled by the “switchtest” output of the Function Run m-file. Within the same timed loop as the true/false case structure is a seperate MATLAB script node containing the Inverse Kinematics m-file. The inputs and outputs of both MATLAB script nodes are as follows:

Inputs for Function Run

- ux: body direction vector component along the x axis
- uy: body direction vector component along the y axis

- uz: body direction vector component along the z axis
- rx: body rotation about the x axis
- ry: body rotation about the y axis
- rz: body rotation about the z axis
- x1: x-axis position of the tip of Limb 1
- y1: y-axis position of the tip of Limb 1
- z1: z-axis position of the tip of Limb 1
- x2: x-axis position of the tip of Limb 2
- y2: y-axis position of the tip of Limb 2
- z2: z-axis position of the tip of Limb 2
- x3: x-axis position of the tip of Limb 3
- y3: y-axis position of the tip of Limb 3
- z3: z-axis position of the tip of Limb 3
- x4: x-axis position of the tip of Limb 4
- y4: y-axis position of the tip of Limb 4
- z4: z-axis position of the tip of Limb 4
- x5: x-axis position of the tip of Limb 5
- y5: y-axis position of the tip of Limb 5
- z5: z-axis position of the tip of Limb 5
- x6: x-axis position of the tip of Limb 6
- y6: y-axis position of the tip of Limb 6
- z6: z-axis position of the tip of Limb 6
- contact: a value, 1 or 2, which determines whether even or odd numbered limbs are in contact

Inputs for Function Run

- Oddx1: the next x-axis position of Limb 1
- Oddy1: the next y-axis position of Limb 1
- Oddz1: the next z-axis position of Limb 1
- Oddx3: the next x-axis position of Limb 3
- Oddy3: the next y-axis position of Limb 3
- Oddz3: the next z-axis position of Limb 3
- Oddx5: the next x-axis position of Limb 5
- Oddy5: the next y-axis position of Limb 5
- Oddz5: the next z-axis position of Limb 5
- Odd2x1: the second next x-axis position of Limb 1 (used for limb switching)
- Odd2y1: the second next y-axis position of Limb 1 (used for limb switching)
- Odd2z1: the second next z-axis position of Limb 1 (used for limb switching)
- Odd2x3: the second next x-axis position of Limb 3 (used for limb switching)
- Odd2y3: the second next y-axis position of Limb 3 (used for limb switching)
- Odd2z3: the second next z-axis position of Limb 3 (used for limb switching)
- Odd2x5: the second next x-axis position of Limb 5 (used for limb switching)
- Odd2y5: the second next y-axis position of Limb 5 (used for limb switching)
- Odd2z5: the second next z-axis position of Limb 5 (used for limb switching)
- Evenx2: the next x-axis position of Limb 2
- Eveny2: the next y-axis position of Limb 2
- Evenz2: the next z-axis position of Limb 2
- Evenx4: the next x-axis position of Limb 4
- Eveny4: the next y-axis position of Limb 4
- Evenz4: the next z-axis position of Limb 4
- Evenx6: the next x-axis position of Limb 6

- Eveny6: the next y-axis position of Limb 6
- Evenz6: the next z-axis position of Limb 6
- Even2x2: the second next x-axis position of Limb 2 (used for limb switching)
- Even2y2: the second next y-axis position of Limb 2 (used for limb switching)
- Even2z2: the second next z-axis position of Limb 2 (used for limb switching)
- Even2x4: the second next x-axis position of Limb 4 (used for limb switching)
- Even2y4: the second next y-axis position of Limb 4 (used for limb switching)
- Even2z4: the second next z-axis position of Limb 4 (used for limb switching)
- Even2x6: the second next x-axis position of Limb 6 (used for limb switching)
- Even2y6: the second next y-axis position of Limb 6 (used for limb switching)
- Even2z6: the second next z-axis position of Limb 6 (used for limb switching)
- contact: a value, 1 or 2, which determines whether even or odd numbered limbs are in contact
- switchtest: a value, which if 1 and not zero, requires that a limb switch takes place

Inputs for Inverse Kinematics

- x1: x-axis position of the tip of Limb 1
- y1: y-axis position of the tip of Limb 1
- z1: z-axis position of the tip of Limb 1
- x2: x-axis position of the tip of Limb 2
- y2: y-axis position of the tip of Limb 2
- z2: z-axis position of the tip of Limb 2
- x3: x-axis position of the tip of Limb 3
- y3: y-axis position of the tip of Limb 3
- z3: z-axis position of the tip of Limb 3

- x4: x-axis position of the tip of Limb 4
- y4: y-axis position of the tip of Limb 4
- z4: z-axis position of the tip of Limb 4
- x5: x-axis position of the tip of Limb 5
- y5: y-axis position of the tip of Limb 5
- z5: z-axis position of the tip of Limb 5
- x6: x-axis position of the tip of Limb 6
- y6: y-axis position of the tip of Limb 6
- z6: z-axis position of the tip of Limb 6

Outputs for Inverse Kinematics

- t21: the joint angle for Revolute Joint 1 on Limb 1
- t31: the joint angle for Revolute Joint 3 on Limb 1
- t41: the joint angle for Revolute Joint 4 on Limb 1
- t22: the joint angle for Revolute Joint 1 on Limb 2
- t32: the joint angle for Revolute Joint 3 on Limb 2
- t42: the joint angle for Revolute Joint 4 on Limb 2
- t23: the joint angle for Revolute Joint 1 on Limb 3
- t33: the joint angle for Revolute Joint 3 on Limb 3
- t43: the joint angle for Revolute Joint 4 on Limb 3
- t24: the joint angle for Revolute Joint 1 on Limb 4
- t34: the joint angle for Revolute Joint 3 on Limb 4
- t44: the joint angle for Revolute Joint 4 on Limb 4
- t25: the joint angle for Revolute Joint 1 on Limb 5
- t35: the joint angle for Revolute Joint 3 on Limb 5
- t45: the joint angle for Revolute Joint 4 on Limb 5

- t26: the joint angle for Revolute Joint 1 on Limb 6
- t36: the joint angle for Revolute Joint 3 on Limb 6
- t46: the joint angle for Revolute Joint 4 on Limb 6

A.3 Function Call

```
%function[Odd,Odd2,Even,Even2,contact] = MARStipPOS(ux,uy,uz,rx,ry,rz,h,
contact,evenold,oddold)
switchtest = 0;
evenold = [x2,y2,z2;x4,y4,z4;x6,y6,z6];
oddold = [x1,y1,z1;x3,y3,z3;x5,y5,z5];
% ####################################################################
%
% This function finds the tip positions for the MARS limbs
%
% Inputs: (ux,uy,uz) The direction vector of the robot.  This is not
%                    necesarily a unit vector.
%         (rx,ry,rz) Rotations about the x, y, and z-axis
%         h  the walking hieght of the robot
%         contact = 1 if odd limbs are in contact 2 for even
%         even  the 3X3 matrix of (x,y,z) limb tip positions (even limbs)
%             [x2 y2  z2]
%             [x4 y4  z4]
%             [x6 y6  z6]
%
%          odd    the 3X3 matrix of (x,y,z) limb tip positions (odd limbs)
%             [x1 y1  z1]
%             [x3 y3  z3]
%             [x5 y5  z5]
%
% Outputs: Odd  the tip positions of the odd numbered limbs
%          Even the tip positions of the even numbered limbs
%          Odd2   the second set of tip positions for the odd numbered
%                 limbs used if the limbs switch with regard to contact and
%                 non-contact
%          Even2  the second set of tip positions for the even numbered
%                 limbs used if the limbs switch with regard to contact and
%                 non-contact
%          contact = 1 if odd limbs are in contact 2 for even
%
% ####################################################################

% Define Variables
r = 0.5; %the radius of the z-axsis singularity buffer cyliner
buffer = 0.5; %the distance the limb tips stay away from the end of the
                        %stride-lines
steph = 0.5; % the hieght of the steps

% find the length of each step based on (ux,uy,uz), efects speed
step = (sum([ux,uy,uz].^2))^0.5 * 0.1;

% Temporary comment (delete after 3-07) Some adjustement may be necesary
```

```
% for (rx,ry,rz) depending on LabVIEW

[Even, Even2, Odd, Odd2, contact] = MARSlimbCONTROL(evenold, oddold,
ux,uy,uz, r, h, steph, step, rx,ry,rz, contact, buffer);

% switch the contact argument
if ~(isempty(Odd2))
    if contact == 1
        contact = 2;
    elseif contact == 2
        contact = 1;
    end
end

if isempty(Odd2)
     switchtest = 1;
     Even2x2 = 0;
     Even2x4 = 0;
     Even2x6 = 0;
     Even2y2 = 0;
     Even2y4 = 0;
     Even2y6 = 0;
     Even2z2 = 0;
     Even2z4 = 0;
     Even2z6 = 0;
     Odd2x1 =0;
     Odd2x3 = 0;
     Odd2x5 = 0;
     Odd2y1 =0;
     Odd2y3 = 0;
     Odd2y5 = 0;
     Odd2z1 =0;
     Odd2z3 = 0;
     Odd2z5 = 0;
     Evenx2 = Even(1,1);
     Evenx4 = Even(2,1);
     Evenx6 = Even(3,1);
     Eveny2 = Even(1,2);
     Eveny4 = Even(2,2);
     Eveny6 = Even(3,2);
     Evenz2 = Even(1,3);
     Evenz4 = Even(2,3);
     Evenz6 = Even(3,3);
     Oddx1 =Odd(1,1);
     Oddx3 = Odd(2,1);
     Oddx5 = Odd(3,1);
     Oddy1 = Odd(1,2);
     Oddy3 = Odd(2,2);
     Oddy5 = Odd(3,2);
     Oddz1 = Odd(1,3);
     Oddz3 = Odd(2,3);
     Oddz5 = Odd(3,3);
else
     Even2x2 = Even2(1,1);
     Even2x4 = Even2(2,1);
     Even2x6 = Even2(3,1);
     Even2y2 = Even2(1,2);
```

```
    Even2y4 = Even2(2,2);
    Even2y6 = Even2(3,2);
    Even2z2 = Even2(1,3);
    Even2z4 = Even2(2,3);
    Even2z6 = Even2(3,3);
    Odd2x1 =Odd2(1,1);
    Odd2x3 = Odd2(2,1);
    Odd2x5 =Odd2(3,1);
    Odd2y1 =Odd2(1,2);
    Odd2y3 = Odd2(2,2);
    Odd2y5 = Odd2(3,2);
    Odd2z1 =Odd2(1,3);
    Odd2z3 = Odd2(2,3);
    Odd2z5 = Odd2(3,3);
    Evenx2 = Even(1,1);
    Evenx4 = Even(2,1);
    Evenx6 = Even(3,1);
    Eveny2 = Even(1,2);
    Eveny4 = Even(2,2);
    Eveny6 = Even(3,2);
    Evenz2 = Even(1,3);
    Evenz4 = Even(2,3);
    Evenz6 = Even(3,3);
    Oddx1 =Odd(1,1);
    Oddx3 = Odd(2,1);
    Oddx5 = Odd(3,1);
    Oddy1 = Odd(1,2);
    Oddy3 = Odd(2,2);
    Oddy5 = Odd(3,2);
    Oddz1 = Odd(1,3);
    Oddz3 = Odd(2,3);
    Oddz5 = Odd(3,3);
end
```

A.4 Inverse Kinematics

```
% ########################################################################
%
% This script performs the inverse kenematics for the MARS limbs
%
% Inputs: (x1,y1,z1)
%         (x2,y2,z2)
%         (x3,y3,z3)
%         (x4,y4,z4)
%         (x5,y5,z5)
%         (x6,y6,z6) the tip position of each limb in the limb specific
%                    cordinate frame
%
% Outputs: (t21,t31,t41)
%          (t22,t32,t42)
%          (t23,t33,t43)
%          (t24,t34,t44)
%          (t25,t35,t45)
%          (t26,t36,t46) the joint angles (t2,t3,t4) for each of the six
%                        limbs
```

```
%
% ########################################################################
%x1=1;x2=1;x3=2;x4=1;x5=2;x6=2;y1=1;y2=2;y3=1;y4=2;y5=2;y6=2;z1=-8;
%z2=-8;z3=-8;z4=-8;z5=-8;z6=-8;

t21 = atan2(z1,y1);
D1 = (x1^2+y1^2+z1^2-61)/60;
t41 = atan2(-(1-D1^2)^0.5,D1);
t31 = atan2((y1^2+z1^2)^0.5,x1-5) - atan2(5+6*cos(t41),6*sin(t41));

t22 = atan2(z2,y2);
D2 = (x2^2+y2^2+z2^2-61)/60;
t42 = atan2(-(1-D2^2)^0.5,D2);
t32 = atan2((y2^2+z2^2)^0.5,x2-5) - atan2(5+6*cos(t42),6*sin(t42));

t23 = atan2(z3,y3);
D3 = (x3^2+y3^2+z3^2-61)/60;
t43 = atan2(-(1-D3^2)^0.5,D3);
t33 = atan2((y3^2+z3^2)^0.5,x3-5) - atan2(5+6*cos(t43),6*sin(t43));

t24 = atan2(z4,y4);
D4 = (x4^2+y4^2+z4^2-61)/60;
t44 = atan2(-(1-D4^2)^0.5,D4);
t34 = atan2((y4^2+z4^2)^0.5,x4-5) - atan2(5+6*cos(t44),6*sin(t44));

t25 = atan2(z5,y5);
D5 = (x5^2+y5^2+z5^2-61)/60;
t45 = atan2(-(1-D5^2)^0.5,D5);
t35 = atan2((y5^2+z5^2)^0.5,x5-5) - atan2(5+6*cos(t45),6*sin(t45));

t26 = atan2(z6,y6);
D6 = (x6^2+y6^2+z6^2-61)/60;
t46 = atan2(-(1-D6^2)^0.5,D6);
t36 = atan2((y6^2+z6^2)^0.5,x6-5) - atan2(5+6*cos(t46),6*sin(t46));
```

A.5 MARSlimbCONTROL

```
function[Even, Even2, Odd, Odd2, contact] = MARSlimbCONTROL(even,
odd, ux,uy,uz, r, h, steph, step, rx,ry,rz, contact, buffer)

% ########################################################################
%
% This function is the overal control of limb tip position and limb
% switching for the gate planing algorythem.
%
% Inputs:  even  the 3X3 matrix of (x,y,z) limb tip positions (even limbs)
%             [x2 y2  z2]
%             [x4 y4  z4]
%             [x6 y6  z6]
%
%          odd    the 3X3 matrix of (x,y,z) limb tip positions (odd limbs)
%             [x1 y1  z1]
%             [x3 y3  z3]
%             [x5 y5  z5]
%
```

```
%          (ux,uy,uz) the direction of the robot
%
%          r  the radius of the z-axis sigularity buffer cylinder
%
%          h  the walking hieght of the robot
%
%          steph  the steping hieght of the robot
%
%          step   the distance of travel for one increment
%
%          (rx,ry,rz) the rotation velocities (incremented rotation amounts)
%                     for rotation about the given axes
%
%          contact = 1 if odd limbs are in contact 2 for even
%
%          buffer = the distance the limb tip must stay away from the
%                   workspace shell
%
%  Outputs:   Even    the 3X3 matrix of (x,y,z) even limb tip positions
%             Odd     the 3X3 matrix of (x,y,z) odd limb tip positions
%             Even2   the 3X3 matrix of (x,y,z) even limb tip positions
%                     for use when switching limbs (two movements are
%                     required)
%             Odd2    the 3X3 matrix of (x,y,z) odd limb tip positions
%                     for use when switching limbs (two movements are
%                     required)
%
%                     Note: Even2 and Odd2 are empty when not switching
%
%             contact = 1 if odd limbs are in contact, 2 for even
%
%             buffer  the distance the limb tips must stay away from the
%                     ends of the stride-lines
%
%  #######################################################################
% This finds the total tip translation due to body rotations
% [dUodd2, dUeven2] = MARStotalTIPtranslation(rx,ry,rz, ux,uy,uz, step, odd, even);
[dUodd2, dUeven2] = MARSlimbFRAMEdirection(ux,uy,uz);

% ########################################################################
% Find stride-line for each limb
[ConOdd, ConEven, NonOdd, NonEven]=MARSfindSTRIDEline(contact, odd,even,
dUodd2,dUeven2, r,h,steph);

% ########################################################################
% Arange stride-line so that the (x1,y1,z1) point is in the forward
% direction of the robot
% [ConOdd,ConEven,NonOdd,NonEven]=
%MARSdirectionSTRIDEline(ConOdd,ConEven,NonOdd,NonEven, odd,even);
[ConOdd,NonOdd,ConEven,NonEven]=MARSdirectionCHECK(dUodd2,dUeven2, ConOdd,NonOdd,ConEven,NonEven);

% ########################################################################
% The non-contact limb tip is not necesaraly at a point that lies on the
% optimized non-contact stride-line.  For this reason the non-contact stride-line
% must be redefined as the line between the limb tip and the starting point
% of the optimised stride-line.  Further, non-contact limbs move in the
% oposite direction of contact limbs
```

```
[NonOdd, NonEven]=MARSnonContactSTRIDEline(contact, NonOdd, odd, NonEven, even);

% ########################################################################
% Now the non-contact directional vector must be found and multiplied by
% the step size
[dUodd, dUeven]=MARSnoncontactSTEPvector(contact, step, NonOdd, NonEven, dUodd2, dUeven2);

% ########################################################################
% Find next limb tip position
[dPodd, dPeven]=MARSnextTIPpos(odd, even, dUodd, dUeven)

% ########################################################################
% Check to see if switch is necesary.  If a contact-noncontact limb switch
% in necesary, change=1
[change]=MARSlimbSWITCHtest(dPodd, dPeven, ConOdd, ConEven, contact, buffer);

% ########################################################################
% Output the limb tip positions.  If there is no switch only one position
% will be output for each limb.  However, if a limb switch is necesary two
% positions will be output for each limb
%[Even, Odd, Even2, Odd2]=MARStipPOSoutput(dPodd, dPeven, change, dUodd2, dUeven2, contact, r, h, steph);
Even=dPeven;
Odd=dPodd;
Even2=[];
Odd2=[];
```

A.6 MARStotalTIPtranslation

```
function[dUodd2, dUeven2] = MARStotalTIPtranslation(rx,ry,rz, ux,uy,uz, step, odd, even)

% #######################################################################
%
% This function finds the limb tip translation due to body rotation and
% translation
%
% Inputs:
%           (rx,ry,rz) the rotation velocities (incremented rotation
%                       amounts for rotation about the given axes
%
%           (ux,uy,uz) the direction of the robot
%
%           steph  the steping hieght of the robot
%
% Outputs:  dUodd2 and dUeven2 are the change in limb tip translation due to
%                     body rotation.  They are seperated by even and odd
%                     limbs.  They are stored in arrays of the form:
%
%                             x-direction y-direction z-direction
%                     limb1   value       value       value
%                     limb3   value       value       value
%                     limb4   value       value       value
%
% #######################################################################
dU=[];
```

```
P = [odd(1,:);even(1,:);odd(2,:);even(2,:);odd(3,:);even(3,:)];
% P is a 6X3 array containing all 6 limb tip positions is (x,y,z) rows
for i=1:6
    % ####################################################################
    % Find Translation in limb cordinates due to body rotations
    % find rotation matricies
    Rx =   [1   0        0          0;
            0   cos(rx)  sin(rx)     0;
            0   -sin(rx) cos(rx)     0;
            0   0        0          1];
    Ry =   [cos(ry)     0   sin(ry) 0;
            0           1   0       0;
            -sin(ry)    0   cos(ry) 0;
            0           0   0       1];
    Rz =   [cos(rz-(6-i)*(60*pi/180))   -sin(rz-(6-i)*(60*pi/180))  0   0;
            sin(rz-(6-i)*(60*pi/180))   cos(rz-(6-i)*(60*pi/180))   0   0;
            0                           0                           1   0;
            0                           0                           0   1];
    % find translation to center of proximal joint
    T = [0,0,0,0;
         0,0,0,7.41;
         0,0,0,0;
         0,0,0,1];
    % find location of center of proximal joint based on rotations
    H = Rx*Ry*Rz*T;
    % find rotation of each limb clockwize
    RRz =  [cos(-(6-i)*(60*pi/180))   -sin(-(6-i)*(60*pi/180))  0   0;
            sin(-(6-i)*(60*pi/180))   cos(-(6-i)*(60*pi/180))   0   0;
            0                         0                         1   0;
            0                         0                         0   1];
    % find rotation of each limb counter-clockwize
    RRz2 =  [cos((6-i)*(60*pi/180))    -sin((6-i)*(60*pi/180))  0   0;
             sin((6-i)*(60*pi/180))    cos((6-i)*(60*pi/180))   0   0;
             0                         0                        1   0;
             0                         0                        0   1];
    % find translation for each limb tip
    A = [0,0,0,P(i,1);
         0,0,0,P(i,2);
         0,0,0,P(i,3);
         0,0,0,1];
    % find limb tip translation due to rotations based on body cordinates
    Tt2 = H+RRz*A;
    % find limb tip translation before rotations based on body cordinates
    Tt1 = RRz*(A+T);
    % find displacement vector due to rotation in terms of body cordinates
    dtB = Tt2 - Tt1;
    % find displacement vector due to rotation in terms of limb cordinates
    dtL = RRz2 * dtB;
    % ####################################################################
    % Find Total limb tip translation in limb cordinates
    % find the translation vector due to rotation in terms of the Limb cordinates
    uvB = [0,0,0,ux;  % unit vector (ux,uy,uz)
          0,0,0,uy;
          0,0,0,uz;
          0,0,0,1];
    uvL = RRz * uvB;  % unit vector adjusted to limb frame
    % TL=the sum of the displacement due to the rotations and the unit
```

```
    % vector (ux,uy,uz) times the step size
    TL = [dtL(1,4),dtL(2,4),dtL(3,4)] + step*[uvL(1,4),uvL(2,4),uvL(3,4)];
    dU = [dU;TL];
end
% Split dU into even and odd limb groups
dUodd2 = [dU(1,:);dU(3,:);dU(5,:)];
dUeven2 = [dU(2,:);dU(4,:);dU(6,:)];
```

A.7 MARSlimbFRAMEdirection

```
function[dUodd2, dUeven2] = MARSlimbFRAMEdirection(ux,uy,uz)

% ######################################################################
%
% This function translates the body corditate direction vector (ux,uy,uz)
% to each limb cordinate frame
%
% Inputs:  (ux,uy,uz) the direction of the robot
%
%           steph  the steping hieght of the robot
%
% Outputs:  dUodd2 and dUeven2 are the change in limb tip translation due to
%                     body translation.  They are seperated by even and odd
%                     limbs.  They are stored in arrays of the form:
%
%                             x-direction y-direction z-direction
%                     limb1   value       value       value
%                     limb3   value       value       value
%                     limb5   value       value       value
%
% ######################################################################

% set up the two output matricies
% Note: the z terms will remain zero and no vertical translation will be
% tacken into acount
dUodd2 = zeros(3,3);
dUeven2 = zeros(3,3);

% Fill in the translations for limb-1 which is oriented in the positive
% y-direction and limb-4 which is oriented in the negative y-direction
% limb-1
dUodd2(1,1) = ux;
dUodd2(1,2) = uy;
% limb-4
dUeven2(2,1) = -ux;
dUeven2(2,2) = -uy;

% The limbs are aranged symmetricaly every 60 degrees around the body of the
% robot
% now the respective anle of rotation about the z-axis for each limb is
% specified
t2=30*pi/180 - 90*pi/180;
t3=330*pi/180 - 90*pi/180;
t5=210*pi/180 - 90*pi/180;
t6=150*pi/180 - 90*pi/180;
```

```
% now the ux and uy translations from body frame to limb frame are made for
% limbs 2,3,5,6
% limb 2
v = [ux; uy; 0];
Rz = [cos(t2), -sin(t2), 0;
      sin(t2), cos(t2),  0;
      0,       0,        0];
new = Rz*v;
dUeven2(1,1) = new(1,1);
dUeven2(1,2) = new(2,1);
% limb 3
Rz = [cos(t3), -sin(t3), 0;
      sin(t3), cos(t3),  0;
      0,       0,        0];
new = Rz*v;
dUodd2(2,1) = new(1,1);
dUodd2(2,2) = new(2,1);
% limb 5
Rz = [cos(t5), -sin(t5), 0;
      sin(t5), cos(t5),  0;
      0,       0,        0];
new = Rz*v;
dUodd2(3,1) = new(1,1);
dUodd2(3,2) = new(2,1);
% limb 6
Rz = [cos(t6), -sin(t6), 0;
      sin(t6), cos(t6),  0;
      0,       0,        0];
new = Rz*v;
dUeven2(3,1) = new(1,1);
dUeven2(3,2) = new(2,1);

% now the direction vectors in the limb frames are made into unit vectors

for i=1:3
    if (sum(dUodd2(i,:).^2))^0.5 ~= 0
    dUodd2(i,:) = dUodd2(i,:)./(sum(dUodd2(i,:).^2))^0.5;
    end
    if (sum(dUeven2(i,:).^2))^0.5 ~= 0
    dUeven2(i,:) = dUeven2(i,:)./(sum(dUeven2(i,:).^2))^0.5;
    end
end
```

A.8 MARSfindSTRIDEline

```
function[ConOdd, ConEven, NonOdd, NonEven]=MARSfindSTRIDEline(contact, odd,even,dUodd2,dUeven2, r,h,steph)

% #######################################################################
%
% This function uses the functions MARScontactLINE and MARSnoncontactLINE
% to find the endponts of stride-lines.  These stride-line endpoints are
% ouput in arays depending on which limb they corespond to and wheather or
% not the limb they correspond to is in contact or not in contact.  The
```

```
% format of these arays in explained bellow in ``Outputs."
%
% Inputs:  contact = 1 if odd limbs are in contact, 2 for even
%          odd    the 3X3 matrix of (x,y,z) limb tip positions (odd limbs)
%             [x1 y1  z1]
%             [x3 y3  z3]
%             [x5 y5  z5]
%          even  the 3X3 matrix of (x,y,z) limb tip positions (even limbs)
%             [x2 y2  z2]
%             [x4 y4  z4]
%             [x6 y6  z6]
%          dUodd2 and dUeven2 are the change in limb tip translation due to
%                     body rotation.  They are seperated by even and odd
%                     limbs.  They are stored in arrays of the form:
%
%                             x-direction y-direction z-direction
%                     limb1   value       value       value
%                     limb3   value       value       value
%                     limb4   value       value       value
%          r  the radius of the z-axis sigularity buffer cylinder
%          h  the walking hieght of the robot
%          steph  the steping hieght of the robot
%
% Outputs:    The outputs of this function are arays of points
%             Each row of the aray is six elements long.  The first three
%             elements are the (x,y,z) position of one point.  The last
%             three elements are the (x,y,z) position of another point.
%             These two points are the end points of a stride-line.
%
%             Each limb has two points associated with the limbs stride
%             line.  If the ``odd'' limbs are in contact and the ``even"
%             limbs are not in contact the outputs will be:
%
%             ConOdd =    [limb-1 points
%                          limb-3 points
%                          limb-5 points]
%             ConEven =   []
%             NonOdd =    []
%             NonEven =   [limb-2 points
%                          limb-4 points
%                          limb-6 points]
%
%             If the ``even'' limbs are in contact and the ``odd'' limbs
%             are not in contact the outputs will be:
%
%             ConOdd =    []
%             ConEven =   [limb-2 points
%                          limb-4 points
%                          limb-6 points]
%             NonOdd =    [limb-1 points
%                          limb-3 points
%                          limb-5 points]
%             NonEven =   []
%
% #######################################################################

% Find stride-line for each limb
```

```
% initialize
ConOdd=[]; ConEven=[]; NonOdd=[]; NonEven=[];
if contact == 1
    for i=1:3
        [x1,y1,z1,x2,y2,z2] = MARScontactLINE(odd(i,1),odd(i,2),odd(i,3),dUodd2(i,1),dUodd2(i,2),dUodd2(i,3),r);
        ConOdd = [ConOdd; x1,y1,z1,x2,y2,z2];
        [x1,y1,z1,x2,y2,z2] = MARSnoncontactLINE(dUeven2(i,1),dUeven2(i,2),dUeven2(i,3),r,h-steph);
        NonEven = [NonEven; x1,y1,z1,x2,y2,z2];
    end
else
    for i=1:3
        [x1,y1,z1,x2,y2,z2] = MARScontactLINE(even(i,1),even(i,2),even(i,3),dUeven2(i,1),dUeven2(i,2),dUeven2(i,3)
        ConEven = [ConEven; x1,y1,z1,x2,y2,z2];
        [x1,y1,z1,x2,y2,z2] = MARSnoncontactLINE(dUodd2(i,1),dUodd2(i,2),dUodd2(i,3),r,h-steph);
        NonOdd = [NonOdd; x1,y1,z1,x2,y2,z2];
    end
end
```

A.9 MARSdirectionSTRIDEline

```
function[ConOdd,ConEven,NonOdd,NonEven]=MARSdirectionSTRIDEline(ConOdd,ConEven,NonOdd,NonEven, odd,even)

% ######################################################################
%
% This function aranges the points which describe a stride-line
% so that the first point is in the forward direction of the limb
% tip motion.
%
% Inputs:     The inputs of this function are arays of points
%             Each row of the aray is six elements long.  The first three
%             elements are the (x,y,z) position of one point.  The last
%             three elements are the (x,y,z) position of another point.
%             These two points are the end points of a stride-line.
%
%             Each limb has two points associated with the limbs stride
%             line.  If the ``odd'' limbs are in contact and the ``even"
%             limbs are not in contact the outputs will be:
%
%             ConOdd =    [limb-1 points
%                          limb-3 points
%                          limb-5 points]
%             ConEven =   []
%             NonOdd =    []
%             NonEven =   [limb-2 points
%                          limb-4 points
%                          limb-6 points]
%
%             If the ``even'' limbs are in contact and the ``odd'' limbs
%             are not in contact the outputs will be:
%
%             ConOdd =    []
%             ConEven =   [limb-2 points
%                          limb-4 points
%                          limb-6 points]
%             NonOdd =    [limb-1 points
```

```
%                           limb-3 points
%                           limb-5 points]
%             NonEven =   []
%
%
%             even  the 3X3 matrix of (x,y,z) limb tip positions (even limbs)
%                 [x2 y2  z2]
%                 [x4 y4  z4]
%                 [x6 y6  z6]
%
%             odd    the 3X3 matrix of (x,y,z) limb tip positions (odd limbs)
%                 [x1 y1  z1]
%                 [x3 y3  z3]
%                 [x5 y5  z5]
%
% Outputs:    ConOdd, ConEven, NonOdd, NonEven--rearanged as necesary
%
% #####################################################################

% Arange stride-line so that the (x1,y1,z1) point is in the forward
% direction of the robot
if ~(isempty(ConOdd))           % test to make sure the array is not empty
    for i=1:3                   % iterates through the three limbs in the array
        if ConOdd(i,2)-odd(i,2) ~= 0        % test if there is displacement in the y-direction
            if ConOdd(i,2)-odd(i,2) <= 0    % test if the displacement in the y-direction is negative
                store = [ConOdd(i,4:6)];
                ConOdd(i,:) = [store,ConOdd(i,1:3)];    % switch the points if displacement is negative
            end
        elseif ConOdd(i,1)-odd(i,1) <= 0    % otherwise test if the displacement in the x-direction is negative
            store = [ConOdd(i,4:6)];
            ConOdd(i,:) = [store,ConOdd(i,1:3)];        % switch the points if displacement is negative
        end
    end
end
if ~(isempty(NonOdd))
    for i=1:3
        if NonOdd(i,2)-odd(i,2) ~= 0
            if NonOdd(i,2)-odd(i,2) <= 0
                store = [NonOdd(i,4:6)];
                NonOdd(i,:) = [store,NonOdd(i,1:3)];
            end
        elseif NonOdd(i,1)-odd(i,1) <= 0
            store = [NonOdd(i,4:6)];
            NonOdd(i,:) = [store,NonOdd(i,1:3)];
        end
    end
end
if ~(isempty(NonEven))
    for i=1:3
        if NonEven(i,2)-even(i,2) ~= 0
            if NonEven(i,2)-even(i,2) <= 0
                store = [NonEven(i,4:6)];
                NonEven(i,:) = [store,NonEven(i,1:3)];
            end
        elseif NonEven(i,1)-even(i,1) <= 0
            store = [NonEven(i,4:6)];
            NonEven(i,:) = [store,NonEven(i,1:3)];
```

```
        end
    end
end
if ~(isempty(ConEven))
    for i=1:3
        if ConEven(i,2)-even(i,2) ~= 0
            if ConEven(i,2)-even(i,2) <= 0
                store = [ConEven(i,4:6)];
                ConEven(i,:) = [store,ConEven(i,1:3)];
            end
        elseif ConEven(i,1)-even(i,1) <= 0
            store = [ConEven(i,4:6)];
            ConEven(i,:) = [store,ConEven(i,1:3)];
        end
    end
end
```

A.10 MARSdirectionCHECK

```
function[ConOdd,NonOdd,ConEven,NonEven]=MARSdirectionCHECK(dUodd2,dUeven2, ConOdd,NonOdd,ConEven,NonEven)

% ######################################################################
% becouse the robot walks with an alternating tripedal gate 3 limbs will be
% in contact with the surface and 3 limbs will be in non-contact with the
% surface.  Becouse the limbs are brocken in to the even and odd groups
% there are 2 posabilites:
%
% Possability 1: Odd limbs in contact and even limbs in non-contact
%
% Possability 2: Even limbs in contact and odd limbs in non-contact
%
% The limb tip stride-line end-points are stored in the arrays: ConOdd,
% NonOdd, ConEven, and NonEven.  Wheather even or odd limbs are in contact
% can be determined by finding if these arays are empty.  For example: if
% the odd limbs are in contact than the ConOdd array will hold the stride
% line end-point values and the NonOdd array will be empty.
%
% Inputs:  dUodd2 and dUeven2 are the change in limb tip translation
%                    vectors due to
%                    body translation.  They are seperated by even and odd
%                    limbs.  They are stored in arrays of the form:
%
%                            x-direction y-direction z-direction
%                    limb1   value       value       value
%                    limb3   value       value       value
%                    limb5   value       value       value
%
%            The other inputs of this function are arays of points
%            Each row of the aray is six elements long.  The first three
%            elements are the (x,y,z) position of one point.  The last
%            three elements are the (x,y,z) position of another point.
%            These two points are the end points of a stride-line.
%
%            Each limb has two points associated with the limbs stride
%            line.  If the ``odd'' limbs are in contact and the ``even"
```

```
%             limbs are not in contact the outputs will be:
%
%             ConOdd =    [limb-1 points
%                          limb-3 points
%                          limb-5 points]
%             ConEven =   []
%             NonOdd =    []
%             NonEven =   [limb-2 points
%                          limb-4 points
%                          limb-6 points]
%
%             If the ``even'' limbs are in contact and the ``odd'' limbs
%             are not in contact the outputs will be:
%
%             ConOdd =    []
%             ConEven =   [limb-2 points
%                          limb-4 points
%                          limb-6 points]
%             NonOdd =    [limb-1 points
%                          limb-3 points
%                          limb-5 points]
%             NonEven =   []
%
% Outputs:    The corectly oriented: ConOdd, NonOdd, ConEven, and NonEven
%               the output arrays are aranged so that the first 3 values of
%               a row represent the end point of the stride-line in the
%               forward direction of the body direction
%
% ######################################################################

% check to see in the odd limbs are in contact (the ConOdd aray contains
% values)
if ~(isempty(ConOdd))
    % so the odd limbs are in contact.  Now we check through the ConOdd and
    % NonEven arrays to make sure the strides lines are pointed in the same
    % direction as the forward direction of the robot.  This direction,
    % translated to each limb cordinate frame is given by the dUodd2 and
    % dUeven2 arays.
    for i=1:3
        % Now we test to see if the vectors are pointed in the same
        % direction.  To do this we test the sign of the dot product.  If
        % the sign is negative we switch the Con or Non points around.
        V1 = [(ConOdd(i,1)-ConOdd(i,4)), (ConOdd(i,2)-ConOdd(i,5)), (ConOdd(i,3)-ConOdd(i,6))];
        V2 = dUodd2(i,:);
        if sign(dot(V1,V2)) < 0
            ConOdd(i,:) = [ConOdd(i,4:6),ConOdd(i,1:3)];
        end
        V1 = [(NonEven(i,1)-NonEven(i,4)), (NonEven(i,2)-NonEven(i,5)), (NonEven(i,3)-NonEven(i,6))];
        V2 = dUeven2(i,:);
        if sign(dot(V1,V2)) < 0
            NonEven(i,:) = [NonEven(i,4:6),NonEven(i,1:3)];
        end
    end
end
% and now the same thing for when the even lombs are in contact
if ~(isempty(ConEven))
    for i=1:3
```

```
        V1 = [(NonOdd(i,1)-NonOdd(i,4)), (NonOdd(i,2)-NonOdd(i,5)), (NonOdd(i,3)-NonOdd(i,6))];
        V2 = dUodd2(i,:);
        if sign(dot(V1,V2)) < 0
            NonOdd(i,:) = [NonOdd(i,4:6),NonOdd(i,1:3)];
        end
        V1 = [(ConEven(i,1)-ConEven(i,4)), (ConEven(i,2)-ConEven(i,5)), (ConEven(i,3)-ConEven(i,6))];
        V2 = dUeven2(i,:);
        if sign(dot(V1,V2)) < 0
            ConEven(i,:) = [ConEven(i,4:6),ConEven(i,1:3)];
        end
    end
end
```

A.11 MARSnonContactSTRIDEline

```
function[NonOdd, NonEven]=MARSnonContactSTRIDEline(contact, NonOdd, odd, NonEven, even)

% ########################################################################
% The non-contact limb tip is not necesaraly at a point that lies on the
% optimized non-contact stride-line.  For this reason the non-contact stride-line
% must be redefined as the line between the limb tip and the starting point
% of the optimised stride-line.  Further, non-contact limbs move in the
% oposite direction of contact limbs
%
% Inputs:     The inputs of this function are the ‘‘Non’’ arays of points
%             Each row of the aray is six elements long.  The first three
%             elements are the (x,y,z) position of one point.  The last
%             three elements are the (x,y,z) position of another point.
%             These two points are the end points of a stride-line.
%
%             Each limb has two points associated with the limbs stride
%             line.  If the ‘‘odd’’ limbs are in contact and the ‘‘even"
%             limbs are not in contact the outputs will be:
%
%             ConOdd =    [limb-1 points
%                          limb-3 points
%                          limb-5 points]
%             ConEven =   []
%             NonOdd =    []
%             NonEven =   [limb-2 points
%                          limb-4 points
%                          limb-6 points]
%
%             If the ‘‘even’’ limbs are in contact and the ‘‘odd’’ limbs
%             are not in contact the outputs will be:
%
%             ConOdd =    []
%             ConEven =   [limb-2 points
%                          limb-4 points
%                          limb-6 points]
%             NonOdd =    [limb-1 points
%                          limb-3 points
%                          limb-5 points]
%             NonEven =   []
%
```

```
%
%             even  the 3X3 matrix of (x,y,z) limb tip positions (even limbs)
%                 [x2 y2  z2]
%                 [x4 y4  z4]
%                 [x6 y6  z6]
%
%             odd    the 3X3 matrix of (x,y,z) limb tip positions (odd limbs)
%                 [x1 y1  z1]
%                 [x3 y3  z3]
%                 [x5 y5  z5]
%
%             contact = 1 if odd limbs are in contact 2 for even
%
% Outputs:    NonOdd, NonEven
%
% ########################################################################

if contact == 2
    if ~(isempty(NonOdd))
        for i=1:3
            NonOdd(i,:) = [NonOdd(i,4:6),odd(i,1:3)];
        end
    end
else
    if ~(isempty(NonEven))
        for i=1:3
            NonEven(i,:) = [NonEven(i,4:6),even(i,1:3)];
        end
    end
end
```

A.12 MARSnoncontactSTEPvector

```
function[dUodd, dUeven]=MARSnoncontactSTEPvector(contact, step, NonOdd, NonEven, dUodd2, dUeven2)

% ######################################################################
% Now the non-contact directional vector must be found and multiplied by
% the step size
%
% Inputs:     The inputs of this function are the ‘‘Non’’ arays of points
%             Each row of the aray is six elements long.  The first three
%             elements are the (x,y,z) position of one point.  The last
%             three elements are the (x,y,z) position of another point.
%             These two points are the end points of a stride-line.
%
%             Each limb has two points associated with the limbs stride
%             line.  If the ‘‘odd’’ limbs are in contact and the ‘‘even"
%             limbs are not in contact the outputs will be:
%
%             ConOdd =    [limb-1 points
%                          limb-3 points
%                          limb-5 points]
%             ConEven =   []
%             NonOdd =    []
%             NonEven =   [limb-2 points
```

```
%                           limb-4 points
%                           limb-6 points]
%
%             If the ‘‘even’’ limbs are in contact and the ‘‘odd’’ limbs
%             are not in contact the outputs will be:
%
%             ConOdd =    []
%             ConEven =   [limb-2 points
%                          limb-4 points
%                          limb-6 points]
%             NonOdd =    [limb-1 points
%                          limb-3 points
%                          limb-5 points]
%             NonEven =   []
%
%             step   the distance of travel for one increment
%
%             contact = 1 if odd limbs are in contact 2 for even
%
%             dUodd2 and dUeven2 are the change in limb tip
%                         translation due to body rotation.  They are
%                         seperated by even and odd limbs.  They are
%                         stored in arrays of the form:
%
%                             x-direction y-direction z-direction
%                     limb1   value       value       value
%                     limb3   value       value       value
%                     limb4   value       value       value
%
% Outputs:  dUodd and dUeven are the non-contact directional vectors.
%             They are the correct length as specified by the step input.
%             They are stored in arrays of the form:
%
%                             x-direction y-direction z-direction
%                     limb1   value       value       value
%                     limb3   value       value       value
%                     limb4   value       value       value
%
% ########################################################################

dUodd=dUodd2; dUeven=dUeven2;

if contact == 2
    for i=1:3
        Tn = NonOdd(i,1:3)-NonOdd(i,4:6);
        dUodd(i,:) = step*(Tn./(sum(Tn.^2))^0.5);
    end
else
    for i=1:3
        Tn = NonEven(i,1:3)-NonEven(i,4:6);
        dUeven(i,:) = step*(Tn./(sum(Tn.^2))^0.5);
    end
end
```

A.13 MARSnextTIPpos

```
function[dPodd, dPeven]=MARSnextTIPpos(odd, even, dUodd, dUeven)

% ########################################################################
%
% Find next limb tip position
%
% Inputs:  even  the 3X3 matrix of (x,y,z) limb tip positions (even limbs)
%             [x2 y2  z2]
%             [x4 y4  z4]
%             [x6 y6  z6]
%
%          odd    the 3X3 matrix of (x,y,z) limb tip positions (odd limbs)
%             [x1 y1  z1]
%             [x3 y3  z3]
%             [x5 y5  z5]
%
%          dUodd and dUeven are the non-contact directional vectors.
%             They are the correct length as specified by the step input.
%             They are stored in arrays of the form:
%
%                            x-direction y-direction z-direction
%                    limb1   value       value       value
%                    limb3   value       value       value
%                    limb4   value       value       value
%
% Outputs:    dPodd = the next limb tip position for the odd limbs
%             dPeven = the next limb tip position fo the even limbs
%
% ########################################################################

% initialize
dPodd=[];dPeven=[];
for i=1:3
    dPodd(i,:)=odd(i,:)+dUodd(i,:);
    dPeven(i,:)=even(i,:)+dUeven(i,:);
end
```

A.14 MARSlimbSWITCHtest

```
function[change]=MARSlimbSWITCHtest(dPodd, dPeven, ConOdd, ConEven, contact, buffer)

% #######################################################################
% Check to see if switch is necesary.  If a contact-noncontact limb switch
% in necesary, change=1
%
% Inputs:     The inputs of this function are the ``Con'' arays of points
%             Each row of the aray is six elements long.  The first three
%             elements are the (x,y,z) position of one point.  The last
%             three elements are the (x,y,z) position of another point.
%             These two points are the end points of a stride-line.
%
%             Each limb has two points associated with the limbs stride
```

```
%             line.  If the ``odd'' limbs are in contact and the ``even"
%             limbs are not in contact the outputs will be:
%
%             ConOdd =    [limb-1 points
%                          limb-3 points
%                          limb-5 points]
%             ConEven =   []
%             NonOdd =    []
%             NonEven =   [limb-2 points
%                          limb-4 points
%                          limb-6 points]
%
%             If the ``even'' limbs are in contact and the ``odd'' limbs
%             are not in contact the outputs will be:
%
%             ConOdd =    []
%             ConEven =   [limb-2 points
%                          limb-4 points
%                          limb-6 points]
%             NonOdd =    [limb-1 points
%                          limb-3 points
%                          limb-5 points]
%             NonEven =   []
%
%             dPodd = the next limb tip position for the odd limbs
%             dPeven = the next limb tip position fo the even limbs
%
%             contact = 1 if odd limbs are in contact 2 for even
%
%             buffer = the distance the limb tip must stay away from the
%                      workspace shell
%
% Outputs:    change: ``0'' signifies that there is not limb switch
%                     ``1'' signifies that a limb switch is necesary
%
% #######################################################################

% initialize
change = 0;
if contact == 1
    for i=1:3
        distV = dPodd(i,:)-ConOdd(i,1:3);
        dist = (sum((distV).^2))^0.5;
        if dist <= buffer
            change = 1;
        end
    end
else
    for i=1:3
        distV = dPeven(i,:)-ConEven(i,1:3);
        dist = (sum((distV).^2))^0.5;
        if dist <= buffer
            change = 1;
        end
    end
end
```

A.15 MARStipPOSoutput

```
function[Even, Odd, Even2, Odd2]=MARStipPOSoutput(dPodd, dPeven, change, dUodd2, dUeven2, contact, r, h, steph)

% ########################################################################
% Output the limb tip positions.  If there is no switch only one position
% will be output for each limb.  However, if a limb switch is necesary two
% positions will be output for each limb

% Inputs:     dPodd = the next limb tip position for the odd limbs
%             dPeven = the next limb tip position fo the even limbs

%             change: ``0'' signifies that there is not limb switch
%                     ``1'' signifies that a limb switch is necesary

%             dUodd2 and dUeven2 are the change in limb tip
%                         translation due to body rotation.  They are
%                         seperated by even and odd limbs.  They are
%                         stored in arrays of the form:
%
%                             x-direction y-direction z-direction
%                     limb1   value       value       value
%                     limb3   value       value       value
%                     limb4   value       value       value
%
%             contact = 1 if odd limbs are in contact 2 for even
%
%          r  the radius of the z-axis sigularity buffer cylinder
%
%          h  the walking hieght of the robot
%
%          steph  the steping hieght of the robot
%
% Outputs:    Even = even limb tip possitions
%             Odd = odd limb tip possitions
%             Even2 = the next set of even limb tip positions is a switch is
%                     necesary
%             Odd2 = the next set of odd limb tip positions is a switch is
%                     necesary
%
% ########################################################################

% initialize
Even=[]; Odd=[]; Even2=[]; Odd2=[];
if change == 0
    Odd = dPodd
    Even = dPeven
elseif change == 1
    % switch the change argument
    change = 0;
    if contact == 2
        % position the contact limb to the next point
        Even = dPeven
        % make the non-contact limb come in contact by finding a
        % non-contact line at the contact hieght and moving the limb tip to
        % the start of this line
        for i=1:3
```

```
            [x1,y1,z1,x2,y2,z2] = MARSnoncontactLINE(dUodd2(i,1),dUodd2(i,2),dUodd2(i,3),r,h);
            if dUodd2(i,2)==0
                if sign(dUodd2(i,1))==sign(x1)
                    Odd(i,:)=[x1,y1,z1];
                else
                    Odd(i,:)=[x2,y2,z2];
                end
            else
                if sign(dUodd2(i,2))==sign(y1)
                    Odd(i,:)=[x1,y1,z1];
                else
                    Odd(i,:)=[x2,y2,z2];
                end
            end
        end
        % raise the contact limb so it is non-contact
        for i=1:3
            [x1,y1,z1,x2,y2,z2] = MARSnoncontactLINE(dUeven2(i,1),dUeven2(i,2),dUeven2(i,3),r,h-steph);
            if dUeven2(i,2)==0
                if sign(dUeven2(i,1))==sign(x1)
                    Even2(i,:)=[x1,y1,z1];
                else
                    Even2(i,:)=[x2,y2,z2];
                end
            else
                if sign(dUeven2(i,2))==sign(y1)
                    Even2(i,:)=[x1,y1,z1];
                else
                    Even2(i,:)=[x2,y2,z2];
                end
            end
        end
        % move the non-contact limb to new contact position
        Odd2 = Odd + dUodd2;
    else
        % position the contact limb to the next point
        Odd = dPodd;
        % make the non-contact limb come in contact by finding a
        % non-contact line at the contact hieght and moving the limb tip to
        % the start of this line
        for i=1:3
            [x1,y1,z1,x2,y2,z2] = MARSnoncontactLINE(dUeven2(i,1),dUeven2(i,2),dUeven2(i,3),r,h);
            if dUeven2(i,2)==0
                if sign(dUeven2(i,1))==sign(x1)
                    Even(i,:)=[x1,y1,z1];
                else
                    Even(i,:)=[x2,y2,z2];
                end
            else
                if sign(dUeven2(i,2))==sign(y1)
                    Even(i,:)=[x1,y1,z1];
                else
                    Even(i,:)=[x2,y2,z2];
                end
            end
        end
        % raise the contact limb so it is non-contact
```

```
        for i=1:3
            [x1,y1,z1,x2,y2,z2] = MARSnoncontactLINE(dUodd2(i,1),dUodd2(i,2),dUodd2(i,3),r,h-steph);
            if dUodd2(i,2)==0
                if sign(dUodd2(i,1))==sign(x1)
                    Odd2(i,:)=[x1,y1,z1];
                else
                    Odd2(i,:)=[x2,y2,z2];
                end
            else
                if sign(dUodd2(i,2))==sign(y1)
                    Odd2(i,:)=[x1,y1,z1];
                else
                    Odd2(i,:)=[x2,y2,z2];
                end
            end
        end
        % move the non-contact limb to new contact position
        Even2 = Even + dUeven2;
    end
end
```

A.16 MARSContactLine

```
function[x1,y1,z1,x2,y2,z2] = MARScontactLINE(X,Y,Z,ux,uy,uz,r)

% ####################################################################
% This function finds the longest suitable stride-line for the
% contact MARS limbs
%
% Inputs: (ux,uy,uz)  the direction of the line
%         r   the radius of the z-axis singularity buffer cylinder
%         h   the walking hieght of the robot
%         (X,Y,Z) the point of contact
%
% Outputs: (x1,y1,z1),(x2,y2,z2) the endpoints of the stride-line segment
%
% ####################################################################

% ensure unit-vector
Ux = ux/(ux^2+uy^2+uz^2)^0.5;
Uy = uy/(ux^2+uy^2+uz^2)^0.5;
Uz = uz/(ux^2+uy^2+uz^2)^0.5;
ux=Ux; uy=Uy; uz=Uz;

x1=-12; y1=-12; z1=-12; x2=12; y2=12; z2=12; % initialize

% next, each 3D workspace boundary shell is tested for intersection
% Note, as long as h>Shel-3 radius, intersection with shell 1 is not
% possible
% 000000000000000000000000000000000000000000000000000000000000000000000
[x,y,z] = MARScylInt(X,Y,Z,ux,uy,uz,r);
% find the distances from point to intersects and specifies the distance as
% positive or negative from the point
if ~(isempty(x))
    dD1 = [X,Y,Z] - [x(1),y(1),z(1)];
```

```
    D(1) = (dD1(1)^2 + dD1(2)^2 + dD1(3)^2)^0.5;
    if uy == 0
        if X>dD1(1)
            D(1)=-D(1);
        end
    else
        if Y>dD1(2)
            D(1)=-D(1);
        end
    end

    if length(x) == 2
        dD2 = [X,Y,Z] - [x(2),y(2),z(2)];
        D(2) = (dD2(1)^2 + dD2(2)^2 + dD2(3)^2)^0.5;
        if uy == 0
            if X>dD2(1)
                D(2)=-D(2);
            end
        else
            if Y>dD2(2)
                D(2)=-D(2);
            end
        end
    end

% determine the closest points
    [short, test] = min(abs(D));
    if sign(D(test)) == -1
        x1 = x(test);
        y1 = y(test);
        z1 = z(test);
    else
        x2 = x(test);
        y2 = y(test);
        z2 = z(test);
    end
end
% 111111111111111111111111111111111111111111111111111111111111111111111
[x,y,z] = MARSshell1int(X, Y, Z, ux, uy, uz, 0);
if ~(isempty(x))
    for i=1:length(x)
        if uy==0
            if x(i)<0
                if x(i)>x1
                    x1=x(i);
                    y1=y(i);
                    z1=z(i);
                end
            elseif x(i)>0
                if x(i)<x2
                    x2=x(i);
                    y2=y(i);
                    z2=z(i);
                end
            end
        else
            if y(i)<0
```

```
                if y(i)>y1
                    x1=x(i);
                    y1=y(i);
                    z1=z(i);
                end
            elseif y(i)>0
                if y(i)<y2
                    x2=x(i);
                    y2=y(i);
                    z2=z(i);
                end
            end
        end
    end
end
% 222222222222222222222222222222222222222222222222222222222222222222222222
[x,y,z] = MARSshell2int(X, Y, Z, ux, uy, uz, 0);
if ~(isempty(x))
    for i=1:length(x)
        if uy==0
            if x(i)<0
                if x(i)>x1
                    x1=x(i);
                    y1=y(i);
                    z1=z(i);
                end
            elseif x(i)>0
                if x(i)<x2
                    x2=x(i);
                    y2=y(i);
                    z2=z(i);
                end
            end
        else
            if y(i)<0
                if y(i)>y1
                    x1=x(i);
                    y1=y(i);
                    z1=z(i);
                end
            elseif y(i)>0
                if y(i)<y2
                    x2=x(i);
                    y2=y(i);
                    z2=z(i);
                end
            end
        end
    end
end
% 3333333333333333333333333333333333333333333333333333333333333333333333
[x,y,z] = MARSshell3int(X, Y, Z, ux, uy, uz, 0);
if ~(isempty(x))
    for i=1:length(x)
        if uy==0
            if x(i)<0
                if x(i)>x1
```

```
                    x1=x(i);
                    y1=y(i);
                    z1=z(i);
                end
            elseif x(i)>0
                if x(i)<x2
                    x2=x(i);
                    y2=y(i);
                    z2=z(i);
                end
            end
        else
            if y(i)<0
                if y(i)>y1
                    x1=x(i);
                    y1=y(i);
                    z1=z(i);
                end
            elseif y(i)>0
                if y(i)<y2
                    x2=x(i);
                    y2=y(i);
                    z2=z(i);
                end
            end
        end
    end
end
% 444444444444444444444444444444444444444444444444444444444444444444444
[x,y,z] = MARSshell4int(X, Y, Z, ux, uy, uz, 0);
if ~(isempty(x))
    for i=1:length(x)
        if uy==0
            if x(i)<0
                if x(i)>x1
                    x1=x(i);
                    y1=y(i);
                    z1=z(i);
                end
            elseif x(i)>0
                if x(i)<x2
                    x2=x(i);
                    y2=y(i);
                    z2=z(i);
                end
            end
        else
            if y(i)<0
                if y(i)>y1
                    x1=x(i);
                    y1=y(i);
                    z1=z(i);
                end
            elseif y(i)>0
                if y(i)<y2
                    x2=x(i);
                    y2=y(i);
```

```
                    z2=z(i);
                end
            end
        end
    end
end
% P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1
[x, y, z] = MARSplane1Int(X, Y, Z, ux, uy, uz);
if ~(isempty(x))
    for i=1:length(x)
        if uy==0
            if x(i)<0
                if x(i)>x1
                    x1=x(i);
                    y1=y(i);
                    z1=z(i);
                end
            elseif x(i)>0
                if x(i)<x2
                    x2=x(i);
                    y2=y(i);
                    z2=z(i);
                end
            end
        else
            if y(i)<0
                if y(i)>y1
                    x1=x(i);
                    y1=y(i);
                    z1=z(i);
                end
            elseif y(i)>0
                if y(i)<y2
                    x2=x(i);
                    y2=y(i);
                    z2=z(i);
                end
            end
        end
    end
end
% P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2
[x, y, z] = MARSplane2Int(X, Y, Z, ux, uy, uz);
if ~(isempty(x))
    for i=1:length(x)
        if uy==0
            if x(i)<0
                if x(i)>x1
                    x1=x(i);
                    y1=y(i);
                    z1=z(i);
                end
            elseif x(i)>0
                if x(i)<x2
                    x2=x(i);
                    y2=y(i);
                    z2=z(i);
```

```
                end
            end
        else
            if y(i)<0
                if y(i)>y1
                    x1=x(i);
                    y1=y(i);
                    z1=z(i);
                end
            elseif y(i)>0
                if y(i)<y2
                    x2=x(i);
                    y2=y(i);
                    z2=z(i);
                end
            end
        end
    end
end
```

A.17 MARSNonContactLine

5

```
function[x1,y1,z1,x2,y2,z2] = MARSnoncontactLINE(ux,uy,uz,r,h)

% ####################################################################
% This function finds the longest suitable stride-line for the
% non-contact MARS limbs
%
% Inputs: (ux,uy,uz)  the direction of the line
%         r   the radius of the z-axis singularity buffer cylinder
%         h   the walking hieght of the robot
%
% Outputs: (x1,y1,z1),(x2,y2,z2) the endpoints of the stride-line segment
%
% ####################################################################

% ensure unit-vector
Ux = ux/(ux^2+uy^2+uz^2)^0.5;
Uy = uy/(ux^2+uy^2+uz^2)^0.5;
Uz = uz/(ux^2+uy^2+uz^2)^0.5;
ux=Ux; uy=Uy; uz=Uz;

% first the tangent point to the z-axis singularity buffer cylinar is
% found
[X, Y, Z, Ux, Uy, b, theta] = MARStanpoint(ux,uy,uz,r,h);
% is the
if (Z>61^0.5)|(Z<11)
    [X, Y, Z] = MARStanpointALT(Ux, Uy, theta, b, r);
end

x1=-12; y1=-12; z1=-12; x2=12; y2=12; z2=12; % initialize

% next, each 3D workspace boundary shell is tested for intersection
```

```
% Note, as long as h>Shel-3 radius, intersection with shell 1 is not
% possible
% 1111111111111111111111111111111111111111111111111111111111111111111111
[x,y,z] = MARSshell1int(X, Y, Z, ux, uy, uz, 0);
if ~(isempty(x))
    for i=1:length(x)
        if uy==0
            if x(i)<0
                if x(i)>x1
                    x1=x(i);
                    y1=y(i);
                    z1=z(i);
                end
            elseif x(i)>0
                if x(i)<x2
                    x2=x(i);
                    y2=y(i);
                    z2=z(i);
                end
            end
        else
            if y(i)<0
                if y(i)>y1
                    x1=x(i);
                    y1=y(i);
                    z1=z(i);
                end
            elseif y(i)>0
                if y(i)<y2
                    x2=x(i);
                    y2=y(i);
                    z2=z(i);
                end
            end
        end
    end
end
% 222222222222222222222222222222222222222222222222222222222222222222222222
[x,y,z] = MARSshell2int(X, Y, Z, ux, uy, uz, 0);
if ~(isempty(x))
    for i=1:length(x)
        if uy==0
            if x(i)<0
                if x(i)>x1
                    x1=x(i);
                    y1=y(i);
                    z1=z(i);
                end
            elseif x(i)>0
                if x(i)<x2
                    x2=x(i);
                    y2=y(i);
                    z2=z(i);
                end
            end
        else
            if y(i)<0
```

```
                if y(i)>y1
                    x1=x(i);
                    y1=y(i);
                    z1=z(i);
                end
            elseif y(i)>0
                if y(i)<y2
                    x2=x(i);
                    y2=y(i);
                    z2=z(i);
                end
            end
        end
    end
end
% 4444444444444444444444444444444444444444444444444444444444444444444444
[x,y,z] = MARSshell4int(X, Y, Z, ux, uy, uz, 0);
if ~(isempty(x))
    for i=1:length(x)
        if uy==0
            if x(i)<0
                if x(i)>x1
                    x1=x(i);
                    y1=y(i);
                    z1=z(i);
                end
            elseif x(i)>0
                if x(i)<x2
                    x2=x(i);
                    y2=y(i);
                    z2=z(i);
                end
            end
        else
            if y(i)<0
                if y(i)>y1
                    x1=x(i);
                    y1=y(i);
                    z1=z(i);
                end
            elseif y(i)>0
                if y(i)<y2
                    x2=x(i);
                    y2=y(i);
                    z2=z(i);
                end
            end
        end
    end
end
% P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1P1
[x, y, z] = MARSplane1Int(X, Y, Z, ux, uy, uz);
if ~(isempty(x))
    for i=1:length(x)
        if uy==0
            if x(i)<0
                if x(i)>x1
```

```
                    x1=x(i);
                    y1=y(i);
                    z1=z(i);
                end
            elseif x(i)>0
                if x(i)<x2
                    x2=x(i);
                    y2=y(i);
                    z2=z(i);
                end
            end
        else
            if y(i)<0
                if y(i)>y1
                    x1=x(i);
                    y1=y(i);
                    z1=z(i);
                end
            elseif y(i)>0
                if y(i)<y2
                    x2=x(i);
                    y2=y(i);
                    z2=z(i);
                end
            end
        end
    end
end
% P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2
[x, y, z] = MARSplane2Int(X, Y, Z, ux, uy, uz);
if ~(isempty(x))
    for i=1:length(x)
        if uy==0
            if x(i)<0
                if x(i)>x1
                    x1=x(i);
                    y1=y(i);
                    z1=z(i);
                end
            elseif x(i)>0
                if x(i)<x2
                    x2=x(i);
                    y2=y(i);
                    z2=z(i);
                end
            end
        else
            if y(i)<0
                if y(i)>y1
                    x1=x(i);
                    y1=y(i);
                    z1=z(i);
                end
            elseif y(i)>0
                if y(i)<y2
                    x2=x(i);
                    y2=y(i);
```

```
                    z2=z(i);
                end
            end
        end
    end
end
```

A.18 MARSCylInt

5

```
function[X, Y, Z] = MARScylInt(x,y,z,ux,uy,uz,r)

% ####################################################################
%
% this function finds the intersection points of a line and a
% sylindar allong the z-axis
%
% Inputs: (x,y,z) point on the line
%         (ux, uy, uz) direction vector of the line
%         r   radius of sylindar
%
% Outputs:    (X,Y,Z) intersection points
%
% ####################################################################

% First the line-circle intersect is found in the x-y perspective
% defining terms

ur = (ux^2 + uy^2)^0.5;
x2 = x+ux;
y2 = y+uy;
D = x*y2 - x2*y;
test = r^2*ur^2-D^2;
sgn = 1;
X=[]; Y=[]; Z=[];
% find out if line intersects
if test>0
    % find sign
    if uy < 0
        sgn = -1;
    end
    % find X and Y
    X(1) = (D*uy + sgn*ux*(test)^0.5)/ur^2;
    X(2) = (D*uy - sgn*ux*(test)^0.5)/ur^2;
    Y(1) = (-D*ux + abs(uy)*(test)^0.5)/ur^2;
    Y(2) = (-D*ux - abs(uy)*(test)^0.5)/ur^2;
    % find value of ``t"
    if ux ~= 0
        t1 = (X(1)-x)/ux;
        t2 = (X(2)-x)/ux;
        % find Z
        Z(1) = z + uz*t1;
        Z(2) = z + uz*t2;
    elseif uy ~= 0
```

```
        t1 = (Y(1)-y)/uy;
        t2 = (Y(2)-y)/uy;
        % find Z
        Z(1) = z + uz*t1;
        Z(2) = z + uz*t2;
    end % in any other case the line is vertical and Z=z
end
```

A.19 MARSTanPoint

5

```
function[X, Y, Z, Ux, Uy, b, theta] = MARStanpoint(ux,uy,uz,r,h)

% #########################################################################
%
% This function finds a suitable tangent point for the cylander which
% buffers the singularity z-axis
%
% Inputs:     r   the radius of the cylander
%             [ux,uy,uz] the direction vector of the line
%             h   the minimum distance from the line to the origin
%
% Outputs:    [X,Y,Z] the tangent point
%
% #########################################################################

% the angle between the stride-line and the z-axis is found
theta = atan2((ux^2 + uy^2)^0.5, uz);

if theta == 0
    Z = -h;
else
% find the length of the third side of the right triange with other
% sides r and h
    b = (h^2 - r^2)^0.5;

% find Z
    Z = b/sin(theta);
end

% make ux, uy a unit vector in the x-y plane
if (ux == 0) & (uy == 0)
    Ux = ux;
    Uy = uy;
else
    Ux = ux/(ux^2+uy^2)^0.5;
    Uy = uy/(ux^2+uy^2)^0.5;
end
% find X and Y
XY = r.*[(Uy),(-Ux)];

X=XY(1); Y=XY(2);
```

A.20 MARSTanPointAlt

5

```
function[X, Y, Z] = MARStanpointALT(Ux, Uy, theta, b, r)

% ####################################################################
%
% This function finds the orthagonal intersection point between
% the stride-line and the height-line
%
% Inputs: (Ux, Uy) the 2d x-y-plane unite vector for the stide-line
%         b the length of the line from the tip of r to q
%         theta the angel between the r-tip, p, and q
%
% Outputs: (X, Y, Z) the point coordinates
%
% ####################################################################

% find the R vector
Uxr = Uy/(Ux^2+Uy^2)^0.5;
Uyr = -Ux/(Ux^2+Uy^2)^0.5;
R = r*[Uxr, Uyr, 0];

% find the B vector
B = [b*cos(theta)*[ -Uyr/(Uyr^2+Uxr^2)^0.5, Uxr/(Uyr^2+Uxr^2)^0.5],-b*sin(theta)];

% add the R and B vectors to find the point

Q = R+B;
X=Q(1); Y=Q(2); Z=Q(3);
```

A.21 MARSShell1Int

5

```
function[X,Y,Z] = MARSshell1int(x1, y1, z1, Ux, Uy, Uz, buffer)

% % DEFINING LIMB DIMENTIONS
% L1=5;   % upper limb length
% L2=6;   % lower limb length

% DEFINING THE LINE
% x1=0; y1=0; z1=-0.2; % point on line
% Ux=1; Uy=0; Uz=0; % Direction of line
ux=Ux/(Ux^2+Uy^2+Uz^2); %Change to unit vector
uy=Uy/(Ux^2+Uy^2+Uz^2); %Change to unit vector
uz=Uz/(Ux^2+Uy^2+Uz^2); %Change to unit vector

% DEFINING THE TORUS SECTION
r=5; % radius from senter of torus to senter of tube
q=6-buffer; % radius from center of tube to torus shell
thetamin=-10*pi/180;    %minimum azimuth
thetamax=190*pi/180;    %maximum azimuth
% phimin=-90*pi/180;      %minimum polar
```

```
% phimax=0*pi/180;        %maximum polar

% FINDING AND VALIDATING POINTS OF INTERSECTION BETWEEN THE LINE AND THE
% SHELL
R=MARSroots(r,q,x1,y1,z1,ux,uy,uz); %substitutes parametric equation of
% line into equation of torus and finds roots for paramater ‘‘t"
a = angle(R);                  %exclude imaginary roots
root=[];                          %exclude imaginary roots
for i=1:length(R)              %exclude imaginary roots
    if a(i) == 0 | a(i) == pi  %exclude imaginary roots
        root=[root,R(i)];            %exclude imaginary roots
    end                        %exclude imaginary roots
end                            %exclude imaginary roots
R=root;                           %exclude imaginary roots
x = x1 + R.*ux; % find coordinates from roots
y = y1 + R.*uy; % find coordinates from roots
z = z1 + R.*uz; % find coordinates from roots
X=[]; Y=[]; Z=[];                  % initialize data sets
for i=1:length(R)                  % remove points that dont lie on shell
    if z(i) < 0
        if x(i)^2+y(i)^2 > r^2
            theta = atan2(y(i),x(i));
            if theta < -pi/2
                theta = theta + 2*pi;
            end
            if (thetamin < theta) & (theta < thetamax)
                X=[X,x(i)];            % update points
                Y=[Y,y(i)];            % update points
                Z=[Z,z(i)];            % update points
            end
        end
    end
end
```

A.22 MARSShell2Int

5

```
function[X,Y,Z] = MARSshell2int(x1, y1, z1, Ux, Uy, Uz, buffer)

% DEFINING LIMB DIMENTIONS
% L1=5;   % upper limb length
% L2=6;   % lower limb length

% DEFINING THE LINE
% x1=0; y1=0; z1=-0.2; % point on line
% Ux=1; Uy=0; Uz=0; % Direction of line
ux=Ux/(Ux^2+Uy^2+Uz^2); %Change to unit vector
uy=Uy/(Ux^2+Uy^2+Uz^2); %Change to unit vector
uz=Uz/(Ux^2+Uy^2+Uz^2); %Change to unit vector

% DEFINING THE TORUS SECTION
r=-5*sin(20*pi/180); % radius from senter of torus to center of tube
q=6-buffer; % radius from center of tube to torus shell
ztorus=-5*cos(20*pi/180); % location of center of torus on the z-axis
```

```
thetamin=-170*pi/180;    %minimum azimuth
thetamax=-10*pi/180;    %maximum azimuth
% phimin=-90*pi/180;      %minimum polar
% phimax=0*pi/180;        %maximum polar

% ADJUSTING THE POINT ON THE LINE TO COMPENSATE FOR THE TORUS CENTER
% LOCATION
z1=z1-ztorus;

% FINDING AND VALIDATING POINTS OF INTERSECTION BETWEEN THE LINE AND THE
% SHELL
R=MARSroots(r,q,x1,y1,z1,ux,uy,uz); %substitutes parametric equation of
% line into equation of torus and finds roots for paramater ``t"
a = angle(R);                   %exclude imaginary roots
root=[];                        %exclude imaginary roots
for i=1:length(R)               %exclude imaginary roots
    if a(i) == 0 | a(i) == pi   %exclude imaginary roots
        root=[root,R(i)];       %exclude imaginary roots
    end                         %exclude imaginary roots
end                             %exclude imaginary roots
R=root;                           %exclude imaginary roots
x = x1 + R.*ux; % find coordinates from roots
y = y1 + R.*uy; % find coordinates from roots
z = z1 + R.*uz + ztorus; % find coordinates from roots
X=[]; Y=[]; Z=[];                   % initialize data sets
% remove non-floating point errors
x = double(x); y = double(y); z = double(z);
% set up variables
xcenter=5*sin(20*pi/180);
ycenter=-5*cos(20*pi/180);
centradius=q+buffer;
for i=1:length(R)                   % remove points that dont lie on shell
    if z(i) < -5*cos(20*pi/180) + 6*sin(20*pi/180)
        if z(i) > -11*cos(20*pi/180)
            xyradius=(x(i)^2+y(i)^2)^0.5;
            test1=(xyradius-xcenter)^2+(z(i)-ycenter)^2;
            test2=centradius^2+buffer;
            if test1 >= test2-0.1
                theta = atan2(y(i),x(i));
                if (thetamin < theta) & (theta < thetamax)

                       X=[X,x(i)];             % update points
                       Y=[Y,y(i)];             % update points
                       Z=[Z,z(i)];             % update points

                end
            end
        end
    end
end                                 % remove points that dont lie on shell
```

A.23 MARSShell3Int

5

```
function[X,Y,Z] = MARSshell3int(x1, y1, z1, ux, uy, uz, buffer)

% ###############################################################
%
% This function determines where a line intersects shell 3
% The function calls function MARScircLineInt.m to find the
% points where the line intersects the sphere
% The function then eliminates points which do not fall on the
% shell section of the sphere
%
% Inputs: (x1, y1, z1) a point on the line
%         (Ux, Uy, Uz) the unit vector of the line
%         buffer a margine to eliminate error
%
% Outputs:    X the set of x cordinates
%             Y the set of y cordinates
%             Z the set of z cordinates
%
% ###############################################################

% set up sphere parameters
r = (25 + 36)^0.5;
xs = 0;
ys = 0;
zs = 0;

% find intersection points
[P, test]=MARScircLineInt(x1, y1, z1, ux, uy, uz, xs, ys, zs, r, buffer);

% Initialize outputs
X=[]; Y=[]; Z=[];

% find intersection
if test == 0
    % iterate through the number of intersection points
    for i = 1:min(size(P))
        x = P(1,i);
        y = P(2,i);
        z = P(3,i);
        angle = atan2(y,x);
        if angle < -pi/2
            angle = angle + 2*pi;
        end
        if  angle > -10*pi/180 & angle < 190*pi/180
            if z < -6
                X=[X,x];
                Y=[Y,y];
                Z=[Z,z];
            end
        else
            if z < -5*cos(20*pi/180) + 6*sin(20*pi/180)
                X=[X,x];
                Y=[Y,y];
                Z=[Z,z];
```

```
            end
        end
    end
end
```

A.24 MARSShell4Int

5

```
function[X,Y,Z] = MARSshell4int(x1, y1, z1, ux, uy, uz, buffer)

% ################################################################
%
% This function determines where a line intersects shell 4
% The function calls function MARScircLineInt.m to find the
% points where the line intersects the sphere
% The function then eliminates points which do not fall on the
% shell section of the sphere
%
% Inputs: (x1, y1, z1) a point on the line
%         (Ux, Uy, Uz) the unit vector of the line
%         buffer a margine to eliminate error
%
% Outputs:    X the set of x cordinates
%             Y the set of y cordinates
%             Z the set of z cordinates
%
% ################################################################

% set up sphere parameters
r = 11;
xs = 0;
ys = 0;
zs = 0;

% find intersection points
[P, test]=MARScircLineInt(x1, y1, z1, ux, uy, uz, xs, ys, zs, r, buffer);

% Initialize outputs
X=[]; Y=[]; Z=[];

% find intersection
if test == 0
    % iterate through the number of intersection points
    for i = 1:min(size(P))
        x = P(1,i);
        y = P(2,i);
        z = P(3,i);
        angle = atan2(y,x);
        if angle < -pi/2
            angle = angle + 2*pi;
        end
        if  angle > -10*pi/180 & angle < 190*pi/180
            if z < 0
                X=[X,x];
                Y=[Y,y];
```

```
                Z=[Z,z];
            end
        else
            if z < -11*cos(20*pi/180)
                X=[X,x];
                Y=[Y,y];
                Z=[Z,z];
            end
        end
    end
end
```

A.25 MARSPlane1Int

5

```
function[X, Y, Z] = MARSplane1Int(x, y, z, ux, uy, uz)

% ####################################################################
%
% This function uses MARSplaneInt to find  the piont of intersecton
% between and line a plane.  The function then tests the point to see
% if it lies on the area where the plane makes up part of the 3D
% workspace shell of the MARS limb
%
% Inputs: (x, y, z) = a point on the line
%         (ux, uy, uz) = the unit vector of the line
%
% Outputs: (X, Y, Z) = a valid point of intersection.  Empty if
%                      there is no valid intersection point
%
% ####################################################################

% Define the angle the plane makes with the x-y-plane
theta = 190*pi/180;

% find the intersection point
[X, Y, Z] = MARSplaneInt(x, y, z, ux, uy, uz, theta);
% Eliminate points in the wrong quadrant
if X<=0
    X=[];Y=[];Z=[];
end
% establish intersection test criteria
xx = (X^2+(X*sin(10*pi/180))^2)^0.5;
yy = Z;
test1 = (xx - 5)^2 + yy^2;
test2 = (xx - 5*sin(20*pi/180))^2 + (yy + 5*cos(20*pi/180))^2;
test3 = xx^2 + yy^2;
% eliminate points on the z-axis
if X==0 | Y==0
    X=[];Y=[];Z=[];
end
% test for no intersection
if X ~([]);
    if max(~((yy<0) & (yy>-11*cos(20*pi/180)) & (test1>36) & (test2>36) & (test3<=121)))>0
        if max(~((test2<=36) & (test1<=36) & (test3>61)))>0
```

```
            X=[];Y=[];Z=[];
        end
    end
end

% test = 1;
% if X == []
%     test = 0;
% else
%     % Test intersection bounds
%     % set up semipolar cordinates
%     xx = (X^2+Y^2)^0.5;
%     yy = Z;
%     if (xx - 5)^2 + yy^2 <= 36
%         if (xx - 5*cos(20*pi/180))^2 + (yy + 6*cos(20*pi/180))^2 <= 36
%             if xx^2 + yy^2 > 25+36
%                 test = 1;
%             end
%         end
%     end
%     if test ~= 1
%         if xx^2 + yy^2 <= 25+36
%             if (xx - 5)^2 + yy^2 > 36
%                 if (xx - 5*cos(20*pi/180))^2 + (yy + 6*cos(20*pi/180))^2 > 36
%                     test = 1;
%                 end
%             end
%         end
%     end
%     if test ~= 1
%         X=[];Y=[];Z=[];
%     end
% end
```

A.26 MARSPlane2Int

5

```
function[X, Y, Z] = MARSplane2Int(x, y, z, ux, uy, uz)

% #####################################################################
%
% This function uses MARSplaneInt to find  the piont of intersecton
% between and line a plane.  The function then tests the point to see
% if it lies on the area where the plane makes up part of the 3D
% workspace shell of the MARS limb
%
% Inputs: (x, y, z) = a point on the line
%         (ux, uy, uz) = the unit vector of the line
%
% Outputs: (X, Y, Z) = a valid point of intersection.  Empty if
%                     there is no valid intersection point
%
% #####################################################################
```

```
% Define the angle the plane makes with the x-y-plane
theta = -10*pi/180;

% find the intersection point
[X, Y, Z] = MARSplaneInt(x, y, z, ux, uy, uz, theta);
% Eliminate points in the wrong quadrant
if X>=0
    X=[];Y=[];Z=[];
end
% establish intersection test criteria
xx = (X^2+(X*sin(10*pi/180))^2)^0.5;
yy = Z;
test1 = (xx - 5)^2 + yy^2;
test2 = (xx - 5*sin(20*pi/180))^2 + (yy + 5*cos(20*pi/180))^2;
test3 = xx^2 + yy^2;
% eliminate points on the z-axis
if X==0 | Y==0
    X=[];Y=[];Z=[];
end
% test for no intersection
if X ~([]);
    if max(~((yy<0) & (yy>-11*cos(20*pi/180)) & (test1>36) & (test2>36) & (test3<=121)))>0
        if max(~((test2<=36) & (test1<=36) & (test3>61)))>0
            X=[];Y=[];Z=[];
        end
    end
end

% test = 1;
% if X == []
%     test = 0;
% else
%     % Test intersection bounds
%     % set up semipolar cordinates
%     xx = (X^2+Y^2)^0.5;
%     yy = Z;
%     if (xx - 5)^2 + yy^2 <= 36
%         if (xx - 5*cos(20*pi/180))^2 + (yy + 6*cos(20*pi/180))^2 <= 36
%             if xx^2 + yy^2 > 25+36
%                 test = 1;
%             end
%         end
%     end
%     if test ~= 1
%         if xx^2 + yy^2 <= 25+36
%             if (xx - 5)^2 + yy^2 > 36
%                 if (xx - 5*cos(20*pi/180))^2 + (yy + 6*cos(20*pi/180))^2 > 36
%                     test = 1;
%                 end
%             end
%         end
%     end
%     if test ~= 1
%         X=[];Y=[];Z=[];
%     end
% end
```

A.27 MARSRoots

5

```
function[R] = MARSroots(r,q,x1,y1,z1,ux,uy,uz)

% ##############################################################
%
% This function helps to find the points at which a line
% intersects the shell of a torus.  The output roots are the
% values of ‘‘t’’ which when usbstituted into the parametric
% equation of a line give points in space.
%
% Inputs: r   radius from senter of torus to senter of tube
%         q   radius from center of tube to torus shell
%         (x1, y1, z1) corrdinates of given point on line
%         (ux, uy, uz) unit vector of line direction
%
% Outputs:    r1, r2, r3, r4  the roots or values of ‘‘t"
%
% ##############################################################

% Set inputs
r2=r^2; r4=r^4;
q2=q^2; q4=q^4; %used for alfa

% Find input powers to save computational time
x12=x1^2; x13=x1^3; x14=x1^4;
y12=y1^2; y13=y1^3; y14=y1^4;
z12=z1^2; z13=z1^3; z14=z1^4;

ux2=ux^2;
uy2=uy^2;
uz2=uz^2;

% Calculate coeficients

a = (ux2+uy2+uz2)^2 / (4*r2*q2);

b = (r2*q^2)^(-1) * ( (ux2+uy2+uz2)*(ux*x1+uy*y1+uz*z1) );

c = (2*r2*q2)^(-1) * (4*uy*uz*y1*z1+4*ux*x1*(uy*y1+uz*z1)+ux2*(-r2-q2+3*x12+y12+z12)+
        uy2*(-r2-q2+x12+3*y12+z12)+uz2*(r2-q2+x12+y12+3*z12));

d = (r2*q2)^(-1) * (ux*x1*(-r2-q2+x12+y12+z12)+uy*y1*(-r2-q2+x12+y12+z12)+uz*z1*(r2-q2+x12+y12+z12));

e = (4*r2*q2)^(-1) * (x14+y14-2*y12*(r2+q2-z12)-2*x12*(r2+q2-y12-z12)+(r2-q2+z12)^2);

% Find Roots

[R]=roots([a, b, c, d, e]);

% r1=R(1,1);r2=R(2,1);r3=R(3,1);r4=R(4,1);
```

A.28 MARSCircLinInt

5

```
function[P,test] = MARScircLineInt(xl, yl, zl, ux, uy, uz, xs, ys, zs, r, buffer)

% ##################################################################
%
% This function finds the points where a line intersects a sphere
% This function is specialized for lines which originate at a
% point within the sphere
%
% Inputs: (xl, yl, zl) point on line
%         (ux, uy, uz) unit vector of line
%         (xs, ys, zs) center of sphere
%         r radius of sphere
%
% Outputs:    P a 3*2 matrix on sphere intersection points
%                 empty in no intersection
%             test if test is 1 not 0 the point is outsie the
%                    sphere
%             buffer: small amount to avoid full extention
% ##################################################################

% Test to see in the point on the line is in inside the sphere
OC=[xs; ys; zs] - [xl; yl; zl];
% Initialize
test=0; P=[];
magOC = (OC(1)^2 + OC(2)^2 + OC(3)^2)^0.5;
if  magOC > r-buffer
    test=1;
end

    L = OC'*[ux; uy; uz];
    HC2 = (r-buffer)^2-magOC^2+L^2;
    if HC2 > 0
        t1=L+(HC2)^0.5;
        t2=L-(HC2)^0.5;
        P=[xl; yl; zl] + t1*[ux; uy; uz];
        P=[P, [xl; yl; zl] + t2*[ux; uy; uz] ];
    end
```

A.29 MARSPlaneInt

5

```
function[X, Y, Z] = MARSplaneInt(x, y, z, ux, uy, uz, theta)

% #######################################################
%
% This function finds the intersection points of a line
% with a plane, assuming the plane passes through the
% point (0,0,0) and contains the z-axis
%
```

```
% Inputs: theta = the anlge the plane forms in the x-y plane
%         (x, y, z) = a point on the line
%         (ux, uy, uz) = the unit vector of the line
%
% Outputs:    (X, Y, Z) = The point of intersection
%             if the line lies in the plane (X, Y, Z)
%             are empty
%
% #######################################################

% Find (a, b, c) a vector normal to the plane
a = sin(theta);
b = cos(theta);
c = 0;

% Find t
denominator = [a, b, c]*[ux, uy, uz]';

if denominator ~= 0
    t = -(x*a + y*b + z*c) / denominator;

    % Find intersection point

    X = x+ux*t;
    Y = y+uy*t;
    Z = z+uz*t;
else
    X=[];Y=[];Z=[];
end

% % Set up three points on the plane
% x1=0;y1=0;z1=0;
% x2=0;y2=0;z2=1;
% x3=cos(theta);
% y3=sin(theta);
% z3=0;
%
% % Set up the numerator
% numerator = [1,1,1,1;x1,x2,x3,x;y1,y2,y3,y;z1,z2,z3,z];
%
% % Set up the denominator
% denominator = [1,1,1,0;x1,x2,x3,ux;y1,y2,y3,uy;z1,z2,z3,uz];
%
% % Find t
% t = det(-(denominator*numerator'));
%
% % Solve for X, Y, Z
%     X = x+ux*t;
%     Y = y+uy*t;
%     Z = z+uz*t;
```

Appendix B

Mathematica script used to find the polynomial terms for stride-line torus intersection roots

Once the parametric equation for a torus was expressed in terms of x, y, z, r, and α, the Mathematica script presented below was used to substitute the parametric equation of a line into the expression.

$$x\left[\mathrm{t}_, x_1_, u_x_\right] = x_1 + t * u_x;$$
$$y\left[\mathrm{t}_, y_1_, u_x_\right] = y_1 + t * u_y;$$
$$z\left[\mathrm{t}_, z_1_, u_x_\right] = z_1 + t * u_z;$$

$$f\left[\mathrm{t}_, \mathrm{r}_, \alpha_, x_1_, y_1_, z_1_, u_x_, u_y_, u_z_\right] =$$
$$\left(\frac{(x_1+t*u_x)^2+(y_1+t*u_y)^2+(z_1+t*u_z)^2-r^2-\alpha^2}{2*r*\alpha}\right)^2 + \left(\frac{(z_1+t*u_z)}{\alpha}\right)^2 - 1;$$

ExpandAll[%];
Apart[%, t];
f[t_] = Collect[%, t];
$\{e, d, c, b, a\}$ = Simplify[CoefficientList[%, _]]

$$\begin{pmatrix} a \\ b \\ c \\ d \\ e \end{pmatrix} //\text{MatrixForm};$$

$g[\text{t_}] = t^4 * a + t^3 * b + t^2 * c + t * d + e;$
$\text{Roots}\left[t^4 * a + t^3 * b + t^2 * c + t * d + e == 0, t\right];$

Appendix C

Walking Algorithm Code Developed by Open Tech

Dan Larimer of Open Tech Inc. developed a set of code for generating walking gates for MARS sing the simplified or circular 2D general workspace. Dan also developed a graphical user interface (GUI) to run the code, a 3D visualizer which renders a model of the robot while walking, and code to communicate with the Dynamixel actuators.

While only the walking algorithm code is presented in this appendix, the GUI and visualizer are available upon request and the code for communicating with the actuators may be available for purchase upon request. Dan Larimer can be reached at: dlarimer@opentechinc.com

A note from Dan Larimer about the code included in this appendix as well as the section of his code pertaining to the walking algorithm follow:

```
/**
    The code below implements the walking algorithm for the MARS robot.  It
    depends upon the following 3rd-Party Libraries:
        - Qt 4.1.x           http://www.troltech.com
        - Open Scene Graph   http://www.openscenegraph.org/
        - Dynamixel Library  see: dlarimer@opentechinc.com

    The code is implemented as two classes, Robot and Arm.  The majority
    of the kinematics code is found in the Arm.cpp implementation.  The
    Robot class serves to aggregate the arms on the body and synchronize
    their movement according to a desired linear and angular velocity for
    the center of the MARS robot's body.

    The code as given does not compile because it is just an excerpt of
    the critical components.  Those skilled in software development should
    have no trouble using the code provided below to replicate the walking
    algorithm.
```

```
*/
//==========================================================================
/// @file Robot.h
//==========================================================================
#ifndef _ROBOT_H_
#define _ROBOT_H_
#include <QObject>
#include ‘‘Arm.h"
#include <vector>
#include <osg/Vec3>
#include ‘‘HeightMap.h"
#include <dynamixel/SerialBus.h>
#include <dynamixel/SerialPort.h>
#include <dynamixel/DX116.h>
#include <dynamixel/Console.h>
#include <dynamixel/macros.h>
#include <dynamixel/DX116.h>


class Robot : public  QObject
{
    Q_OBJECT
    public:
        Robot();
        std::vector<Arm*>& getArms() { return m_arms; }


        void setBus( SerialBus* b );

        int step( int delta_time_ms = 10, osg::Vec3* delta_pos = NULL);

        void setBodyPosition( const osg::Vec3& xyz );
        void setBodyNormal( const osg::Vec3& norm );
        void setBodyRotation( float rot );

        void setBodyVelocity( const osg::Vec3& vel );
        void setBodyNormalVelocity( const osg::Vec3& vec );
        void setBodyRotateVelocity( float rotate_vel );

        bool setHandPosition( int hand, const osg::Vec3& world_xyz );

        osg::Vec3 getBodyPosition()const         { return m_body_pos;         }
        osg::Vec3 getBodyNormal()const           { return m_body_normal;      }
        float     getBodyRotation()const         { return m_body_rot;         }

        osg::Vec3 getBodyVelocity()const         { return m_body_vel;         }
        osg::Vec3 getBodyNormalVelocity()const   { return m_body_normal_vel;  }
        float     getBodyRotationVelocity()const { return m_body_rot_vel;     }

        osg::Vec3 armToRobot( Arm* a, const osg::Vec3& point,
                                bool direction_only = false )const;
        osg::Vec3 robotToArm( Arm* a, const osg::Vec3& point,
                                bool direction_only = false )const;

        osg::Vec3 robotToWorld( const osg::Vec3& point,
                                bool direction_only = false )const;
        osg::Vec3 worldToRobot( const osg::Vec3& point,
```

```
                                bool direction_only = false )const;

        osg::Vec3 armToWorld( Arm* a, const osg::Vec3& point,
                                bool is_vec = false )const;
        osg::Vec3 worldToArm( Arm* a, const osg::Vec3& point,
                                bool is_vec = false )const;

        osg::Vec3 getGroundPoint( float x, float y )const;
        float     getHeightAboveGround( const osg::Vec3& pt )const;


    public slots:
        void readArms();
        void printArms();
        void configureArms();
        void disableArms();
        void enableArms();
        void updateActuators();
        void calibrateZero();

    private:
        float simToActuator( int id, float angle );
        SerialBus*          m_bus;

        bool                m_configured;

        osg::Vec3 calculateNextLiftArmPosition( Arm* arm, const osg::Vec3& new_world_tip,
        const osg::Vec3& tip_vel, float delta_time_s );
        float     calculateStridePercent( Arm* a, const osg::Vec3& world_tip_pos );

        std::vector<Arm*>   m_arms;

        osg::Vec3           m_body_pos;
        osg::Vec3           m_body_normal;
        float               m_body_rot;

        osg::Vec3           m_body_vel;
        osg::Vec3           m_body_normal_vel;
        float               m_body_rot_vel;

        float               m_step_height;

        HeightMap*          m_ground;

        bool                m_set;

        QTime               m_time;
        float               m_cal[65][3];
        float               m_last_goals[65];
};


#endif

//============================================================================
/// @file Robot.cpp
```

```
//==========================================================================
#include ‘‘Robot.h"
#include <osg/Matrix>
#include <osg/Vec3>

#define SQ(X) (X*X)

Robot::Robot()
{
    m_configured = false;
    m_arms.push_back( new Arm() );
    m_arms.push_back( new Arm() );
    m_arms.push_back( new Arm() );
    m_arms.push_back( new Arm() );
    m_arms.push_back( new Arm() );
    m_arms.push_back( new Arm() );

    m_bus = NULL;

    m_set = true;

    m_ground = new HeightMap( 1024 );

    m_body_pos    = osg::Vec3( 1024/200, 1024/200, .22);
    m_body_normal = osg::Vec3( 0, 0, 1 );
    m_body_normal.normalize();
    m_body_rot    = 0; //M_PI/4;

    m_body_vel        = osg::Vec3(0,0,0);
    m_body_normal_vel = osg::Vec3( 0, 0, 1 );
    m_body_rot_vel    = 0;

    m_step_height = .05;


    osg::Matrix rotate = osg::Matrix::rotate( M_PI/3, 0, 0, 1 );
    osg::Vec3   shoulder_pos(0,.1778,0); // 7 inches

    float angle = 0;


    printf( ‘‘0) shoulder_pos: %f, %f, %f\n", shoulder_pos[0], shoulder_pos[1], shoulder_pos[2] );
    m_arms[0]->setShoulderPosition( shoulder_pos );
    m_arms[0]->setShoulderOrientation( osg::Vec3( -angle, 0, 0 ) );
    m_arms[0]->setState( Arm::SUPPORT );
    assert( setHandPosition( 0, getGroundPoint( m_body_pos.x() + shoulder_pos.x()*1.5, m_body_pos.y() +
    shoulder_pos.y()*1.5 ) ) );

    shoulder_pos = rotate * shoulder_pos;
    angle += M_PI/3;
    printf( ‘‘1) shoulder_pos: %f, %f, %f\n", shoulder_pos[0], shoulder_pos[1], shoulder_pos[2] );
    m_arms[1]->setShoulderPosition( shoulder_pos );
    m_arms[1]->setShoulderOrientation( osg::Vec3( -angle, 0, 0 ) );
    m_arms[1]->setState( Arm::LIFT );
    assert( setHandPosition( 1, getGroundPoint( m_body_pos.x() + shoulder_pos.x()*1.5, m_body_pos.y() +
    shoulder_pos.y()*1.5 ) ) );
```

```
    shoulder_pos = rotate * shoulder_pos;
    angle += M_PI/3;
    printf( ‘‘2) shoulder_pos: %f, %f, %f\n", shoulder_pos[0], shoulder_pos[1], shoulder_pos[2] );
    m_arms[2]->setShoulderPosition( shoulder_pos );
    m_arms[2]->setShoulderOrientation( osg::Vec3( -angle, 0, 0 ) );
    m_arms[2]->setState( Arm::SUPPORT );
    assert( setHandPosition( 2, getGroundPoint( m_body_pos.x() + shoulder_pos.x()*1.5, m_body_pos.y() +
    shoulder_pos.y()*1.5 ) ) );

    shoulder_pos = rotate * shoulder_pos;
    angle += M_PI/3;
    printf( ‘‘3) shoulder_pos: %f, %f, %f\n", shoulder_pos[0], shoulder_pos[1], shoulder_pos[2] );
    m_arms[3]->setShoulderPosition( shoulder_pos );
    m_arms[3]->setShoulderOrientation( osg::Vec3( -angle, 0, 0 ) );
    m_arms[3]->setState( Arm::LIFT );
    assert( setHandPosition( 3, getGroundPoint( m_body_pos.x() + shoulder_pos.x()*1.5, m_body_pos.y() +
    shoulder_pos.y()*1.5 ) ) );

    shoulder_pos = rotate * shoulder_pos;
    angle += M_PI/3;
    printf( ‘‘4) shoulder_pos: %f, %f, %f\n", shoulder_pos[0], shoulder_pos[1], shoulder_pos[2] );
    m_arms[4]->setShoulderPosition( shoulder_pos );
    m_arms[4]->setShoulderOrientation( osg::Vec3( -angle, 0, 0 ) );
    m_arms[4]->setState( Arm::SUPPORT );
    assert( setHandPosition( 4, getGroundPoint( m_body_pos.x() + shoulder_pos.x()*1.5, m_body_pos.y() +
    shoulder_pos.y()*1.5 ) ) );

    shoulder_pos = rotate * shoulder_pos;
    angle += M_PI/3;
    printf( ‘‘5) shoulder_pos: %f, %f, %f\n", shoulder_pos[0], shoulder_pos[1], shoulder_pos[2] );
    m_arms[5]->setShoulderPosition( shoulder_pos );
    m_arms[5]->setShoulderOrientation( osg::Vec3( -angle, 0, 0 ) );
    m_arms[5]->setState( Arm::LIFT );
    assert( setHandPosition( 5, getGroundPoint( m_body_pos.x() + shoulder_pos.x()*1.5, m_body_pos.y() +
    shoulder_pos.y()*1.5 ) ) );



    QTimer* act_update = new QTimer(this);
    assert_connect( act_update, SIGNAL(timeout()), this, SLOT(updateActuators()) );
    act_update->start(1000/20);
}

void Robot::setBus( SerialBus* b )
{
    m_bus = b;
}

osg::Vec3 Robot::getGroundPoint( float x, float y )const
{
//    printf( ‘‘getGroundPoint( %f, %f )\n", x, y );
    return m_ground->getPoint( x * 100, y * 100 ) * 0.01;
}

void Robot::setBodyPosition( const osg::Vec3& xyz )
{
    m_body_pos = xyz;
```

```
}

void Robot::setBodyNormal( const osg::Vec3& norm )
{
    m_body_normal = norm;
}

void Robot::setBodyRotation( float rot_rad )
{
    m_body_rot = rot_rad;
}

bool Robot::setHandPosition( int arm, const osg::Vec3& world_coord )
{
    assert( arm < 6 && arm >= 0 );
    printf( ‘‘setHandPosition( %f, %f, %f ) world coord \n", world_coord.x(), world_coord.y(), world_coord.z() );
    osg::Vec3 arm_coord = worldToArm( m_arms[arm], world_coord );
    printf( ‘‘setHandPosition( %f, %f, %f ) arm coord \n", arm_coord.x(), arm_coord.y(), arm_coord.z() );
    return m_arms[arm]->setTipPosition( arm_coord ) == Arm::NONE;
}


void Robot::setBodyVelocity( const osg::Vec3& vel )
{
    m_body_vel = vel;
}
void Robot::setBodyRotateVelocity( float rot_z )
{
    m_body_rot_vel = rot_z;
}
void Robot::setBodyNormalVelocity( const osg::Vec3& vec )
{
    m_body_normal_vel = vec;
}


int Robot::step( int delta_time_ms, osg::Vec3*  )
{
    float delta_time_s = delta_time_ms / 1000.0;

    // save the current state incase of error
    osg::Vec3 current_body_pos    = m_body_pos;
    float     current_body_rot    = m_body_rot;
    osg::Vec3 current_body_normal = m_body_normal;

    // calculate the new state
    osg::Vec3 next_body_pos = m_body_pos + m_body_vel * delta_time_s;
    float     next_body_rot = m_body_rot + m_body_rot_vel * delta_time_s;
    while( next_body_rot > 2*M_PI )  next_body_rot -= 2*M_PI;
    while( next_body_rot < -2*M_PI ) next_body_rot += 2*M_PI;
    osg::Vec3 next_body_normal =
                        m_body_normal + m_body_normal_vel * delta_time_s;
    next_body_normal.normalize();

    // get the current tip positions
    std::vector<osg::Vec3> current_world_tips;
    for( unsigned int i = 0; i < m_arms.size(); i++ )
```

```
{
    osg::Vec3 tip =  armToWorld(m_arms[i], m_arms[i]->getTipPosition());
    current_world_tips.push_back( tip );
}

// update the body position so that coordinate conversions work
// with the assumption of the new position
m_body_pos = next_body_pos;
m_body_rot = next_body_rot;
m_body_normal = next_body_normal;

// get the new tip positions
std::vector<osg::Vec3> new_world_tips;
for( unsigned int i = 0; i < m_arms.size(); i++ )
{
    osg::Vec3 tip = armToWorld(m_arms[i], m_arms[i]->getTipPosition());
    new_world_tips.push_back( tip );
}

// calculate the velocity of the tips in world coordinates
std::vector<osg::Vec3> world_tip_velocity;
for( unsigned int i = 0; i < new_world_tips.size(); i++ )
{
    world_tip_velocity.push_back(
            new_world_tips[i] - current_world_tips[i] );
}

// transform the world velocities into arm velocities
std::vector<osg::Vec3> arm_tip_velocity;
for( unsigned int i = 0; i < new_world_tips.size(); i++ )
{
    arm_tip_velocity.push_back(
        worldToArm( m_arms[i], world_tip_velocity[i], true ) );
}


// a place to store the new tip positions and angular velocities
std::vector<osg::Vec3> new_tip_pos;
new_tip_pos.resize(6);
std::vector<osg::Vec4> arm_ang_vel;
arm_ang_vel.resize(6);

Arm::ERROR err = Arm::NONE;

bool    swap_legs = false;
float   min_height = .00;
float   max_stride_percent = 0;
// keep the arm tips in the same position in world coordinates
for( unsigned int i = 0; i < m_arms.size(); i++ )
{
    int state = m_arms[i]->getState();
    if(  state == Arm::SUPPORT )
    {
        // the support arms must stay at the same position in
        // world coodinates, so therefore we must move them
        // in the opposite direction than the body movement
        // generated
```

```
        err =
            m_arms[i]->testArmMove( arm_tip_velocity[i] * -1, delta_time_s,
                                    &arm_ang_vel[i] );


        // test failed
        if( err != Arm::NONE )
        {
            if( err == Arm::OUT_OF_WORKSPACE )
            {
                swap_legs = true;
            }
           // break;
        }
        new_tip_pos[i] =
            m_arms[i]->getTipPosition() - arm_tip_velocity[i];
        float stride_percent = calculateStridePercent( m_arms[i], new_world_tips[i] );
        if( stride_percent > max_stride_percent )
            max_stride_percent = stride_percent;

 //       printf( ``stride percent: %f\n", stride_percent );

        /*
        if( .01 * (1-stride_percent) < min_height )
        {
            min_height = 3*.03 * (1-stride_percent);
            m_step_height = min_height;
        }
        */

    }
}
//print_debug( ``max_stride_percent: %f", max_stride_percent);

if( max_stride_percent >= .95 )
{
    max_stride_percent = 1;
    swap_legs = true;
}
if( max_stride_percent >= .90 )
{
    max_stride_percent = 1;
}

for( unsigned int i = 0; i < m_arms.size(); i++ )
{
    int state = m_arms[i]->getState();
    if(  state != Arm::SUPPORT )
    {
        //m_step_height = 0.1 * sqrtf(sqrtf(1-max_stride_percent));
        m_step_height = 0.1 * (1-max_stride_percent*max_stride_percent*max_stride_percent);
        new_tip_pos[i] = calculateNextLiftArmPosition( m_arms[i], new_world_tips[i],
        world_tip_velocity[i], delta_time_s );
    }
}

if( swap_legs )
```

```
    {
        m_step_height = 0;
        for( unsigned int i = 0; i < m_arms.size(); i++ )
        {
            int state = m_arms[i]->getState();
            if(  state != Arm::SUPPORT )
                new_tip_pos[i] = calculateNextLiftArmPosition( m_arms[i], new_world_tips[i],
                world_tip_velocity[i], delta_time_s );
        }

        printf( ``swap\n'' );
        for( unsigned int i = 0; i < m_arms.size(); i++ )
        {
            if( m_arms[i]->getState() == Arm::SUPPORT )
                m_arms[i]->setState( Arm::LIFT );
            else
                m_arms[i]->setState( Arm::SUPPORT );
        }
    }

    // don't move
    if( err != Arm::NONE )
    {
        m_body_pos = current_body_pos;
        m_body_rot = current_body_rot;
        m_body_normal = current_body_normal;
        return err;
    }

    // no errors, update the positions
    for( unsigned int i = 0; i < m_arms.size(); i++ )
    {
        m_arms[i]->setTipPosition( new_tip_pos[i] );
    }


    return Arm::NONE;
}

float Robot::getHeightAboveGround( const osg::Vec3& pt )const
{
    return pt.z() - getGroundPoint( pt.x(), pt.y() ).z();
}

float Robot::calculateStridePercent( Arm* arm, const osg::Vec3& world_tip_pos )
{
        osg::Vec3 shoulder_world = armToWorld( arm, osg::Vec3(0,0,0) );
        float h = shoulder_world.z() - getHeightAboveGround( shoulder_world );
        float a = arm->getArmLength();
        assert( SQ(a) - SQ(h) > 0 );
        float d = sqrt( SQ(a) - SQ(h) );

        osg::Vec3 robot_ground_pos = robotToWorld( osg::Vec3(0,0,0 ) );
        robot_ground_pos[2] = shoulder_world[2];

        osg::Vec3 dir_to_center = shoulder_world - robot_ground_pos;//
        armToWorld( arm, osg::Vec3( 0, -1, 0 ), true );
```

```
        osg::Vec3 dxy_to_center = dir_to_center;
        /*
        ( dir_to_center * osg::Vec3( 1, 0, 0 ),
                                 dir_to_center * osg::Vec3( 0, 1, 0 ), 0 );
                                 */
        dxy_to_center.normalize();
        dxy_to_center *= d/2;

        d *= .95; // safty margin

        osg::Vec3 shoulder_ground = shoulder_world;
        shoulder_ground[2] -= h;

        osg::Vec3 center = shoulder_ground + dxy_to_center;

        osg::Vec3 distance_vec = world_tip_pos - center;
        distance_vec[2] = 0;

        return distance_vec.length() / (d/2.0);

}


osg::Vec3 Robot::calculateNextLiftArmPosition( Arm* arm, const osg::Vec3& world_tip_pos,
const osg::Vec3& world_tip_vel, float delta_time_s )
{
        osg::Vec3 shoulder_world = armToWorld( arm, osg::Vec3(0,0,0) );
        float h = getHeightAboveGround( shoulder_world );
        float a = arm->getArmLength();
        float d = 0;
        if( SQ(a) - SQ(h) > 0 )
            d = sqrt( SQ(a) - SQ(h) );

        osg::Vec3 robot_ground_pos = robotToWorld( osg::Vec3(0,0,0 ) );
        robot_ground_pos[2] = shoulder_world[2];

        osg::Vec3 dir_to_center = shoulder_world - robot_ground_pos;//
        armToWorld( arm, osg::Vec3( 0, -1, 0 ), true );
        osg::Vec3 dxy_to_center = dir_to_center;
        /*
        ( dir_to_center * osg::Vec3( 1, 0, 0 ),
                                 dir_to_center * osg::Vec3( 0, 1, 0 ), 0 );
                                 */
        dxy_to_center.normalize();
        dxy_to_center *= d/2;

        d *= .75; // safty margin


        osg::Vec3 shoulder_ground = shoulder_world;
        shoulder_ground[2] -= h;

        osg::Vec3 w_tip_vel = world_tip_vel;
        w_tip_vel[2] = 0;
        w_tip_vel.normalize();
        w_tip_vel *= d/2;
```

```
        osg::Vec3 goal = shoulder_ground + dxy_to_center + w_tip_vel;
        goal = osg::Vec3( goal.x(), goal.y(), getHeightAboveGround( goal ) + m_step_height );

        osg::Vec3 tip_to_goal = goal - world_tip_pos; //armToWorld( arm, arm->getTipPosition() );
        osg::Vec3 tip_to_goal_delta = tip_to_goal;
        tip_to_goal[2] *= 2; // weigh things in favor of height
        tip_to_goal.normalize();

        tip_to_goal[0] *= world_tip_vel.length() * 1.25; // go 50% faster than the ground legs
        tip_to_goal[1] *= world_tip_vel.length() * 1.25; // go 50% faster than the ground legs
        tip_to_goal[2] *= world_tip_vel.length() * 5; // go 50% faster than the ground legs

        if( fabs(tip_to_goal[0]) > fabs(tip_to_goal_delta[0]) )
            tip_to_goal[0] = tip_to_goal_delta[0];
        if( fabs(tip_to_goal[1]) > fabs(tip_to_goal_delta[1]) )
            tip_to_goal[1] = tip_to_goal_delta[1];
        if( fabs(tip_to_goal[2]) > fabs(tip_to_goal_delta[2]) )
            tip_to_goal[2] = tip_to_goal_delta[2];

        osg::Vec3 arm_tip_to_goal = worldToArm( arm, tip_to_goal, true );

        osg::Vec4 ang_vel;
        Arm::ERROR err = arm->testArmMove( arm_tip_to_goal, delta_time_s, &ang_vel );
        if( err != Arm::NONE )
        {
            if( err == Arm::OUT_OF_WORKSPACE )
            {
                printf( ‘‘Error moving lift arm’’ );
                return arm->getTipPosition();
            }
            if( err >= Arm::EXCEEDED_MAXIMUM_SH_ANGULAR_VELOCITY && err <=
            Arm::EXCEEDED_MAXIMUM_ANGULAR_VELOCITY )
            {
                #define MAX_ANG_VEL ((114./60.)*2.*M_PI)  // 114 RPM -> 11.7286 rad/sec
                printf( ‘‘Error moving lift arm, scaling speed to max’’ );
                ang_vel.normalize();
                ang_vel *= MAX_ANG_VEL;

                osg::Vec4 new_ang = arm->getArmAngles() + ang_vel;
                return arm->calculateTipPosition( new_ang );
            }
            printf( ‘‘Error moving lift arm, couldn’t move\n’’ );
            return arm->getTipPosition();
        }
        return worldToArm( arm, world_tip_pos ) + arm_tip_to_goal;
}


/**
    Converts ’point’ in arm coordinates to a point in robot coordinates.
*/
osg::Vec3 Robot::armToRobot( Arm* arm, const osg::Vec3& point, bool is_vec )const
{
    osg::Vec3 sp = arm->getShoulderPosition();
    osg::Vec3 so = arm->getShoulderOrientation();

    osg::Matrix rot_h = osg::Matrix::rotate( so[0], 0, 0, 1 );
```

```
    osg::Matrix rot_p = osg::Matrix::rotate( so[1], 1, 0, 0 );
    osg::Matrix rot_r = osg::Matrix::rotate( so[2], 0, 1, 0 );
    osg::Matrix rot = rot_r * rot_p * rot_h;
    osg::Matrix inv_rot = osg::Matrix::inverse(rot);

    osg::Vec3 rot_point = rot * point;

    if( !is_vec )
        return rot_point + sp;
    else
        return rot_point;
}
/**
    Converts 'point' in robot coordinates to a point in 'arm' coordinates.
*/
osg::Vec3 Robot::robotToArm( Arm* arm, const osg::Vec3& point, bool is_vec )const
{
    osg::Vec3 sp = arm->getShoulderPosition();
    osg::Vec3 so = arm->getShoulderOrientation();

    osg::Matrix rot_h = osg::Matrix::rotate( so[0], 0, 0, 1 );
    osg::Matrix rot_p = osg::Matrix::rotate( so[1], 1, 0, 0 );
    osg::Matrix rot_r = osg::Matrix::rotate( so[2], 0, 1, 0 );
    osg::Matrix rot = rot_r * rot_p * rot_h;
    osg::Matrix inv_rot = osg::Matrix::inverse(rot);

    osg::Vec3 rtn;
    if( !is_vec )
        rtn = point - sp;
    else
        rtn = point;

    osg::Vec3 rot_point = inv_rot * rtn;

    osg::Vec3 check = armToRobot( arm, rot_point, is_vec );

    osg::Vec3 error = check - point;
    if( (check - point).length2() > .001 )
    {
    printf( ``error robotToArm( %f, %f, %f ) -> %f, %f, %f -> check: %f, %f, %f  error: %f, %f, %f \n",
             point.x(), point.y(), point.z(),
             rot_point.x(), rot_point.y(), rot_point.z(),
             check.x(), check.y(), check.z(),
             error.x(), error.y(), error.z()
             );
    }

    return rot_point;
}

osg::Vec3 Robot::worldToRobot( const osg::Vec3& point, bool is_vec )const
{
    osg::Vec3 up(0,0,1);

    float ang = acos( up * m_body_normal );
    osg::Vec3 rot_axis = up ^ m_body_normal;
    rot_axis.normalize();
```

```
    osg::Matrix body_rot = osg::Matrix::rotate( m_body_rot, 0, 0, 1 );

    osg::Matrix norm_rot = osg::Matrix::rotate( ang, rot_axis.x(),
                                                     rot_axis.y(),
                                                     rot_axis.z() );
    //osg::Matrix rot = body_rot * norm_rot;
    osg::Matrix rot = norm_rot * body_rot;
    osg::Matrix inv_rot = osg::Matrix::inverse(rot);

    osg::Vec3 rtn;

   // subtract translate
    if( !is_vec )
        rtn = point - m_body_pos;
    else
        rtn = point;

   // inverse rotate
    osg::Vec3 robot_pt = inv_rot * rtn;


    /*
    printf( ‘‘worldToRobot( %f, %f, %f ) -> %f, %f, %f \n",
             point.x(), point.y(), point.z(),
             robot_pt.x(), robot_pt.y(), robot_pt.z() );
             */
    return robot_pt;
}

osg::Vec3 Robot::robotToWorld( const osg::Vec3& point, bool is_vec )const
{
    osg::Vec3 up(0,0,1);

    float ang = acos( up * m_body_normal );
    osg::Vec3 rot_axis = up ^ m_body_normal;
    rot_axis.normalize();

    osg::Matrix body_rot = osg::Matrix::rotate( m_body_rot, 0, 0, 1 );

    osg::Matrix norm_rot = osg::Matrix::rotate( ang, rot_axis.x(),
                                                     rot_axis.y(),
                                                     rot_axis.z() );
    osg::Matrix rot = norm_rot * body_rot;

    osg::Vec3 robot_pt = rot * point;

    if( !is_vec )
        return robot_pt + m_body_pos;
    else
        return robot_pt;
}

osg::Vec3 Robot::armToWorld( Arm* a, const osg::Vec3& point, bool is_vec )const
{
    return robotToWorld( armToRobot( a, point, is_vec ), is_vec );
}
```

```
osg::Vec3 Robot::worldToArm( Arm* a, const osg::Vec3& point, bool is_vec )const
{
    return robotToArm( a, worldToRobot( point, is_vec ), is_vec );
}

void Robot::readArms()
{
    m_bus->setReturnLevel( 0xFE, 1 ); // on read only
    for( int id = 1; id <= 18; id++ )
    {
            DX116* act = m_bus->getActuator(id);
            if( act == NULL )
            {
                print_error( ‘‘Warning, Actuator: %d was not found on the bus", id );
                act = new DX116( id, m_bus );
                m_bus->addActuator( act );
            }
            act->setReturnLevel( DX116::ON_READ_INSTRUCTIONS );
            act->requestStatusUpdate();
            usleep(1000);
    }
}
void Robot::printArms()
{
    for( int i = 1; i <= 18; i++ )
    {
            char id = i;
            DX116* act = m_bus->getActuator(id);
            if( act == NULL )
            {
                print_error( ‘‘Warning, Actuator: %d was not found on the bus", id );
                act = new DX116( id, m_bus );
                m_bus->addActuator( act );
            }
            print_debug( ‘‘id: %d    angle: %f", id, act->getCurrentPosition() );
    }
    /*
    for( int leg = 1; leg <= 6; leg++ )
    {
        for( int motor = 1; motor <= 4; motor++ )
        {
            char id = 10 * leg + motor;
            DX116* act = m_bus->getActuator(id);
            if( act == NULL )
            {
                print_error( ‘‘Warning, Actuator: %d was not found on the bus", id );
                act = new DX116( id, m_bus );
                m_bus->addActuator( act );
            }
            print_debug( ‘‘id: %d    angle: %f", id, act->getCurrentPosition() );
        }
    }
    */
}
```

```
void Robot::configureArms()
{
    if( !m_bus )
        return;

    memset( &m_cal, 0, 18*3*sizeof(float) );
    /*
    m_cal[1][0] = 168;
    m_cal[2][0] = 164;
    m_cal[3][0] = 191;
    m_cal[4][0] = 187;
    m_cal[5][0] = 194;
    m_cal[6][0] = 169;
    m_cal[7][0] = 191;
    m_cal[8][0] = 171;
    m_cal[9][0] = 168;
    m_cal[10][0] = 191;
    m_cal[11][0] = 172;
    m_cal[12][0] = 168;
    m_cal[13][0] = 167;
    m_cal[14][0] = 189;
    m_cal[15][0] = 166;
    m_cal[16][0] = 193;
    m_cal[17][0] = 187;
    m_cal[18][0] = 192;
   */
m_cal[1][0] = 156.598240;
m_cal[2][0] = 157.771261;
m_cal[3][0] = 167.155425;
m_cal[4][0] = 157.478006;
m_cal[5][0] = 165.689150;
m_cal[6][0] = 136.950147;
m_cal[7][0] = 138.123167;
m_cal[8][0] = 159.530792;
m_cal[9][0] = 144.868035;
m_cal[10][0] = 160.703812;
m_cal[11][0] = 154.252199;
m_cal[12][0] = 163.049853;
m_cal[13][0] = 157.184751;
m_cal[14][0] = 156.598240;
m_cal[15][0] = 153.958944;
m_cal[16][0] = 157.478006;
m_cal[17][0] = 157.771261;
m_cal[18][0] = 159.530792;
for( int i = 0; i <= 18; i++ )
{
    m_cal[i][0] += 28;
}
   /*
    m_cal[1][0] = 180;
    m_cal[2][0] = 180;
    m_cal[3][0] = 180;
    m_cal[4][0] = 180;
    m_cal[5][0] = 180;
    m_cal[6][0] = 180;
    m_cal[7][0] = 180;
    m_cal[8][0] = 180;
```

```
m_cal[9][0] = 180;
m_cal[10][0] = 180;
m_cal[11][0] = 180;
m_cal[12][0] = 180;
m_cal[13][0] = 180;
m_cal[14][0] = 180;
m_cal[15][0] = 180;
m_cal[16][0] = 180;
m_cal[17][0] = 180;
m_cal[18][0] = 180;
*/

/*
for( int l = 1; l <= 6; l++ )
{
    m_cal[10*l+1][1] = 65;
    m_cal[10*l+1][2] = 80;
}
*/
for( int l = 0; l < 6; l++ )
{
    m_cal[3*l+1][1] = 70;
    m_cal[3*l+1][2] = 270;
}
for( int l = 1; l <= 6; l++ )
{
    m_cal[3*l+2][1] = 55;  // min
    m_cal[3*l+2][2] = 280; // max
}
for( int l = 1; l <= 6; l++ )
{
    m_cal[3*l+3][1] = 90;
    m_cal[3*l+3][2] = 280;
}


// set the return status level for all motors

m_bus->setReturnLevel( 0xFE, 1 ); // on read only
m_bus->setReturnDelayTime( 0xFE, 25 );
m_bus->setPunch( 0XFE, 2 );
m_bus->setCompliance( 0XFE, 16, 0, 16, 0 );
m_bus->setTorque( 0XFE, .95 );
m_bus->setTorqueEnabled( 0XFE, true );

for( int id = 1; id <= 18; id++ )
{
        DX116* act = m_bus->getActuator(id);
        if( act == NULL )
        {
            print_error( ‘‘Warning, Actuator: %d was not found on the bus", id );
            act = new DX116( id, m_bus );
            m_bus->addActuator( act );
            exit(1);
        }
        act->setReturnLevel( DX116::ON_READ_INSTRUCTIONS );
        act->setComplianceMarginAndSlope( 32, 1, 32, 1 , 35);
```

```
            act->setTorqueEnabled(true);
            act->setCurrentTorqueLimit( .99 );
            act->setOperatingAngleLimits(10,290);
            act->setGoalSpeed( 40 );
            act->setGoalPosition( m_cal[id][0] ); //m_cal[3*leg+motor+1][0] );
            //simToActuator( 3*leg+motor, 0 ) );
            act->writeCommand();
    }

    /*
    for( int leg = 0; leg < 6; leg++ )
    {
        for( int motor = 0; motor < 3; motor++ )
        {
            char id = 3 * (leg) + motor + 1;

            DX116* act = m_bus->getActuator(id);
            if( act == NULL )
            {
                print_error( ``Warning, Actuator: %d was not found on the bus", id );
                act = new DX116( id, m_bus );
                m_bus->addActuator( act );
                exit(1);
            }
            print_debug( ``id: %d", id );
            //m_bus->setTorque( id, 1 );
            act->setReturnLevel( DX116::ON_READ_INSTRUCTIONS );
            m_bus->setReturnDelayTime( 0xFE, 25 );
            //m_bus->setCompliance( id, 30, 3, 30, 3 );
            act->setComplianceMarginAndSlope( 30, 3, 30, 3 , 35);
            //m_bus->setTorque( id, .75 );
            act->setTorqueEnabled(true);
            act->setCurrentTorqueLimit( .75 );
            //act->setPunch( 35 );

            act->setGoalSpeed( 10 );
            print_error( ``setGoalPos( m_cal[%d][0] = %f )", id, m_cal[3*leg+motor+1][0] );
            act->setGoalPosition( 180 ); //m_cal[3*leg+motor+1][0] ); //simToActuator( 3*leg+motor, 0 ) );
            act->setOperatingAngleLimits(m_cal[3*leg + motor][1], m_cal[3*leg+motor][2]);

            m_bus->setAngleLimits(3*leg+motor,
                                  m_cal[3*leg + motor][1],
                                  m_cal[3*leg + motor][2]);
            act->writeCommand();
        }
    }
    */

    m_configured = true;
}

void Robot::disableArms()
{
    if( !m_bus )
        return;

    m_bus->setTorqueEnabled( 0xFE, false );
```

```
    m_bus->setTorque( 0xFE, 0 );
}

void Robot::enableArms()
{
    if( !m_bus )
        return;

    m_bus->setTorqueEnabled( 0xFE, true );
    m_bus->setTorque( 0xFE, 0 );
}

/**
    Angle in degrees with 0 being no change to the coordinate system

    id is used to correct for errors in different
    @return 0 -> 150 deg in actuator coordinates
*/
float Robot::simToActuator( int id, float angle )
{
    float a = angle;
    if( (id - 1) % 3 == 1 )
        a = m_cal[id][0] - angle;
    if( (id - 1) % 3 == 2 )
        a = m_cal[id][0] + angle;

    if( id % 3 == 2 )
        a -= 95;

    // heading
    if( (id - 1) % 3 == 0 )
    {
        int leg = (id-1) / 3 + 1;
        switch( leg )
        {
            case 1:
                a = m_cal[id][0] - angle;
                break;
            case 2:
                if( angle > 90 )
                    angle -= 360;
                a = m_cal[id][0] - angle - 120;
                break;
            case 3:
                a = m_cal[id][0] - angle + 120;
                break;
            case 4:
                a = m_cal[id][0] - angle;
                break;
            case 5:
                a = m_cal[id][0] - angle - 120;
                break;
            case 6:
                a = m_cal[id][0] - angle + 120;
                break;
        }
    }
```

```
    print_error( ``id: %d    angle: %f   out: %f ``, id, angle, a );
    return a;
   /*
    if( id % 3 == 2  )
        a = m_cal[id][0] - angle;

    else if( id % 3 == 2 )
    {
        int leg = (id / 3) + 1;
        switch (leg)
        {
            case 1:
                a = m_cal[id][0] - angle;
                break;
            case 2:
                if( angle > 90 )
                    angle -= 360;
                a = m_cal[id][0] - angle - 120;
                break;
            case 3:
                a = m_cal[id][0] - angle + 120;
                break;
            case 4:
                a = m_cal[id][0] - angle;
                break;
            case 5:
                a = m_cal[id][0] - angle - 120;
                break;
            case 6:
                a = m_cal[id][0] - angle + 120;
                break;
        }

    }
    else
        a = angle + m_cal[id][0];
    if( a > m_cal[id][2] )
        a = m_cal[id][2];
    else if ( a < m_cal[id][1] )
        a = m_cal[id][1];

    return a;
    */
}

/**
    This simulator has angles in heading / pitch / roll of shoulder
    The robot is configured with roll / heading / pitch
*/
osg::Vec4 simAnglesToActAngles( const osg::Vec4& ang )
{
    osg::Vec3 fa(0,1,0);
    /*
    osg::Matrix rotate_h = osg::Matrix::rotate( ang[0], 0, 0, 1 );
    osg::Matrix rotate_p = osg::Matrix::rotate( ang[0], 1, 0, 0 );
    osg::Matrix rotate_r = osg::Matrix::rotate( ang[0], 0, 1, 0 );
```

```
    fa = rotate_r * rotate_p * rotate_h * fa;
    */

    // roll = 0
    osg::Vec4 rtn;
    rtn[0] = ang[2];
    rtn[1] = ang[0];
    rtn[2] = ang[1];
    rtn[3] = ang[3];

    return rtn;
}

/**
    Sets the 0 position on all of the motors
*/
void Robot::calibrateZero()
{
    for( unsigned int i = 0; i < m_arms.size(); i++ )
    {
        int id = i * 10;
        for( int j = 0; j < 4; j++ )
        {
            id = i * 10 + j + 1;
            // todo read motor position
        }
    }
}

void Robot::updateActuators()
{
    if( !m_configured )
        return;

    int elapsed = m_time.restart();
    float elapsed_s = elapsed / 1000.0;


    for( int id = 1; id <= 18; id++ )
    {
        int arm_num = (id-1)/3;

        Arm* a = m_arms[arm_num];

        // convert HPRE to HPE
        osg::Vec4 angles = a->getArmAngles();

        int ang_num = (id - 1) % 3;

        if( ang_num == 2 ) ang_num = 3;

            angles[ang_num] = RAD2DEG( angles[ang_num] );
            float goal  = simToActuator( id, angles[ang_num] );
            float speed = fabs((goal - m_last_goals[id])/elapsed_s);
            m_last_goals[id] = goal;
            DX116* act = m_bus->getActuator(id);
            if( act )
```

```
            {
                //print_debug( ‘‘id: %d   input-goal: %f motor-goal: %f   speed: %f", id,
                angles[j], goal, speed );
                //if( id % 10 <= 2 )
                //if( id / 10 == 4 )
                 act->setGoal( goal, speed );
            }
            else
            {
                print_error( ‘‘No such act: %d", id );
                exit(1);
            }
    }

    /*
    for( unsigned int i = 1; i <= m_arms.size(); i++ )
    {
        Arm* a = m_arms[i-1];

        int id = (i-1) * 3 + 1;
        osg::Vec4 angles = simAnglesToActAngles( a->getArmAngles() );
        for( int j = 0; j < 3; j++ )
        {
            id = i * 3 + j;

            angles[j] = RAD2DEG( angles[j] );
            float goal  = simToActuator( id, angles[j] );
            float speed = fabs((goal - m_last_goals[id])/elapsed_s);
        //    if( goal - m_last_goals[id]  == 0 )
        //        continue;
            m_last_goals[id] = goal;
            DX116* act = m_bus->getActuator(id);
            if( act )
            {
                //print_debug( ‘‘id: %d   input-goal: %f motor-goal: %f   speed: %f", id,
                angles[j], goal, speed );
                //if( id % 10 <= 2 )
                //if( id / 10 == 4 )
                    act->setGoal( goal, speed );
            }
        }
    }
    */
}

//===========================================================================
/// @file Arm.h
//===========================================================================
#ifndef _ARM_H_
#define _ARM_H_
#include <math.h>
#include <osg/Vec4>
#include <osg/Vec3>


class Robot;
```

```
/**
    @class Arm
    @breif Performs calculations for a robotic arm



*/
class Arm
{
    public:
        enum STATE {
            SUPPORT = 0,
            LIFT    = 1
        };

        enum ERROR {
            NONE = 0,
            REACHED_RANGE_LIMIT = 1,
            EXCEEDED_MAXIMUM_SH_ANGULAR_VELOCITY = 2,
            EXCEEDED_MAXIMUM_SP_ANGULAR_VELOCITY = 3,
            EXCEEDED_MAXIMUM_SR_ANGULAR_VELOCITY = 4,
            EXCEEDED_MAXIMUM_EP_ANGULAR_VELOCITY = 5,
            EXCEEDED_MAXIMUM_ANGULAR_VELOCITY = 6,
            HIT_STRIDE_LIMIT = 7,
            EXCEEDED_MAXIMUM_ROT_SPEED = 8,
            OUT_OF_WORKSPACE = 9 // it is not possible to meet the constraints
        };

        Arm( double fa_len = .1524, double ua_len = .127,
             double sh = 0, double sp = 0, double sr = 0, double ep = -M_PI/2 ); //-M_PI/2*1.5);

        ERROR testArmMove( const osg::Vec3& delta_xyz, float delta_sec,
                           osg::Vec4*       ang_vel = 0 )const;

        osg::Vec3 getShoulderPosition()const;
        void      setShoulderPosition( const osg::Vec3& xyz );

        osg::Vec3 getShoulderOrientation()const;
        void      setShoulderOrientation( const osg::Vec3& hpr );

        void   setArmLength( double s_to_e, double e_to_t );
        double getUpperArmLength()const;
        double getLowerArmLength()const;
        double getArmLength()const { return getUpperArmLength() + getLowerArmLength(); };

        ERROR setTipPosition( const osg::Vec3& txyz );
        osg::Vec3 getTipPosition()const;

        void setArmAngles( const osg::Vec4& angles );
        osg::Vec4 getArmAngles()const;


        void  calculateTipPosition();
        ERROR calculateArmAngles();

        void setState( int s );
        int  getState()const { return m_state; }
```

```
        osg::Vec3 calculateTipPosition( const osg::Vec4& ang )const;

    private:
        osg::Vec4 calculateArmAngles( const osg::Vec3&, int* error = 0)const;

        int   m_state;

        double m_fa_length;
        double m_ua_length;

        osg::Vec3 m_tip;

        osg::Vec3 m_shoulder_xyz;
        osg::Vec3 m_shoulder_hpr;

        double m_sh; // shoulder heading
        double m_sp; // shoulder pitch
        double m_sr; // shoulder roll
        double m_ep; // elbo pitch
};

#endif


//==========================================================================
/// @file Arm.cpp
//==========================================================================
#include ‘‘Arm.h"
#include <osg/Matrix>
#include <osg/Vec3>

#define RAD2DEG(X) (X/M_PI*180.0)
#define SQ(X) (X*X)
#define MAX_ANG_VEL ((114./60.)*2.*M_PI)  // 114 RPM -> 11.7286 rad/sec

Arm::Arm( double fa_len, double ua_len,
          double sh, double sp, double sr, double ep )
{
    m_fa_length = fa_len;
    m_ua_length = ua_len;

    m_sh = sh;
    m_sp = sp;
    m_sr = sr;
    m_ep = ep;

    m_state = LIFT;

    calculateTipPosition();
}

void Arm::setState( int s )
{
    m_state = s;
}
/**
```

```
    This method will test to see if it is possible to move the arm
    tip by the delta_xyz in arm coordinates.
*/
Arm::ERROR Arm::testArmMove( const osg::Vec3& delta_xyz, float delta_sec,
                                     osg::Vec4* ang_vel )const
{
    osg::Vec4 current_angles(m_sh, m_sp, m_sr, m_ep);
    int err = Arm::NONE;

    osg::Vec4 new_angles = calculateArmAngles( m_tip + delta_xyz, &err );
    if( err != Arm::NONE )
    {
        return (Arm::ERROR)err;
    }

    osg::Vec4 avel = new_angles - current_angles;

    for( int i = 0; i < 4; i++ )
    {
        if( avel[i] > M_PI )
            avel[i] -= 2*M_PI;
        else if( avel[i] < -M_PI )
            avel[i] += 2*M_PI;
        if( fabs(avel[i]) > MAX_ANG_VEL * delta_sec )
        {
             if( ang_vel )
                 *ang_vel = avel;
//            printf( ‘‘ang vel[%d] = %f   MAX == %f\n", i, avel[i], MAX_ANG_VEL * delta_sec );
            return (Arm::ERROR)(EXCEEDED_MAXIMUM_SH_ANGULAR_VELOCITY+i);
        }
    }
    if( ang_vel )
        *ang_vel = avel;
    return Arm::NONE;
}

/**
    @param ua - upper arm length
    @param fa - fore arm length
*/
void Arm::setArmLength( double ua, double fa )
{
    m_fa_length = fa;
    m_ua_length = ua;
}

double Arm::getUpperArmLength()const
{
    return m_ua_length;
}

double Arm::getLowerArmLength()const
{
    return m_fa_length;
}

Arm::ERROR Arm::setTipPosition( const osg::Vec3& pos )
```

```
{
    m_tip = pos;
    return  calculateArmAngles();
}

osg::Vec3 Arm::getTipPosition()const
{
    return m_tip;
}

osg::Vec4 Arm::getArmAngles()const
{
    return osg::Vec4( m_sh, m_sp, m_sr, m_ep );
}

void Arm::setArmAngles( const osg::Vec4& hpr_ep )
{
    m_sh = hpr_ep[0];
    m_sp = hpr_ep[1];
    m_sr = hpr_ep[2];
    m_ep = hpr_ep[3];
    calculateTipPosition();
}

osg::Vec3 Arm::calculateTipPosition( const osg::Vec4& angles )const
{
    osg::Matrix rotate_h = osg::Matrix::rotate( angles[0], 0, 0, 1 );
    osg::Matrix rotate_p = osg::Matrix::rotate( angles[1], 1, 0, 0 );
    osg::Matrix rotate_r = osg::Matrix::rotate( angles[2], 0, 1, 0 );
    osg::Matrix translate_ua = osg::Matrix::translate( 0, m_ua_length, 0 );
    osg::Matrix rotate_ep = osg::Matrix::rotate( angles[3], 1, 0, 0 );
    osg::Matrix translate_fa = osg::Matrix::translate( 0, m_fa_length, 0 );

    osg::Matrix transform = translate_fa * rotate_ep *
                            translate_ua * rotate_r * rotate_p * rotate_h;

    osg::Vec3  point(0,0,0);
    return point * transform;
}

/**
    Determines the position of the tip relative to the
    base based upon the arm angles
*/
void Arm::calculateTipPosition()
{
/*
    printf( ‘‘source angles( sh: %f, sp: %f, sr: %f, ep: %f, ua: %f  fa: %f )\n", RAD2DEG(m_sh),
                                                               RAD2DEG(m_sp),
                                                               RAD2DEG(m_sr),
                                                               RAD2DEG(m_ep), m_ua_length, m_fa_length );
                                                               */
    osg::Matrix rotate_h = osg::Matrix::rotate( m_sh, 0, 0, 1 );
    osg::Matrix rotate_p = osg::Matrix::rotate( m_sp, 1, 0, 0 );
    osg::Matrix rotate_r = osg::Matrix::rotate( m_sr, 0, 1, 0 );
    osg::Matrix translate_ua = osg::Matrix::translate( 0, m_ua_length, 0 );
    osg::Matrix rotate_ep = osg::Matrix::rotate( m_ep, 1, 0, 0 );
```

```
    osg::Matrix translate_fa = osg::Matrix::translate( 0, m_fa_length, 0 );

    osg::Matrix transform = translate_fa * rotate_ep *
                            translate_ua * rotate_r * rotate_p * rotate_h;

    osg::Vec3  point(0,0,0);
    m_tip = point * transform;

 //   printf( ``calculated tip( %f, %f, %f )\n", m_tip.x(), m_tip.y(), m_tip.z() );

//    osg::Vec4 ang = calculateArmAngles( m_tip );

/*
    printf( ``rev_calc angles: sh: %f, sp: %f, sr: %f ep: %f\n", RAD2DEG(ang[0]), RAD2DEG(ang[1]),
    RAD2DEG(ang[2]), RAD2DEG(ang[3]) );
    */
}
/**
    Calculates the angles in the arm based upon the position of the tip
*/
osg::Vec4 Arm::calculateArmAngles( const osg::Vec3& tip, int* error  )const
{
    //return calculateArmAngles2( tip, error );
    //return calculateArmAngles3( tip, error );

   // printf( ``source tip( %f, %f, %f )\n", tip.x(), tip.y(), tip.z() );

    if( tip.length2() < 0.0001 )
    {
        printf( ``OUT_OF_WORKSPACE tip (%f, %f, %f)\n", tip.x(), tip.y(), tip.z()  );
        if( error )
        {
            *error = OUT_OF_WORKSPACE;
            printf( ``*error = %d\n", *error );
        }
        return osg::Vec4( 0, 0, 0, 0 );
    }


    double A2 = SQ(tip.x()) + SQ(tip.y()) + SQ(tip.z());
    double A  = sqrt(A2);
    double U  = m_ua_length;
    double U2 = SQ(U);
    double F  = m_fa_length;
    double F2 = SQ(F);

    //printf( ``x: %f y: %f z: %f\n", tip.x(), tip.y(), tip.z() );
    //printf( ``f = (F2 - U2 - A2) / (-2 * A * U )\n %f = (%f - %f - %f) / (-2*%f*%f )\n",
    //        (F2 - U2 - A2) / (-2 * A * U ),
    //        F2, U2, A2, A, U );

    double ang = (F2 - U2 - A2) / (-2 * A * U );
    if( fabs(ang) > 1 )
    {
        printf( ``OUT_OF_WORKSPACE tip (%f, %f, %f), ang == %f\n", tip.x(), tip.y(), tip.z(), ang  );
        if( error )
            *error = OUT_OF_WORKSPACE;
```

```
        return osg::Vec4( 0, 0, 0, 0 );
    }

    double f = acos( ang );

//    printf( ``f: %f\n", RAD2DEG(f) );
//    printf( ``f-Ps: %f\n", RAD2DEG( atan2( m_tz, sqrt( SQ(m_tx) + SQ(m_ty) ) ) ) );
    double d = atan2( tip.z(), sqrt( SQ(tip.x()) + SQ(tip.y()) ) );
    double Ps = 0;
    if( f > d )
        Ps = f + d;
    else
        Ps = f - d;

    double a = acos( (A2 - F2 - U2)/ (-2*F*U) );
//    printf( ``a: %f\n", RAD2DEG(a) );
    double Pe = a - M_PI;

    double Hs = atan2( -tip.x(), tip.y() );

    return osg::Vec4( Hs, Ps, 0, Pe );


/*
    printf( ``calculated angles( sh: %f, sp: %f, sr: %f, ep: %f )\n", RAD2DEG(m_sh),
                                                                  RAD2DEG(m_sp),
                                                                  RAD2DEG(m_sr),
                                                                 RAD2DEG(m_ep) );
                                                                 */
}


/**
    Calculates the angles in the arm based upon the position
*/
Arm::ERROR Arm::calculateArmAngles()
{
//    printf( ``source tip( %f, %f, %f )\n", m_tip.x(), m_tip.y(), m_tip.z() );
    int err = NONE;
    osg::Vec4 angles = calculateArmAngles( m_tip, &err );
    if( err != NONE )
        return ERROR(err);
    m_sh = angles[0];
    m_sp = angles[1];
    m_sr = angles[2];
    m_ep = angles[3];
    return (ERROR)err;
/*
    printf( ``calculated angles( sh: %f, sp: %f, sr: %f, ep: %f )\n", RAD2DEG(m_sh),
                                                                  RAD2DEG(m_sp),
                                                                  RAD2DEG(m_sr),
                                                                  RAD2DEG(m_ep) );
                                                                  */
}
```

```
/**
    @returns the vector from the center of the robot to the
    shoulder joint in robot coordinates
*/
osg::Vec3 Arm::getShoulderPosition()const
{
    return m_shoulder_xyz;
}

/**
    @param xyz - the vector from the center of the robot to the
                 shoulder joint in robot coordinates
*/
void Arm::setShoulderPosition( const osg::Vec3& xyz )
{
    m_shoulder_xyz = xyz;
}

/**
    @returns the heading/pitch/roll of the shoulder coordinate space
*/
osg::Vec3 Arm::getShoulderOrientation()const
{
    return m_shoulder_hpr;
}

/**
    @param hpr - the heading/pitch/roll of the shoulder
                 coordinate space
*/
void Arm::setShoulderOrientation( const osg::Vec3& hpr )
{
    m_shoulder_hpr = hpr;
}
```

Vita

Mark Showalter was born in Nashville, Tennessee to his expectant mother, father, and sister. At eight months of age he and his family moved to Waynesboro, Virginia, where his parents still reside. At the age of five Mark and his family left for a three-year term of service with the Mennonite Central Committee (MCC) in Bangladesh. There his father helped to improve agricultural methods and his mother worked in health care. After returning to Virginia, Mark graduated from Eastern Mennonite High School in 1998. Earning his Associates Degree from Blue Ridge Community College in 2000, Mark then served for a year as an AmeriCorps member at the Mitchell-Yancy Habitat for Humanity affiliate in North Carolina. Mark earned his Bachelors Degree in Mechanical Engineering from Virginia Tech in 2005. During his undergraduate studies, Mark worked for a total of twelve months over two terms as a co-op with ExxonMobil. In anticipation of his senior year, Mark helped to start the Renewable Energy Senior Design Project in Virginia Tech Mechanical Engineering. Having worked in Virginia Tech Robotics and Mechanisms Laboratory (RoMeLa) as an undergraduate researcher, Mark joined RoMeLa as a graduate student, earning his Masters Degree in Mechanical Engineering in 2008. While pursuing his Masters Degree, Mark has served as a graduate advisor for the Renewable Energy Senior Design Project, volunteered as a mentor for the First Robotics Team 401, worked in collaboration with Open Tech building a serpentine robot, and worked in the Virginia Tech Department of Industrial and Systems Engineering on a sea state simulation platform.

Bibliography

[1] Jet Propulsion Laboratory. *LEMUR: Legged Ecursion Mechanical Utility Rover*, number 11, Pasadena, CA, USA, 2001. Kluwer Academic Publishers.

[2] Brett Kennedy, Hrand Agazarian, Yang Cheng, Michael Garrett, Terry Huntsberger, Lee Magnone, Avi Okon, and Matthew Robinson. Limbed excursion mechanical utility rover: Lemur ii. Technical report, California Institute of Technology, 4800 Oak Grove Drive, M/S 82-105, Pasadena, California 91109.

[3] Brett Kennedy, Avi Okon, Hrand Agazarian, Michael Garrett, Terry Huntsberger, Lee Magnone, Matthew Robinson, and Julie Townsend. The lemur ii-class robots for inspection and maintenace of orbital structures: A system description. Technical report, California Institute of Technology, 4800 Oak Grove Drive, M/S 82-105, Pasadena, California 91109.

[4] R. Wagner, L. Hobson., and Brett Kennedy. Autonomous walking inspection and maintenance robot. In *Advances in the Astronoutical Sciences*, volume 121, pages 235–249, 2005.

[5] Kenneth S. Espenchied, Roger D. Quinn, Hillel J. Chiel, and Randall D. Beer. Leg coordination mechanisms in the stick insect applied to hexapod robot locomotion. volume 1, pages 455–468. Adaptive Behavior, 1993.

[6] Kenneth S. Espenchied, Roger D. Quinn, Randall D. Beer, and Hillel J. Chiel. Biologically based distributed control and local reflexes improve rough terrain locomotion in a hexapod robot. Number 18, pages 59–64. Robotics and Autonomous Systems, 1996.

[7] Thomas J. Allen, Roger D. Quinn, Richard J. Bachmann, and Roy E. Ritzmann. Abstracted biological principles applied with reduced actuation improve mobility of legged vehicles.

[8] Roger D. Quinn and Roy E. Ritzmann. Construction of a hexapod robot with cockroach kinematics benefits both robotics and biology. volume 10, pages 239–254. Connection Science, 1998.

[9] Hillel J. Chiel, Randall D. Beer, Roger D. Quinn, and Kenneth S. Espenchied. Robustness of a distributed neural network controller for locomotion in a hexapod robot. volume 8, pages 293–303. IEEE Transactions on Robotics and Automation, June 1992.

[10] S. M. Song and K. J. Waldron. *Machines That Walk: The Adaptive Suspension Vehicle*. MIT Press, Cambridge, 1989.

[11] Hong-Sen Yan. *History of Mechanism and Machine Science Volume 3 - Reconstruction Designs of Lost Ancient Chineese Machinery*. Springer, 2007.

[12] L. A. Rygg. Mechanical horse. Patent 491,927, February 14 1893.

[13] A. Ehrlich. Vehicle propelled by steppers. Patent 1,691,233, November 13 1928.

[14] E. Snell. Reciprocating load carrier. Patent 2,430,537, November 11 1947.

[15] W. E. Urschel. Walking tractor. Patent 2,491,064, December 13 1949.

[16] H. W. Wallace. Vehicle. Patent 2,371,368, March 13 1945.

[17] *Legged Robots*, volume 29, June 1986.

[18] D. J. Todd. *Walking Machines an Introduction Legged Robots*. Korgan Page Ltd., 1985.

[19] Marc D. Donner. *Real-Time Control of Walking*, volume 7 of *Progress in Computer Science*. Dirkhäuser, 1987.

[20] Robert T. Schroer, Matthew J. Boggess, Richard J. Bachmann, Roger D. Quinn, and Roy E. Ritzmann. Comparing cockroach and whegs robot body motions. New Orleans, 2004. IEEE Conference on Robotics and Automation (ICRA '04). Case Western Reserve University, Cleveland, Ohio, U.S.A.

[21] Rodney A. Brooks. A robust layered control system for a mobile robot. volume RA-2, pages 14–23. IEEE Journal of Robotics and Automation, March 1986.

[22] Rodney A. Brooks. A robot that walks; emergent behaviors from a carefully evolved network. pages 692–696, 545 Technology Square, Cambridge, MA 02139, 1989. IEEE.

[23] ASME International Design Engineering Technical Conferences and Computers and Information in Engineering Conference. *Maximizing Walking Step Length for a Near Omni-Directional Hexapod Robot*, number DETC2004/MECH-57531, September - October 2004.

[24] Lung-Wen Tsai. *Robot Analysis The Mechanics of Serial and Parallel Manipulators*. John Wiley and Sons, Inc., 1999.

[25] Tsuneo Yoshika. *Foundations of Robotics Analysis and Control*. The MIT Press, 1990.

[26] Marn W. Spong and M. Vidyasagar. *Robot Dyanmics and Control*. John Wiley Sons, 1989.

[27] Department of Engineering and Applied Sciences. *Arthropod Grasping and Manipulation a Literature Review*, April 2001.

[28] Doug Bowman. Power Point slides detailing a method for finding the intersection points of a line with a sphere from Dr. Bowman's class on 3D graphics.

[29] Peter Wapperom, 2006. Conversation with Peter Wapperom in which he described a method for finding the intersection of a line with a torus.