Energy-efficient Gait Control Schema of a Hexapod Robot with Dynamic Leg Lengths

by Ryan Cafarelli, Master of Science

A Thesis Submitted in Partial
Fulfillment of the Requirements
for the Degree of
Master of Science
in the field of Computer Science

Advisory Committee:

Gary Mayer, PhD, Chair

Dennis Bouvier, PhD

Igor Crk, PhD

Jerry Weinberg, PhD

Graduate School
Southern Illinois University Edwardsville
December, 2017

ProQuest Number: 10684042

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

ProQuest 10684042

Published by ProQuest LLC (2017). Copyright of the Dissertation is held by the Author.

All rights reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

ABSTRACT

ENERGY-EFFICIENT GAIT CONTROL SCHEMA OF A HEXAPOD ROBOT WITH DYNAMIC LEG LENGTHS

by

RYAN CAFARELLI

Chairperson: Professor Gary Mayer, PhD

Walking robots consume considerable amounts of power, which leads to short mission times. Many of the tasks that require the use of walking robots, rather than wheeled, often require extended periods of time between the possibility of charging. Therefore, it is extremely important that, whenever possible, the gait a walking robot uses is as efficient as possible in order to extend overall mission time. Many approaches have been used in order to optimize the gait of a hexapod robot; however, little research has been done on how enabling the leg segments of a hexapod to extend will impact the efficiency of its gait. In this thesis, a joint space model is defined that includes both rotational joints as well as prismatic joints for expanding and contracting individual leg segments. A genetic algorithm (GA) is used to optimize the efficiency of a gait using the joint space based on a tripod gait. Other considerations for the gait include stability and dragging, which affects overall efficiency of a gait. The results of preliminary runs of the GA show the impacts of changing the weights of a multi-objective function, the number of generations, the number of parents retained between generations and the mutation rate. Further experiments show the impact of dynamic leg lengths on the overall efficiency of a hexapod tripod gait.

TABLE OF CONTENTS

ABSTRACT ii

LIST OF FIGURES v

LIST OF TABLES vii

Chapter

1. INTRODUCTION 1

1.1 Hypotheses 1
1.1.1 Flat terrain efficiency 1

2. HEXAPOD ROBOTS 2

2.1 Hexapod Configuration 2
2.2 Hexapod Gait 3

3. ENERGY CONSUMPTION 7

3.1 Prismatic Power 8
3.2 Rotational Power 12

4. INTUITIVE DYNAMICS OF A SINGLE LEG 21

5. GENETIC ALGORITHM 26

5.1 Background 26
5.1.1 Constraint satisfaction 29
5.2 Application 29
5.2.1 Chromosome 30
5.2.2 Reproduction 31
5.2.3 Cross-over operator 31
5.2.4 Mutation operator 32
5.2.5 Fitness function 33
5.2.6 Efficiency objective score 33
5.2.7 Dragging objective score 34
5.2.8 Stability objective score 42

6. EXPERIMENTATION 48

6.1 Experimental Setup . . . 48
6.2 Experimental Results . . . 50
6.2.1 Number of parents kept . . . 50
6.2.2 Number of generations . . . 51
6.2.3 Mutation rate . . . 51
6.2.4 Weight of efficiency . . . 52
6.2.5 Best settings . . . 55
7. CONCLUSION . . . 58
7.1 Future Work . . . 60
REFERENCES . . . 63
APPENDICES . . . 65
A. RAW RESULTS . . . 65
B. GENETIC ALGORITHM CODE . . . 71

LIST OF FIGURES

Figure		Page
2.1	Depiction of an insect leg	2
2.2	Robotic interpretation of an insect leg	2
2.3	Typical leg configurations	3
2.4	Waveform interpretation of different gaits	4
2.5	Robot body from the xy-plane	5
2.6	Joint reference frames	6
3.1	Mechanical aspects of the dynamic leg	8
3.2	Geometry of the force along the axis of the upper leg segment	9
3.3	Geometry of the force along the axis of the lower leg segment	10
3.4	Geometry of the force against the thread of the worm gear	11
3.5	Geometry to help calculate the portion of the robot body force which is in the x-direction and z-direction	16
3.6	Geometry to help calculate the portion of the upper leg segment force which is in the x-direction and z-direction	18
5.1	Example of two chromosomes crossing over to form a child	26
5.2	Example of a chromosomes being mutated	27
5.3	Flow diagram describing a GA	28
5.4	Four phases of the robot gait defined for the GA	30
5.5	Pseudo-algorithm for the cross-over operation	32
5.6	Pseudo-algorithm for the mutation operation	32
5.7	Geometry that describes the distance between the hip and end effector	35
5.8	Robot profile with two legs of the supporting tripod	36
5.9	Robot profile with two legs of the supporting tripod split into triangles to calculate leg height	36
5.10	Angle of the robot body with the ground in the defined plane	38
5.11	Angle of the robot body with the ground in a parallel plane as previously defined but containing the remaining leg in the supporting tripod	39
5.12	Cross section of the robot in the yz-plane including the line whose height is h_2'	40
5.13	Cross section of the robot in the yz-plane including the supported and unsupported legs	41
5.14	Projection of the Center of Gravity within the statically stable region	43
5.15	Statically stable region and projection of the center of gravity from the top profile	45
5.16	Geometry for the line defining the upper limit of the statically stable buffer zone between the end effectors of legs 2 and 3	46
6.1	Dimensions of the experimental hexapod body	48

6.2 Constraints of the experimental hexapod legs 49
6.3 Gait for each leg based on the best settings from the controlled experiments 56

LIST OF TABLES

Table — Page

4.1 Difference in power consumption between dynamic vs. static leg length systems . . . 24
4.2 Difference in distance between dynamic vs. static leg length systems . . . 24
4.3 Difference in efficiency between dynamic vs. static leg length systems . . 25

6.1 Control values for each experiment . . . 50
6.2 Results of parent retention experiment . . . 50
6.3 Results of number of generations experiment . . . 51
6.4 Results of mutation rate experiment . . . 52
6.5 Results of mutation rate experiment based on overall objective score . . 52
6.6 Results of mutation rate experiment based on efficiency . . . 53
6.7 Results of high mutation rate experiment based on overall objective score 53
6.8 Results of high mutation rate experiment based on efficiency . . . 54
6.9 Results of high mutation rate experiment based on the number of trials without dragging and stability . . . 54
6.10 Raw results of the gait based on the best settings from the controlled experiments . . . 57

A.1 Fitness scores of parent retention experiment . . . 66
A.2 Fitness scores of number of generations experiment . . . 66
A.3 Results of mutation rate experiment based on overall objective score . . 67
A.4 Results of mutation rate experiment based on efficiency . . . 67
A.5 Results of high mutation rate experiment based on overall objective score 68
A.6 Results of high mutation rate experiment based on efficiency . . . 69
A.7 Results of high mutation rate experiment based on the trials without dragging and stability . . . 70

CHAPTER 1

INTRODUCTION

Autonomous walking robots are becoming more important in many applications because of the lack of regular terrain or flat surfaces. These terrain types offer challenges for wheeled and tracked vehicles. The downfall to walking robots is that they consume significantly more energy than wheeled and tracked configurations. Because of the increased drain on the battery it is very important for walking robots to have efficient gaits in order to extend the battery life and therefore the mission length. Previous studies have focused on robotic hexapod vehicles with static leg lengths. The concept of dynamic leg-length adds a translational degree of freedom to both leg segments that make up each of the hexapods legs. This thesis presents related works and research to extend their findings to study the effects of allowing for dynamic leg length on the expected energy consumption per unit distance on regular, flat surfaces.

1.1 Hypotheses

1.1.1 Flat terrain efficiency

A leg-length configuration can be found such that the energy consumption per unit distance over a flat terrain will decrease for a hexapod walking robot with dynamic leg-length in comparison to a similar robotic hexapod vehicle with static leg lengths.

CHAPTER 2

HEXAPOD ROBOTS

2.1 Hexapod Configuration

The hexapod leg design is inspired by the leg configuration of an insect. An insect leg is able to rotate horizontally between the thorax and the coxa and vertically in two separate locations along the leg. Figure 2.1 depicts an insect leg and how the segments relate to one another. Figure 2.2 depicts a robotic abstraction of the insect leg.

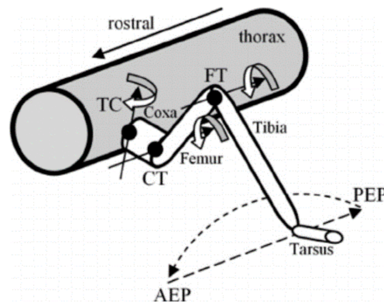

Figure 2.1: Depiction of an insect leg

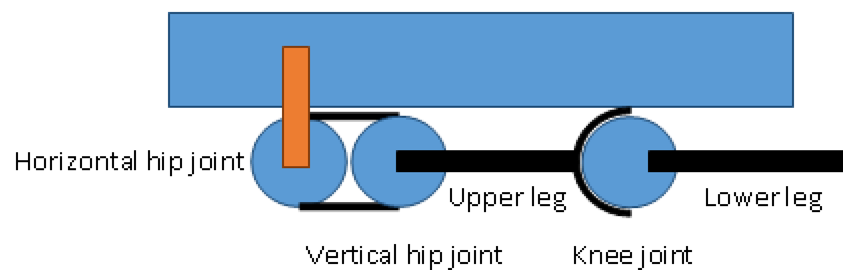

Figure 2.2: Robotic interpretation of an insect leg

A hexapod leg can be oriented to support the body in three types of configuration, as shown in Figure 2.3: insect, reptile and mammal. The insect configuration orients the knee joint above the hip joints, the reptile configuration orients the upper leg directly out from the body, and the mammal configuration orients the knee joint below the hip joint.

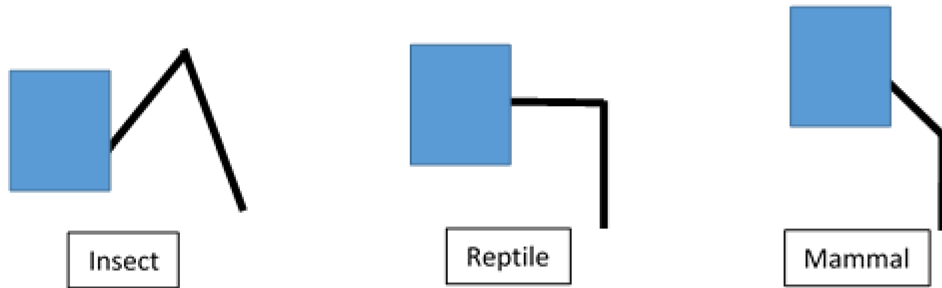

Figure 2.3: Typical leg configurations

In all three cases, the length of the leg segments are assumed static. As discussed in [1], the mammalian leg configuration is the most energy efficient configuration. Therefore, for comparison purposes the mammalian leg configuration will be used.

2.2 Hexapod Gait

The gait of a robot describes the motion of its legs as it walks. In other words, a gait is represented by the trajectories of each of the motors in each leg. Consider a single leg, a gait is represented by the trajectories of the horizontal hip, vertical hip and knee joints over time. One of the advantages to hexapod robots is the ability to move one or more legs at a time, stop motion abruptly, and remain statically stable. Static stability is the ability for a body to remain upright when at rest. Hexapod robots can stay statically stable as long as three of their legs are in contact with the ground and the center of gravity of the mass of the robot lies within the triangle defined by the footholds of each of the legs on the ground. Since a hexapod has six legs, up to three of its legs can be changing their position while the other three maintain stability. This fact allows for a number of commonly used gaits to consider while walking such as ripple, wave and tripod, which are depicted in Figure 2.4. As discussed in [1], a hexapod performing an alternating-tripod gait can achieve its highest speed. Other papers such as [2], [3] and [4] describe other methods for gait generation that lead to specialized gaits based on static leg length systems.

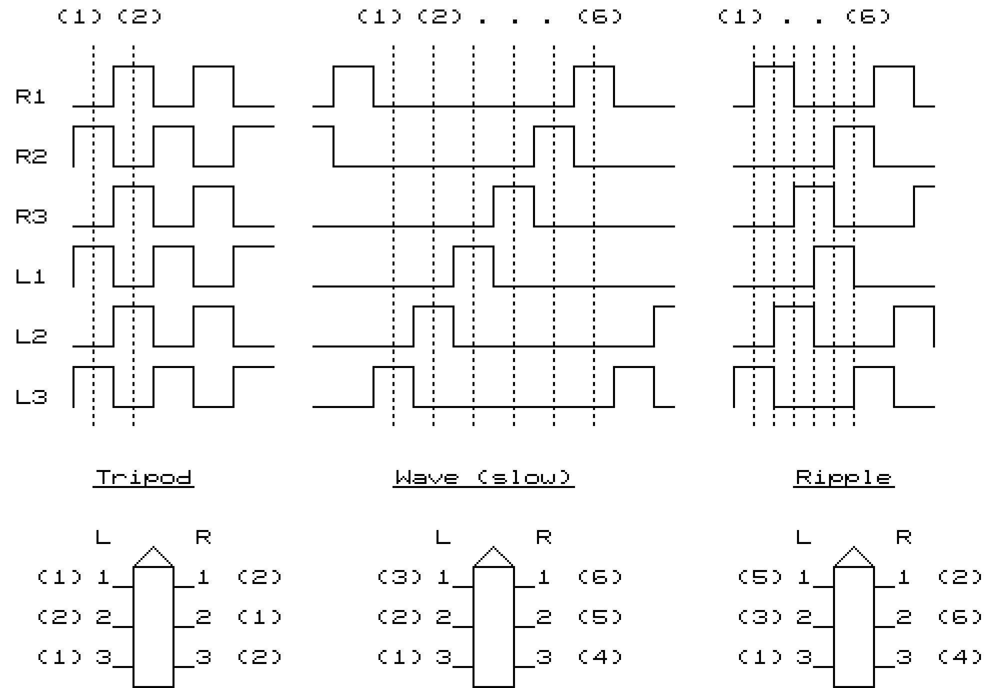

Figure 2.4: Waveform interpretation of different gaits

When analyzing a forward walking gait, only the vertical hip and knee joints need to be considered. This is because the horizontal hip joint mainly controls the turning of the robot [5]. Based on this research the horizontal hip joint of the hexapod will be fixed in the forward direction as the forward gait is optimized.

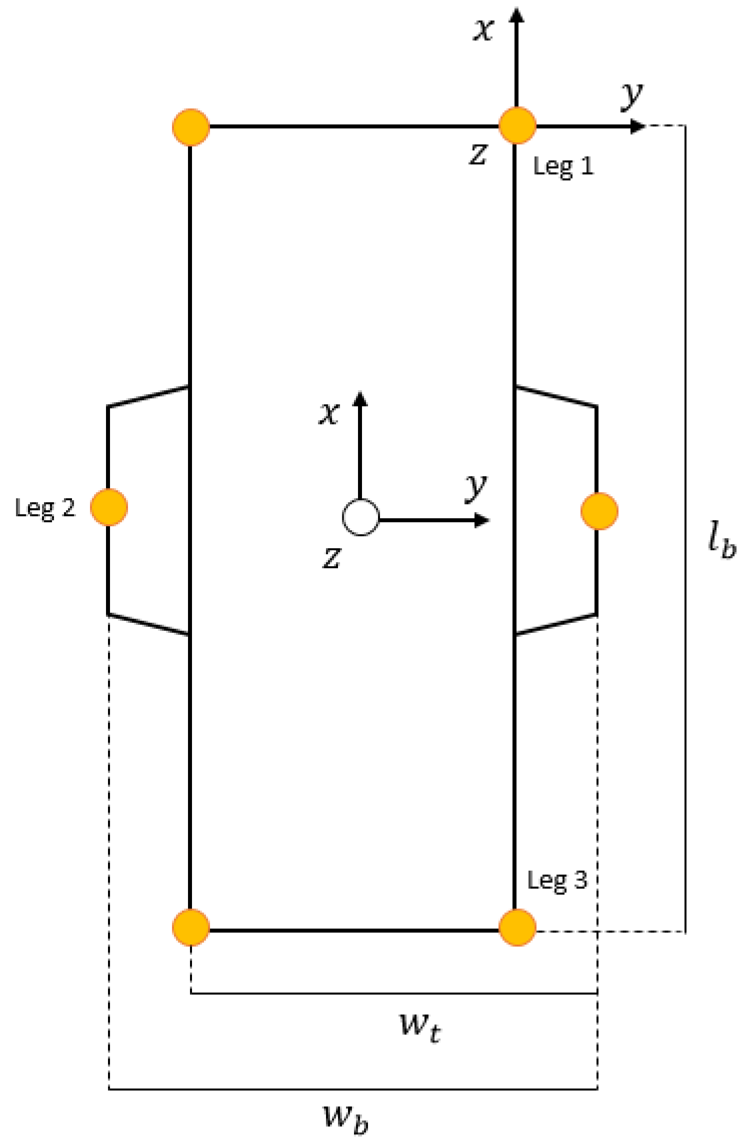

Figure 2.5: Robot body from the xy-plane

Figure 2.5 shows the reference frames for the robot body and one of the hip joints for one of the legs from the worlds xy-plane from above the robot. The figure also shows the outer dimensions of the robot body; l_b is the length of the robot body, w_b is the width of the robot body between the center legs, and w_t is the width of a tripod. Figure 2.6 shows the reference frames for the hip and knee joints.

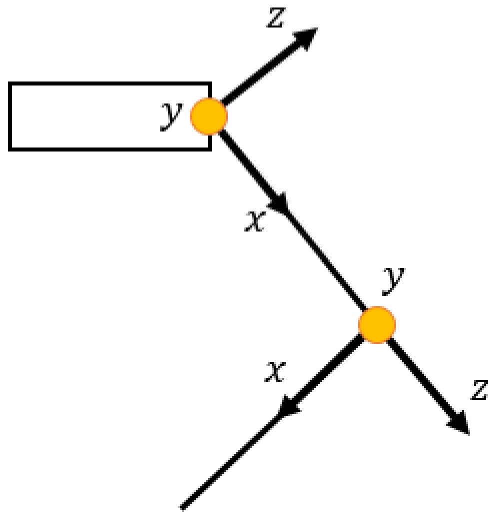

Figure 2.6: Joint reference frames

CHAPTER 3

ENERGY CONSUMPTION

One of the downfalls of walking robots is that they consume a signicant amount of energy compared to their wheeled counterparts. To extend mission time it is important to have an efficient gait. In other words, to extend mission time, the distance traveled per unit of power consumed must be maximized

$$\epsilon = d/P, P \neq 0 \tag{3.1}$$

where d is the normalized distance and P is the overall power for the gait. To calculate the overall power each phase needs to be calculated independently. A phase, as defined in this thesis, is one part of the overall gait. A gait is comprised of four phases, two supported phases and two unsupported phases. Supported phases occur when the tripod being optimized is in contact with the ground and the weight of the robot and the other tripod are being supported. Unsupported phases occur when the tripod being optimized is not in contact with the ground and only each legs own weight is being supported.

$$P = \sum_{phase} P_i \tag{3.2}$$

where P_i is the power consumed during phase i. In order to calculate the power used during a particular phase, the force on the end effector, the foot, can be calculated.

$$P_i = P_r + P_p \tag{3.3}$$

where P_r is the power for the revolute motors and P_p is the power for the prismatic motors.

3.1 Prismatic Power

The prismatic motors being used in this research work like worm gears, as shown in Figure 3.1. The motor is fixed to the outer shell of the leg segment and is attached to a screw that rotates within the leg segment. The inner leg segment is attached to a nut which travels up and down the screw increasing or decreasing the overall length of the leg segment. To calculate the force on the prismatic motors the following variables need to be defined:the mass of the robot body is m_r, he mass of a single leg segment is m_l, the angle between the plane of the robot body and the upper leg segment is q_0, the angle between the upper leg segment and the lower leg segment is q_1, the length of the upper leg segment is l_0 and the length of the lower leg segment is l_1.

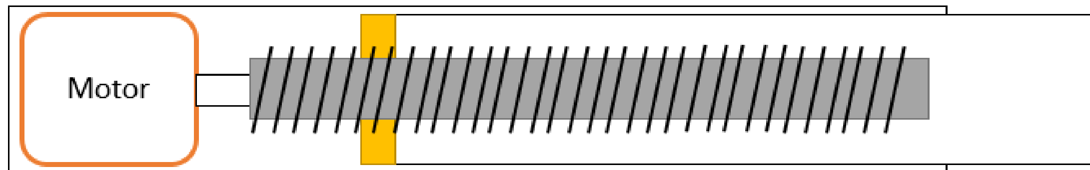

Figure 3.1: Mechanical aspects of the dynamic leg

The first step in calculating the energy consumed by the prismatic motors is to determine the force being applied directly along the axis of the leg segment. Figure 3.2 shows the geometry for calculating the force along the upper leg segment.

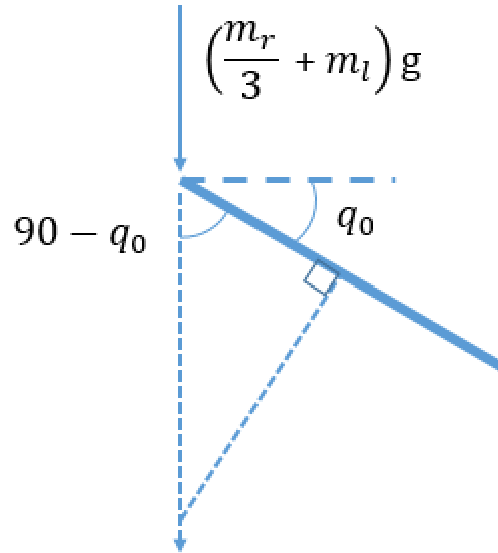

Figure 3.2: Geometry of the force along the axis of the upper leg segment

Using Figure 3.2 the force along the upper leg segment, F_{l_0}, is calculated to be

$$F_{l_0} = \left(\frac{m_r}{3} + 2m_l\right) g \cos(90 - q_0) \tag{3.4}$$

if in a supported phase, and

$$F_{l_0} = 2m_l g \cos(90 - q_0) \tag{3.5}$$

if in an unsupported phase.

Figure 3.3 shows the geometry for calculating the force along the lower leg segment.

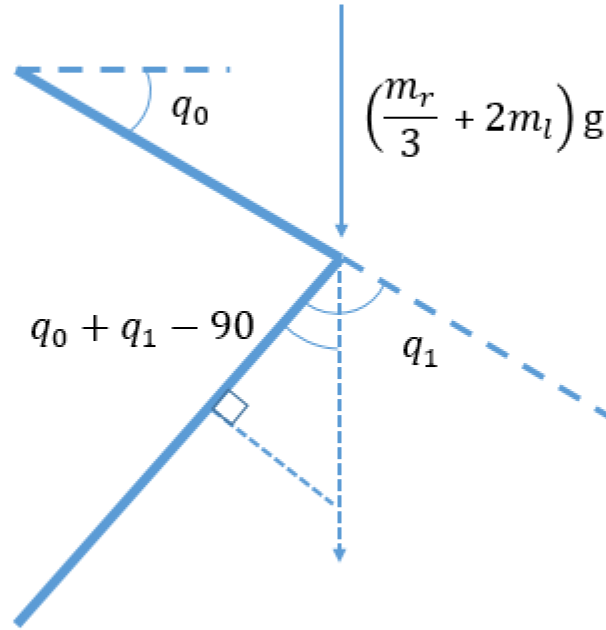

Figure 3.3: Geometry of the force along the axis of the lower leg segment

Using Figure 3.3 the force along the lower leg segment is calculated to be

$$F_{l_1} = \left(\frac{m_r}{3} + 3m_l\right) g\cos(q_0 + q_1 - 90) \tag{3.6}$$

if in a supported phase and

$$F_{l_1} = m_l g\cos(q_0 + q_1 - 90) \tag{3.7}$$

if in an unsupported phase.

The second step in calculating the energy consumed by the prismatic motors is to determine the torque on the motor due to the force along the axis. Figure 3.4 shows the inclined plane which represents a single rotation of the motor along the surface of a thread.

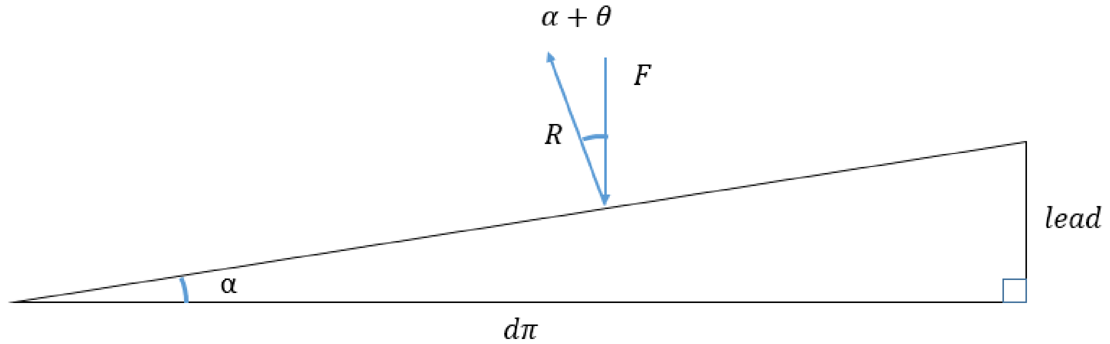

Figure 3.4: Geometry of the force against the thread of the worm gear

Let F represent the force along the axis, R is the normal force perpendicular to the surface, α is the thread angle such that

$$\alpha = \arctan\left(\frac{lead}{d\pi}\right) \tag{3.8}$$

θ is the angle of friction such that

$$\theta = \arctan(\mu) \tag{3.9}$$

where μ is the coefficient of friction, assumed to be 0.5 for plastic-on-plastic friction, d is the diameter which is used to calculate the circumference of the screw, assumed to be 0.01 m, and *lead* is the distance between threads, assumed to be 0.006 m.

The torque on the system, τ, is calculated as

$$\tau = F\frac{d}{2}\tan(\alpha + \theta) \tag{3.10}$$

To calculate the power requirement to expand or contract the leg segment, the torque is multiplied by the speed of a motor and a conversion factor provided by [6].

$$P_p = \sum_{leg\,segment} \tau * RPM * 0.1047 \tag{3.11}$$

where RPM is defined as the number of rotations per minute required to turn the worm gear to shorten or lengthen the leg segment the specified distance in one second. Assuming that each phase occurs in one second, the required RPM for the prismatic motor is

$$RPM = \frac{\Delta l}{lead} * \frac{60\,sec}{1\,min} \tag{3.12}$$

where Δl is the change in leg segment length in meters per second.

3.2 Rotational Power

To calculate the force on the end effector from the revolute joints a Jacobian transformation matrix, $J_{ee}(q)$ must be calculated. The Jacobian is then used to translate the effects of gravitational forces, F_x, due to the robot moving through space, into torque and power required at each of the motors in order to perform a given gait. [7] describes the approach for calculating the translation of force using Jacobians. The Jacobian, J_{ee}, is made up of two parts, linear and angular velocities J_v and J_w, respectively.

$$J_{ee} = \begin{bmatrix} J_v \\ J_w \end{bmatrix} \tag{3.13}$$

Because of previously stated assumptions, the legs of the robot in this system only move in the xz-plane of the holistic robotic system. Therefore, the transformation matrix, T, for the knee of the robot with respect to the hip is

$$T = \begin{bmatrix} R & D \\ 0 & I \end{bmatrix} = \begin{bmatrix} \cos(q_0) & -\sin(q_0) & l_0\cos(q_0) \\ \sin(q_0) & \cos(q_0) & l_0\sin(q_0) \\ 0 & 0 & 1 \end{bmatrix} \tag{3.14}$$

where R is the rotation matrix and D is the translation matrix. From the knee, the end effector reference frame is defined as

$$x = \begin{bmatrix} l_1\cos(q_1) \\ l_1\sin(q_1) \\ 1 \end{bmatrix}, \tag{3.15}$$

a simplified transformation matrix as the orientation of the end effector is not of concern with respect to the knee. Therefore, only a translation matrix is necessary. The end effector reference frame with respect to the hip is defined by simply multiplying the transformation matrices

$$Tx = \begin{bmatrix} \cos(q_0) & -\sin(q_0) & l_0\cos(q_0) \\ \sin(q_0) & \cos(q_0) & l_0\sin(q_0) \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} l_1\cos(q_1) \\ l_1\sin(q_1) \\ 1 \end{bmatrix} = \begin{bmatrix} l_1\cos(q_0+q_1) + l_0\cos(q_0) \\ l_1\sin(q_0+q_1) + l_0\sin(q_0) \\ 1 \end{bmatrix} \tag{3.16}$$

The linear part, J_v, of the Jacobian is defined by taking the partial derivative of the simplified reference matrix

$$J_v = \begin{bmatrix} dx/dq_0 & dx/dq_1 \\ dz/dq_0 & dz/dq_1 \end{bmatrix} = \begin{bmatrix} -l_1\sin(q_0+q_1) - l_0\sin(q_0) & -l_1\sin(q_0+q_1) \\ l_1\cos(q_0+q_1) + l_0\cos(q_0) & l_1\cos(q_0+q_1) \end{bmatrix} \tag{3.17}$$

The angular rotation, ω, is described by the rotation of the leg about an axis. In this

system the leg rotates only about the y-axis. Because both joints are rotating in the same direction about the y-axis the angular rotation is simplified down to

$$\omega = \begin{bmatrix} q_0 + q_1 \end{bmatrix} \tag{3.18}$$

The angular part, J_w, of the Jacobian is defined by taking the partial derivative of the rotation matrix

$$J_w = \begin{bmatrix} d\omega/dq_0 & d\omega/dq_1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \end{bmatrix} \tag{3.19}$$

Therefore, the overall Jacobian for the end effector is

$$J_{ee} = \begin{bmatrix} -l_1 \sin(q_0 + q_1) - l_0 \sin(q_0) & -l_1 \sin(q_0 + q_1) \\ l_1 \cos(q_0 + q_1) + l_0 \cos(q_0) & l_1 \cos(q_0 + q_1) \\ 1 & 1 \end{bmatrix} \tag{3.20}$$

The lack of any reference to the prismatic motors in the preceding matrices is due to (1) the fact that prismatic gears, by definition, operate along a specific coordinate axis and do not create a rotational force, and (2) the aforementioned assumption that the prismatic gears do not allow backdriving of the motor, which negates the need to constantly apply power to the prismatic motors in order to maintain a specific pose.

To determine the force on each of the servos, the force vector representing the end effector forces, F_x, (i.e., the weight of the robotic system as a counter-force against the surface) needs to be translated into the joint space (i.e., the torque created by the motors to keep the robot standing and moving).

$$F_q = J_{ee}^T F_x \tag{3.21}$$

where

$$F_x = \begin{bmatrix} f_x \\ f_z \\ \tau \end{bmatrix} \tag{3.22}$$

where f_x is the force due to gravity in the x-direction, f_z is the force due to gravity in the z-direction, and τ is the torque on the end effector about the y-axis.

During phases in which a leg's end effector is touching the surface (a supporting phase), the forces on the end effector are defined to be the force of a portion of the robot body weight supported by the given end effector, a portion of the weight of the legs from the unsupported tripod, the force imparted from each leg segments own weight, and the torque of the body and leg segment's on the end effector. For simplicity it will be assumed that the force of the robot body is equally distributed between each leg in the supporting tripod. It will also be assumed that the force from the weight of each leg segment includes the mass of the segment as well as the motor at the top of the segment.

During a supporting phase, the mass of the robot body is distributed between the x-axis and z-axis. During an unsupported phase, the mass of the robot and all other leg segments are not taken into account, only the mass of the individual leg segment creates a force (i.e., holding the leg up when gravity is pulling it down). By using Figure 3.5 it can be shown that the force in the x-direction, f_x, is

$$f_x = a\frac{m_r}{3} g \cos(\theta + q_0) \tag{3.23}$$

where g is the acceleration due to gravity ($9.81 m/s^2$) and the force in the z-direction, f_z, is

$$f_z = a\frac{m_r}{3} g \sin(\theta + q_0) \tag{3.24}$$

where the law of cosine is used to calculate the distance between the hip and the end effector of the leg, a, and the angle between the upper leg segment and the line between the hip and end effector, θ,

$$a^2 = l_0^2 + l_1^2 - 2l_0 l_1 \cos(180 - q_1) \tag{3.25}$$

$$\theta = \arccos\left(\frac{l_1^2 - a^2 - l_0^2}{2al_0}\right) \tag{3.26}$$

and m_r is the portion of the mass of the robot, including the unsupported legs, being supported by this end effector.

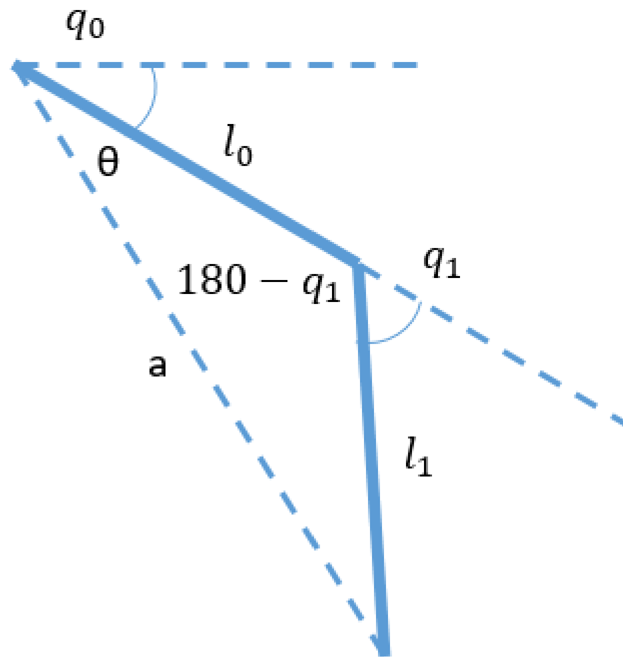

Figure 3.5: Geometry to help calculate the portion of the robot body force which is in the x-direction and z-direction

With the assumption that the end effector intersects the surface of the ground as a point, the forces due to the weights of the robot and respective legs can be represented

as vector forces in the robotic system's x-direction and z-direction. These forces, when offset from the axis of rotation of the shoulder and knee motors, impart a torque that these motors must counter. This torque is proportional to the force at the end effector, and the distance between the end effector and the rotational joint. When the leg is in an unsupported phase, the torque on a leg's rotational joints, τ, is created only by the mass of that leg's segments and the force effect due to gravitational pull upon those segments.

$$\tau = \frac{m_r}{3} g \sin(90 - q_0 - \theta) a \tag{3.27}$$

Similarly, the mass of the upper leg segment is distributed between the x-axis and z-axis. Figure 3.5 shows the geometry that describes the force of the mass of the robot body in the x-direction, f_x, is

$$f_x = b m_l g \cos(\beta + q_0) \tag{3.28}$$

and the force in the z-direction, f_z, is

$$f_z = b m_l g \sin(\beta + q_0) \tag{3.29}$$

where the law of cosine is used to calculate the distance between the end effector and the center of mass for the upper leg segment, b, and the angle between the upper leg segement and the line between the center of mass and end effector, β,

$$b^2 = (\frac{3l_0}{4})^2 + l_1^2 - 2(\frac{3l_0}{4}) l_1 \cos(180 - q_1) \tag{3.30}$$

$$\beta = \arccos\left(\frac{l_1^2 - b^2 - (\frac{3l_0}{4})^2}{2b(\frac{3l_0}{4})}\right) \tag{3.31}$$

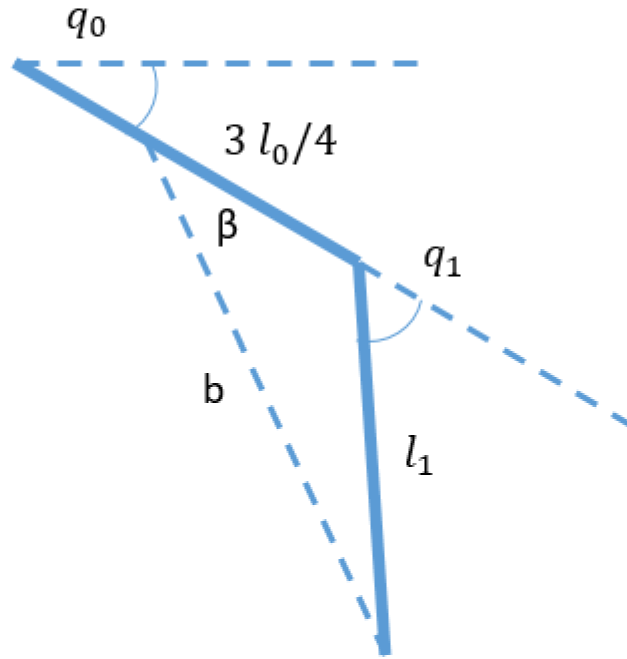

Figure 3.6: Geometry to help calculate the portion of the upper leg segment force which is in the x-direction and z-direction

By using Figure 3.6 it can be shown that the torque, τ, due to the upper leg segment is

$$\tau = m_l g \sin(90 - q_0 - \beta) b \tag{3.32}$$

Finally, the mass of the lower segment is distributed between the x-axis and z-axis. The mass of the lower segment is directly along the leg segment so the calculation is much more straight-forward. The force in the x-direction, f_x, is

$$f_x = \frac{3}{4} l_1 m_l g \cos(q_0 + q_1) \tag{3.33}$$

and the force in the z-direction, f_z, is

$$f_z = \frac{3}{4} l_1 m_l g \sin(q_0 + q_1) \tag{3.34}$$

The torque, τ, due to the lower leg segment is

$$\tau = m_l g \sin\left(90 - q_0 - q_1\right) \left(\frac{3l_1}{4}\right) \tag{3.35}$$

By summing like forces the overall force matrix is

$$F_x = \begin{bmatrix} a\frac{m_r}{3} g \cos(\theta + q_0) + b m_l g \cos(\beta + q_0) + \frac{3}{4} l_1 m_l g \cos(q_0 + q_1) \\ a\frac{m_r}{3} g \sin(\theta + q_0) + b m_l g \sin(\beta + q_0) + \frac{3}{4} l_1 m_l g \sin(q_0 + q_1) \\ \frac{m_r}{3} g \sin(90 - q_0 - \theta) a + m_l g \sin(90 - q_0 - \beta) b + m_l g \sin\left(90 - q_0 - q_1\right)\left(\frac{3l_1}{4}\right) \end{bmatrix} \tag{3.36}$$

With the defined forces and torque the total force can be calculated by transforming them from the x-domain into the q-domain using J_{ee} then summing the integrals of the elements over the change in angle for each joint

$$F = \sum_{F_q} \int_{q_{0_i}}^{q_{0_f}} \int_{q_{1_i}}^{q_{1_f}} F_{q_i} \, dq_1 \, dq_0 \tag{3.37}$$

where F is the total force for a given phase, q_{0_i} and q_{0_f} are the initial and final positions of the q_0 motor, respectively; and q_{1_i} and q_{1_f} are the initial and final positions of the q_1 motor, respectively.

Additional assumptions about the length of each leg segment, l_0 and l_1, need to be made to calculate the total force. The assumption being made is that the overall force for each end effector has an upper bound dependent on the length of the leg segments either at the beginning or end of a phase. The force of each phase is calculated by taking the minimum of the force calculated based on the leg length at the beginning and end of the phase.

Finally, the power of the revolute motors,P_r, is calculated by converting the mechanical force into electrical power. To calculate the power, the calculated force is multiplied by the speed of a motor and a conversion factor provided by [6].

$$P_r = F * RPM * 0.1047 \tag{3.38}$$

where RPM is defined as the number of rotations per minute required to rotate the hip and knee joints the specified angle in one second. Assuming that each phase occurs in one second, the required RPM for the rotational motor is

$$RPM = (\Delta q_0 + \Delta q_1)\, radians/sec * \frac{1\, rotation}{2\pi\, radians} * \frac{60\, sec}{1\, min} \tag{3.39}$$

where Δq_0 is the change in the hip angle in radians per second and Δq_1 is the change in the knee angle in radians per second.

CHAPTER 4

INTUITIVE DYNAMICS OF A SINGLE LEG

In this thesis, the efficiency of different robotic leg systems are compared based on efficiency. The intuition behind the hypothesis is that even with the added mass of additional prismatic motors, the extra distance gained in a gait using dynamic leg length overcomes the additional power consumption. To show the intuition, a comparison will be performed using two static leg systems and one dynamic leg system. Leg system 1 uses a static leg length with leg segments measuring 6 cm. Leg system 2 uses a static leg length with leg segments measuring 12 cm. Finally, Leg system 3 uses a dynamic leg length with leg segments measuring between 6 cm and 12 cm. Each of the leg systems will perform a propel phase where the initial angle measures of the hip and knee joints are 45 degrees. The final angle measures of the hip and knee joints are 135 and 0 degrees, respectfully. Assuming the mass of the robot body is 10 kg, leg segments are 1 kg, the force and distance of each system can be calculated to compare the efficiency of each system.

By using Equation 3.3, the power consumption of each system for a single propulsion phase can be calculated for the dynamic leg system.

To calculate P_p, the forces along the leg segments need to be calculated based on the final position of the leg. Equations 3.4 and 3.6 are used to calculate the forces on the respective leg segments.

$$F_{l_0} = \left(\frac{10}{3} + 2 * 2\right) 9.81 \cos(90 - 135) = 50.87N \tag{4.1}$$

$$F_{l_1} = \left(\frac{10}{3} + 3 * 2\right) 9.81 \cos(135 + 0 - 90) = 64.74N \tag{4.2}$$

α is the thread angle such that

$$\alpha = \arctan\left(\frac{0.006}{0.01\pi}\right) = 0.19\, radians \tag{4.3}$$

θ is the angle of friction such that

$$\theta = \arctan(0.5) = 0.46\, radians \tag{4.4}$$

The torque on the upper leg segment is calculated as

$$\tau_{l_0} = 50.87\, N \frac{0.01\, m}{2} \tan(0.19 + 0.46) = 0.194\, Nm \tag{4.5}$$

The torque on the lower leg segment is calculated as

$$\tau_{l_1} = 64.74\, N \frac{0.01\, m}{2} \tan(0.19 + 0.46) = 0.247\, Nm \tag{4.6}$$

After calculating the torque, the speed is calculated as

$$RPM_{l_0} = RPM_{l_1} = \frac{0.06\, m/sec}{0.006\, m/rotation} * \frac{60\, sec}{1\, min} = 600\, RPM \tag{4.7}$$

Finally, the electrical power for extending both leg segments 6 cm is calculated to be

$$P_p = 0.194\, Nm * 600\, RPM * 0.1047 + 0.247\, Nm * 600\, RPM * 0.1047 = 27.704\, J \tag{4.8}$$

To calculate P_r the Jacobian and the forces acting on the robot must be calculated in order to determine the forces on the motors.

$$F_q = J_{ee}^T F_x \tag{4.9}$$

Assuming that the rotational force is greater for greater leg lengths, the rotational force will be calculated with shorter leg lengths, 6 cm, to minimize the total power. The Jacobian matrix for the dynamic leg system is

$$J_{ee} = \begin{bmatrix} -6\sin(q_0+q_1) - 6\sin(q_0) & -6\sin(q_0+q_1) \\ 6\cos(q_0+q_1) + 6\cos(q_0) & 6\cos(q_0+q_1) \\ 1 & 1 \end{bmatrix} \tag{4.10}$$

and the force in the x-domain is

$$F_x = \begin{bmatrix} 52.32a\cos(\theta+q_0) + 19.62b\cos(\beta+q_0) + 88.29\cos(q_0+q_1) \\ 52.32a\sin(\theta+q_0) + 19.62b\sin(\beta+q_0) + 88.29\sin(q_0+q_1) \\ 52.32\sin(90-q_0-\theta)a + 19.62\sin(90-q_0-\beta)b + 397.305\sin(90-q_0-q_1) \end{bmatrix} \tag{4.11}$$

where

$$a^2 = 72 - 72\cos(180-q_1) \tag{4.12}$$

$$\theta = \arccos\left(\frac{a^2}{12a}\right) = \arccos\left(\frac{a}{12}\right) \tag{4.13}$$

$$b^2 = 56.25 - 54\cos(180-q_1) \tag{4.14}$$

and

$$\beta = \arccos\left(\frac{15.75-b^2}{9b}\right) \tag{4.15}$$

The total force of the rotational motors, F is calculated by integrating each of the

elements in F_q with previously defined integration bounds and summing them.

The speed required to turn the hip and knee joints the specified angle in one second is calculated to be

$$RPM = (\frac{\pi}{2} + \frac{\pi}{4})\, radians/sec * \frac{1\, rotation}{2\pi\, radians} * \frac{60\, sec}{1\, min} = 22.5\, RPM \tag{4.16}$$

Finally, the power is calculated to be

$$P_r = F * 22.5\, RPM * 0.1047 = 34.72\, J \tag{4.17}$$

By applying these calculations to each system, the power consumption for a single propulsion phase within that system is defined in Table 4.1.

Initial Leg Length	Final Leg Length	Mass of Leg Segment	Power Consumption (J)
6	6	1	15.3
12	12	1	27.39
6	12	2	34.72

Table 4.1: Difference in power consumption between dynamic vs. static leg length systems

The distance that each leg system covers is calculated as the difference in hip position between the beginning and end of the phase, in the x-direction. Assuming each leg system uses the same initial and final angles of 45 and 135, respectfully, for the hip joint and 45 and 0, respectfully, for the knee joint.

Initial Leg Length	Final Leg Length	Initial x Position	Final x Position	Distance (m)
6	6	1.76	-8.49	0.13
12	12	3.51	-16.97	0.25
6	12	1.76	-16.97	0.21

Table 4.2: Difference in distance between dynamic vs. static leg length systems

Using the distance and power consumption the overall efficiency is calculated to be

Initial Leg Length	Final Leg Length	Efficiency (m/J)
6	6	0.0085
12	12	0.0091
6	12	0.0060

Table 4.3: Difference in efficiency between dynamic vs. static leg length systems

CHAPTER 5

GENETIC ALGORITHM

5.1 Background

A genetic algorithm (GA) is a type of evolutionary algorithm that uses a search-based approach for finding an optimal solution when a search space is too large to feasibly solve in some fixed time. A GA models an abstraction of natural selection within a species over numerous generations. Each individual within a population is described by a chromosome that is an array of values used to determine an individual's relative fitness within the population. At the end of each generation, the fitness scores of each individual are used to determine the probability of reproducing with another individual within the population. Once two individuals have been selected for reproduction, one or more offspring can be generated using cross-over and mutation operators. The cross-over operator defines one or more positions in a chromosome to be used to replicate sections of each parent, combining them to form a child. An example of this is shown in Figure 5.1.

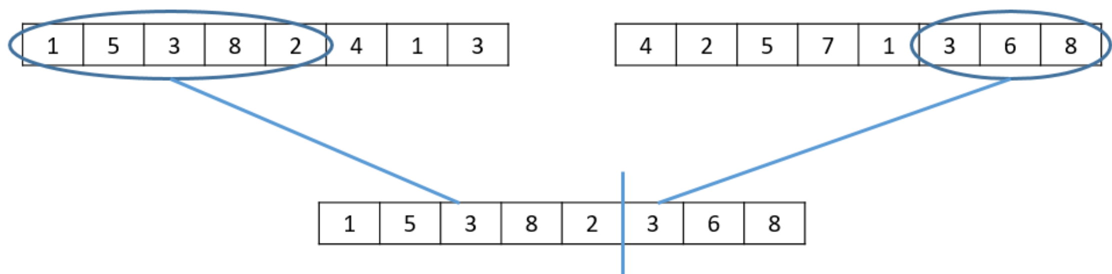

Figure 5.1: Example of two chromosomes crossing over to form a child

Once a new child has been created, the mutation operator is used to update the value of some number of positions in the array. An example of this is shown in Figure 5.2.

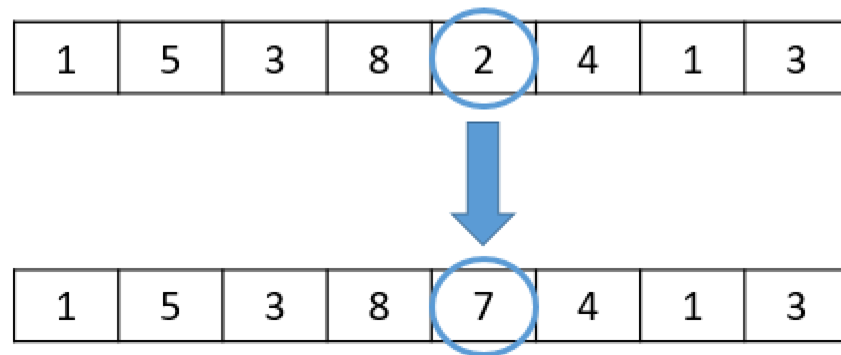

Figure 5.2: Example of a chromosomes being mutated

During each generation, N offspring are generated that are used as the population for the next generation. The algorithm is repeated until a terminating condition such as a fixed number of generations or the convergence of a solution has been met.

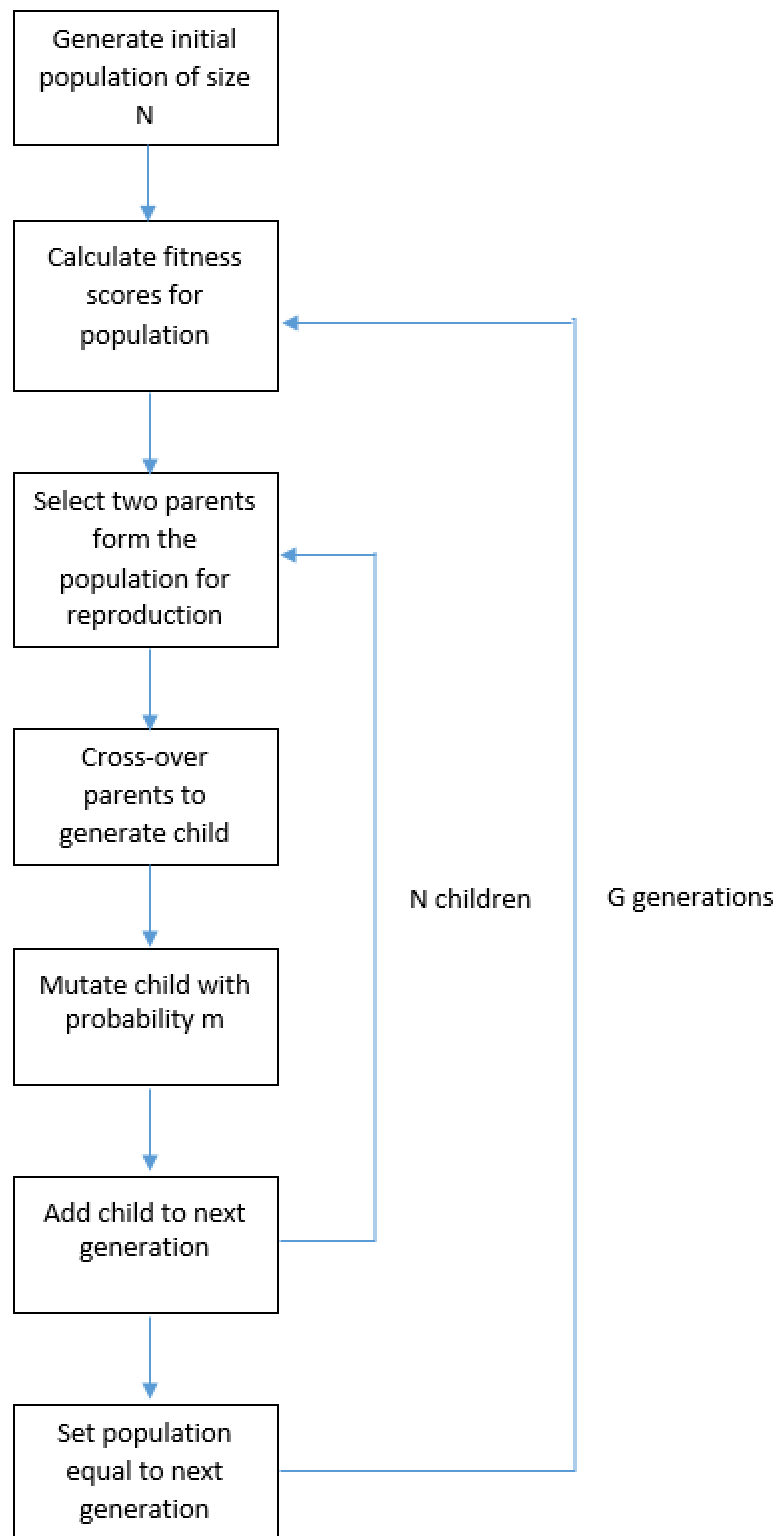

Figure 5.3: Flow diagram describing a GA

5.1.1 Constraint satisfaction

In [8] and [9], a number of different approaches have been suggested for constraining large GAs by adding constraints to the reproductive cycle. Approaches include: eliminating infeasible solutions, repairing infeasible solutions, modifying genetic operators and applying penalties for infeasibility. Eliminating infeasible solutions ensures that all individuals in the population have a feasible genetic makeup but may remove extremely valuable pieces of genetic material. Repairing infeasible solutions can be very beneficial for quickly generating children but it is often difficult to identify the correct part of the chromosome to repair because of potential dependencies within the genetic makeup. Using modified genetic operators allows the implementation of explicit constraints when generating the offspring. Applying penalties for infeasibility allows for infeasible offspring to be generated but reduces the fitness score of the individual, which reduces the probability of reproduction and passing on infeasible genetic material to the next generation.

5.2 Application

GA's and other simulation and optimization techniques have been used in order to determine energy efficient or stable gaits for static leg length hexapod systems. [10] describes mathematical equations to define the oscillation of the end effector over time whose constants are varied in order to determine the most efficient oscillation. [11] describes a search algorithm for finding the most efficient gait over uneven terrain. Other papers using GA or other simulation and optimization techniques include [1], [5], [12], [13], [14], [15] and [16]. None of the papers found during the literature review focused their research on the energy efficiency of dynamic leg length hexapod systems. To determine if dynamic leg segment length can generate more efficient gaits for a hexapod robot a GA is defined. To use a GA to answer this question the problem must be defined by the following: chromosome, reproduction, fitness function and experimental factors.

5.2.1 Chromosome

A chromosome is defined as a complete stride for a robot tripod gait. Each allele represents one of four phases within a stride: front propel, back propel, lift and descend. Each allele describes the position at the beginning of a phase in the tripod gait. Each value in the allele is a double, either representing the angle of a joint or the length of a leg segment. Based on previous work described in previous sections, the act of propulsion in a forward direction is handled by the vertical hip and knee joints. This fact, along with the hypothesis that states by allowing the upper and/or lower leg segments to expand and contract during the gait a more efficient stride can be achieved, each leg in the tripod will have four degrees of freedom (hip and knee joints, upper and lower leg segments). Because of the setup and tripod gait, the assertion is made that only one of the two tripods need to be optimized. The assumption is that the second tripod would achieve the same optimal gait with a two-phase shift.

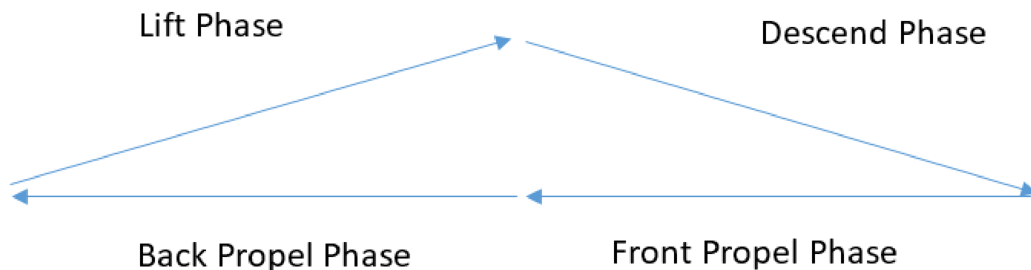

Figure 5.4: Four phases of the robot gait defined for the GA

An allele, which is a subset of the chromosome, is described by the joint space for all legs of a single phase. The joint space of the phase is defined by the 12-ple,

$$A = \langle q_0 1, q_1 1, l_0 1, l_1 1, q_0 2, q_1 2, l_0 2, l_1 2, q_0 3, q_1 3, l_0 3, l_1 3 \rangle \tag{5.1}$$

where q_0 and q_1 are the hip and knee angles for leg i, respectfully, l_0 and l_1 are the

upper and lower leg lengths for leg i, respectfully. With four alleles corresponding to four phases in a chromosome, each chromosome will be a 48-ple.

5.2.2 Reproduction

The first step in generating the next generation of chromosomes is selecting, from the population of chromosomes, one or two parents in order to produce offspring. For the model, parents are chosen in two ways: asexual reproduction of the fittest individuals and sexual reproduction based on randomly selected parents. Asexual reproduction will be performed by the n best individuals based on the overall fitness function. Asexual reproduction will begin with a child chromosome being generated as an exact copy of the parent chromosome. Sexual reproduction will be performed by selecting two parents from the population. The selection process will be a random draw from the population based on the relative fitness of each individual (i.e. chromosome C has a fitness of 10 and the sum of all fitness values is 100, the relative fitness of C is 0.1). After two unique individuals have been selected, the parents will be sent to the cross-over operator to generate a child. After a child has been generated, it will be sent to the mutation operator before being added to the next generation of the population. Sexual reproduction will be performed s times, which generates a total population of $N = n + s$ for the next generation.

5.2.3 Cross-over operator

The cross-over operator begins by taking an input of two parent chromosomes. Because of the dependence between legs within a phase, cross-over points are defined at the boundaries of each phase. To perform the cross-over, a randomly generated integer between zero and three is used to determine the cross-over point. The generated value, which can also represent an allele in the chromosome, is pulled from the first parent to be used in the cross-over along with the next allele in the stride sequence. The remaining two alleles will come from the second parent to create the child.

```
Procedure: cross-over

given (c1 = chromosome, c2 = chromosome) \\ parent chromosomes
a[0,3] = array of alleles
c = chromosome \\ child
n = random integer between 0 and 3 \\ cross-over point
c.a[n] = c1.a[n]
c.a[(n+1)%4] = c1.a[(n+1)%4]
c.a[(n+2)%4] = c2.a[(n+2)%4]
c.a[(n+3)%4] = c2.a[(n+3)%4]
```

Figure 5.5: Pseudo-algorithm for the cross-over operation

5.2.4 Mutation operator

The mutation operator begins by passing in a child chromosome and a mutation rate, r. The operator loops over each element in the chromosome and determines if that element will be mutated by randomly generating a value and comparing it to r. In order to create feasible leg configurations, the standard mutation operator has been modified to apply the physical constraints of each joint or leg segment where appropriate.

```
Procedure: mutate

given (r = mutation rate,  c = chromosome)
for i = 0 to 47
        n = random number
        if n < r then
                c[i] = random number bounded by corresponding joint or segment
        end if
next i
```

Figure 5.6: Pseudo-algorithm for the mutation operation

In addition to adding the constraints of the physical system, a secondary check is performed to eliminate infeasible solutions. This check ensures that each leg in the tripod is supported by its foot and not its knee, and that the overall gait is progressing in the

expected direction within each phase.

5.2.5 Fitness function

A multi-objective fitness function will be used to evaluate each chromosome added to the next generations population. The main objective of this algorithm is to generate the most efficient gait. The most efficient gait is determined by maximizing the distance traveled in the x-direction per unit of energy consumed. Another consideration for the fitness function is whether the legs will fight one another while transitioning through the phases. This is determined using the dragging objective. Finally, consideration is given for the “drunken walk.” A “drunken walk” is one where the center of gravity of the robot moves to a position that increases the risk of stumbling to an unacceptable level. To compare the relative importance of each of the factors in the multi-objective function, each of the factors will be normalized with weights applied to each.

$$Fitness = w_0 * \epsilon + w_1 * Dr + w_2 * St \tag{5.2}$$

where

$$w_0 + w_1 + w_2 = 1 \tag{5.3}$$

The fitness function, $Fitness$, is the value to be maximized.

5.2.6 Efficiency objective score

The efficiency, ϵ, of a gait is defined as

$$\epsilon = \frac{Di}{P} \tag{5.4}$$

where Di is the normalized distance function

$$Di = \frac{D}{2 * L} \tag{5.5}$$

where D is the distance the robot is covering in the x-direction and L is the fully extended leg length; $2 * L$ is the maximum distance any stride could cover if the shoulder is rotated completely forward at the beginning of the propel phases then completely backward at the end of the propel phases. Dividing by the maximum length allows the distance factor, Di, to be standardized within the range $[0, 1]$.

The overall power, P, is as defined in Equation 3.2; however, because the prismatic and rotational power is calculated separately only an upper bound for the overall power can be determined. To define the upper bound of the power, the prismatic power will be calculated at the beginning and the end of the phase. Based on the leg segment lengths associated with the calculation of the prismatic power, the associated rotational power will be calculated. The minimum of the two power calculations will be used as the overall power, P.

5.2.7 Dragging objective score

The dragging objective, Dr, is a binary function that compares the height of the hip of each leg in the middle of the propel phases against the height of the hip between the lift and descend phases. This function is the only consideration for the second tripod in the system, aside from the power requirement for unsupported phases. While the first tripod is in the middle of the propel phases, the second tripod is between the lifting and descending phases. In order for the second tripod to swing freely forward, all of the legs must be higher in the z-direction than in the first tripod; otherwise, the second tripod will drag its feet. If dragging occurs, the dragging objective submits a score of 0 to the genetic algorithm's fitness function. When dragging does not occur, the dragging objective function submits a 1.

To determine the height of each of the hips, the plane defined by the hips of the supporting legs must be defined. The plane will be described by the angle of the body in the robotic system's xz-plane and the yz-plane. These two angles will be used to define whether the center of gravity projection lies within the statically stable region. The first calculation that must be performed is to determine the distance between the hip and the end effector of a given leg. Figure 5.7 describes the distance between the hip and the end effector. The line segment between the hip and the end effector will be referred to by $\overline{hf}$.

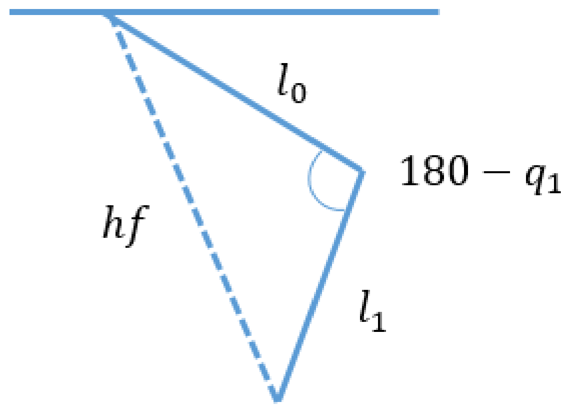

Figure 5.7: Geometry that describes the distance between the hip and end effector

Using Figure 5.7 the distance between the hip and the end effector is calculated to be

$$hf^2 = l_0^2 + l_1^2 - 2l_0 l_1 \cos(180 - q_1) \tag{5.6}$$

The angle of the plane defined by the side of the robot with two legs of the supporting tripod, which is not necessarily in the world space's xz-plane.

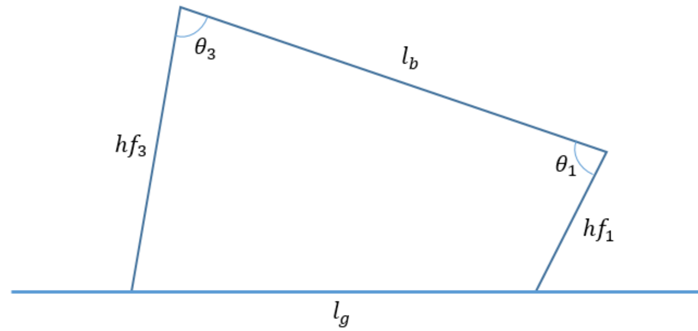

Figure 5.8: Robot profile with two legs of the supporting tripod

In Figure 5.8, θ_i is the angle between $\overline{hf_i}$ and the robot body, l_b is the length of the robot body between legs 1 and 3, and l_g is the distance between the end effectors of legs 1 and 3. The height of each hip, in the defined plane, is calculated by splitting the quadrilateral shown in Figure 5.8 into two triangles which are shown in Figure 5.9.

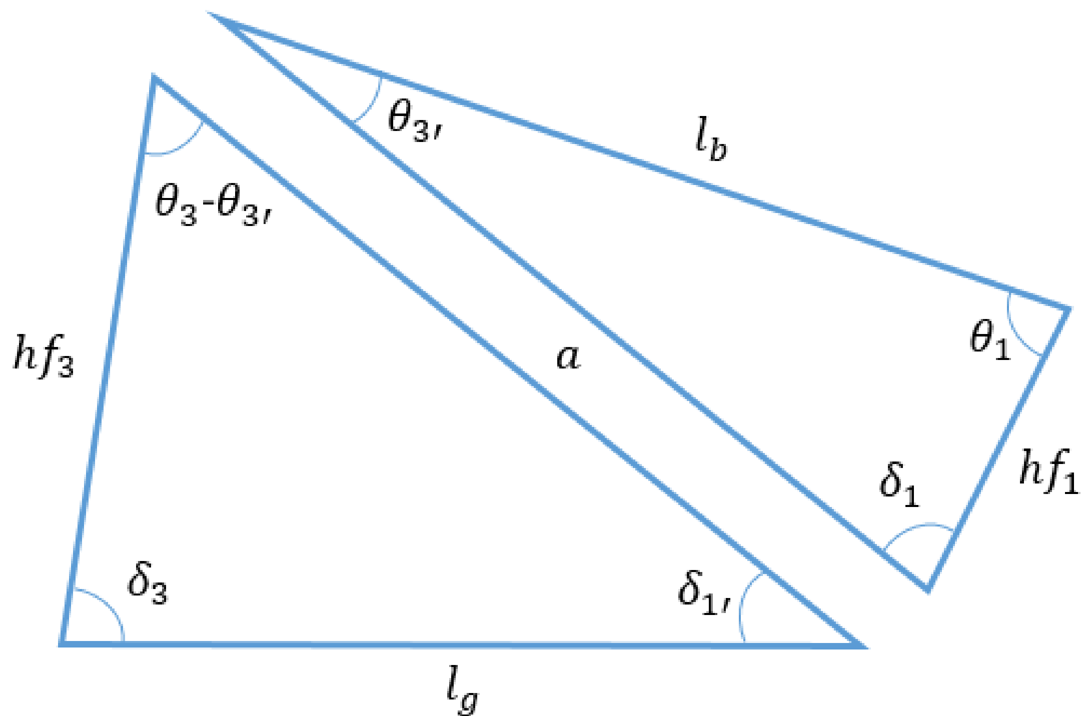

Figure 5.9: Robot profile with two legs of the supporting tripod split into triangles to calculate leg height

Using the two triangles in Figure 5.9 and the law of cosines, the height of each hip in the defined plane is calculated using the following equations:

$$a^2 = hf_1^2 + l_b^2 - 2hf_1 l_b \cos(\theta_1) \tag{5.7}$$

$$\theta_{3'} = \arccos\left(\frac{hf_1^2 - l_b^2 - a^2}{2al_b}\right) \tag{5.8}$$

$$\delta_1 = 180 - \theta_1 - \theta_{3'} \tag{5.9}$$

$$l_g^2 = a^2 + hf_3^2 - 2ahf_3 \cos(\theta_3 - \theta_{3'}) \tag{5.10}$$

$$\delta_{1'} = \arccos\left(\frac{hf_3^2 - l_g^2 - a^2}{2al_g}\right) \tag{5.11}$$

$$\delta_3 = 180 - \delta_{1'} - (\theta_3 - \theta_{3'}) \tag{5.12}$$

Figure 5.10 shows the angle of the robot body with the ground in the defined plane, δ_{13}.

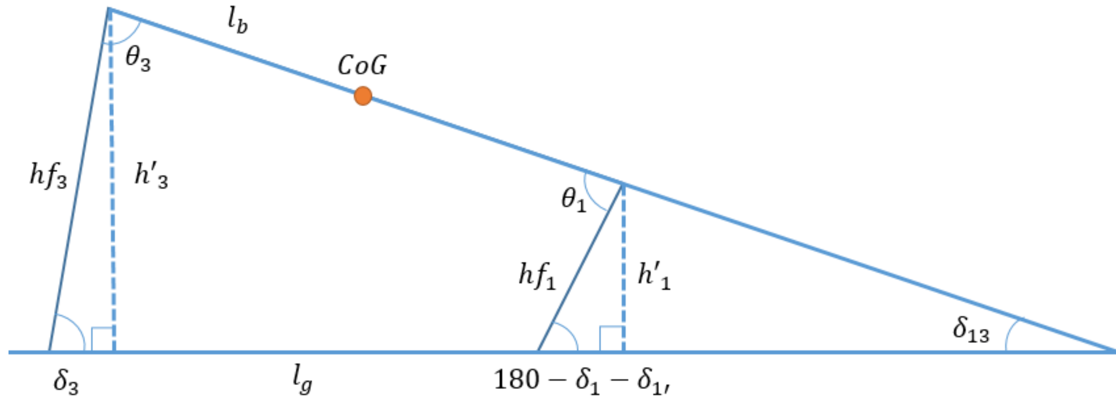

Figure 5.10: Angle of the robot body with the ground in the defined plane

Figure 5.10 also defines the triangles which are used to calculate the height of each hip in the defined plane.

$$\delta_{13} = 180 - \delta_3 - \theta_3 \tag{5.13}$$

$$h_1' = hf_1 \sin(108 - \delta_1 - \delta_{1'}) \tag{5.14}$$

$$h_3' = hf_3 \sin(\delta_3) \tag{5.15}$$

Figure 5.11 describes a plane parallel to the previously define plane and containing leg 2, which is used to calculate the height of leg 2 in that plane.

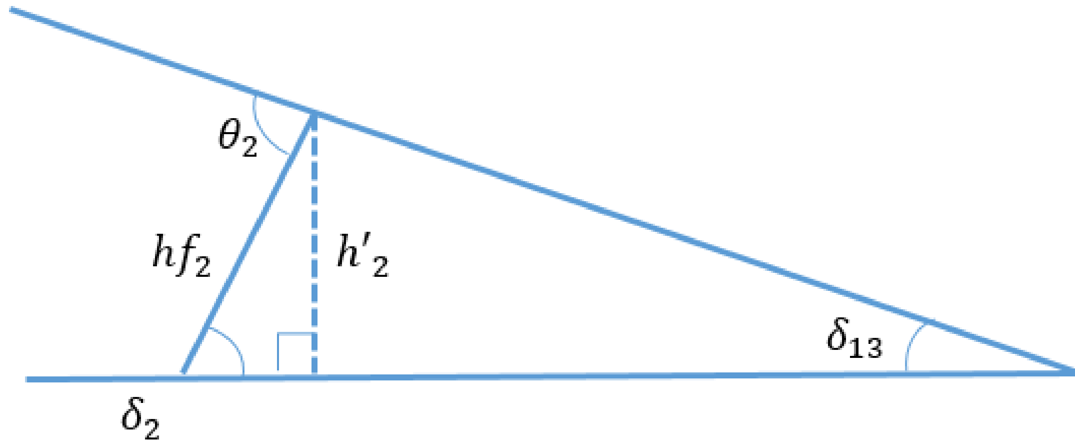

Figure 5.11: Angle of the robot body with the ground in a parallel plane as previously defined but containing the remaining leg in the supporting tripod

Figure 5.11 also defines the triangles which are used to calculate the height of the hip of leg 2 in the defined plane.

$$\delta_2 = 180 - \delta_{13} - (180 - \theta_2) \tag{5.16}$$

$$h'_2 = hf_2 \sin(\delta_2) \tag{5.17}$$

Based on similar calculations the height of the legs in the unsupported tripod in the defined plane are

$$\delta_{iu} = 180 - \delta_{13} - (180 - \theta_{iu}) \tag{5.18}$$

$$h'_{iu} = hf_{iu} \sin(\delta_{iu}) \tag{5.19}$$

where the subscript iu denotes leg i in the unsupported tripod.

Knowing that the robot is symmetrical, the point half way between the hips of legs 1

and 3 is in line with the hip of leg 2 and the center of gravity. Figure 5.12 describes the world systems yz-plane which will be used to calculate the angle between the robot body and the ground and the true height of each hip.

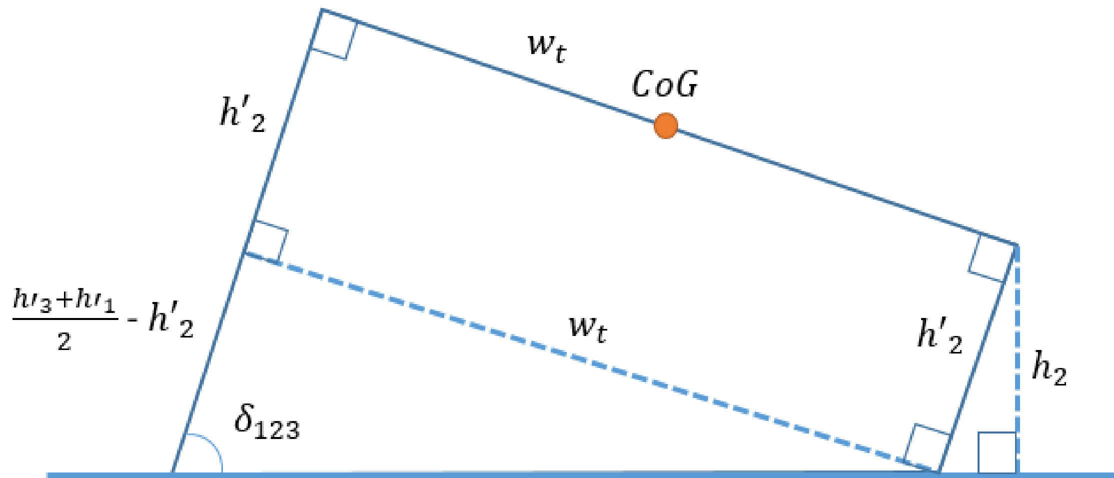

Figure 5.12: Cross section of the robot in the yz-plane including the line whose height is h'_2

Figure 5.12 shows the height of the midpoint between legs 1 and 3, and the width of a tripod, w_t. Using these values, the true height of each of the hips can be calculated using the following equations:

$$\delta_{123} = \arctan\left(\frac{w_t}{\frac{h'_3+h'_1}{2} - h'_2}\right) \tag{5.20}$$

$$h_1 = h'_1 \sin(\delta_{123}) \tag{5.21}$$

$$h_2 = h'_2 \sin(\delta_{123}) \tag{5.22}$$

$$h_3 = h_3' \sin(\delta_{123}) \tag{5.23}$$

Again, knowing that the robot is symmetrical, the hips of the supported and unsupported legs at similar positions in their respective tripods are in line with one another. This fact can be used in order to make the final determination of whether an unsupported leg is dragging. Figure 5.13 shows the relationship between the supported and unsupported legs in the world systems yz-plane.

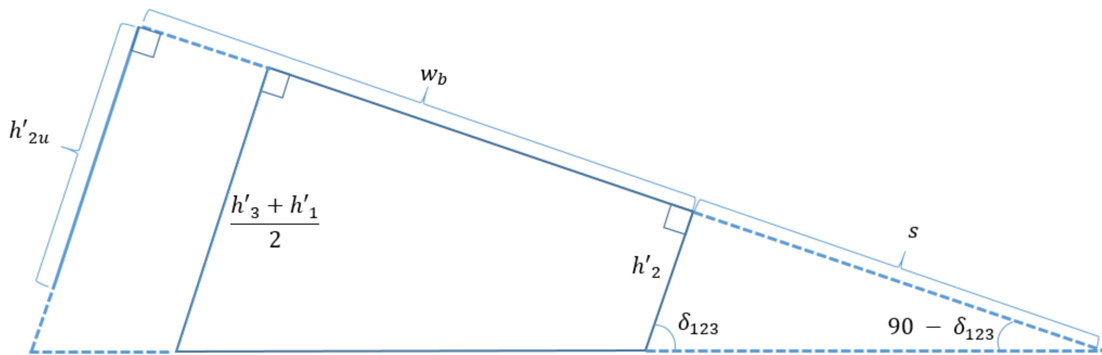

Figure 5.13: Cross section of the robot in the yz-plane including the supported and unsupported legs

Using Figure 5.13, the distance between the hip of leg 2 and the ground in the parallel plane of the robot body is

$$s = \frac{h_2'}{\tan(90 - \delta_{123})} \tag{5.24}$$

Using s, the maximum for the height of the unsupported leg in the previously defined plane is

$$\max(h_{2u}') = (w_b + s)\tan(90 - \delta_{123}) \tag{5.25}$$

Similarly, the maximum for the height of unsupported legs 1 and 3 in the previously

defined plane are

$$s_i = \frac{h'_i}{\tan(90 - \delta_{123})} \tag{5.26}$$

Using s_i, the maximum for the height of the unsupported leg in the previously defined plane is

$$\max(h'_{iu}) = (s_i - w_t - (w_b - w_t)) \tan(90 - \delta_{123}) \tag{5.27}$$

If the height of all unsupported legs in the previously defined plane are less than $\max(h'_{iu})$, for their respective legs, then the unsupported legs are not dragging.

Dragging can occur when the supported tripod is in the middle of the propel phases and the unsupported tripod is between the lifting and descending phases. If a dragging leg exists, then the dragging objective score for use in the genetic algorithms fitness function is zero, otherwise, it is one.

5.2.8 Stability objective score

The stability objective, St, is a binary function that compares the center of gravity projection to the edges of the statically stable region of the robot. The statically stable region of the robot is defined to be the triangle generated by the end effectors of each of the legs in contact with the ground. The closer the projection of the center of gravity is to the edge of the stable region, the more likely external forces are to causing the robot to tip over.

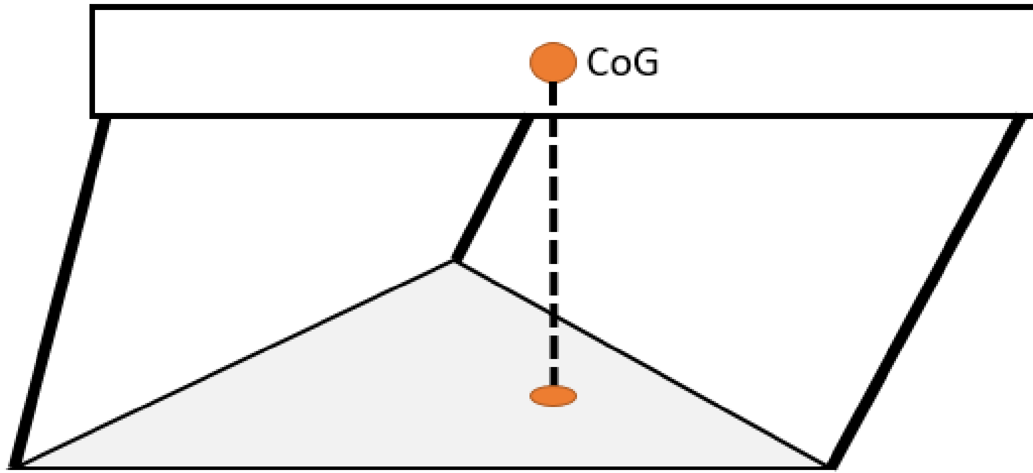

Figure 5.14: Projection of the Center of Gravity within the statically stable region

Using the calculations from the dragging objective score section, it is possible to define the (x, y) coordinates for each of the end effectors and the center of gravity in the world system's reference frame. In order to determine these coordinates, the projection in the z-direction of the hip on leg 1 will be used as the origin in the world system's reference frame. The position of the projection in the z-direction of each hip and the center of gravity are defined as follows

$$Leg1 : x_1 = 0,\, y_1 = 0 \tag{5.28}$$

$$Leg2 : x_2 = \frac{l_b}{2},\, y_2 = w_t \tag{5.29}$$

$$Leg3 : x_3 = l_b,\, y_3 = 0 \tag{5.30}$$

$$CoG : x_{CoG} = \frac{l_b}{2},\, y_{CoG} = w_t - \frac{w_b}{2} \tag{5.31}$$

By applying the offsets due to the angle of the robot body in the world system's reference frame, the (x, y) coordinates for the end effectors and the projection of the center of gravity are defined as follows

$$Leg1 : x_1 = hf_1 \cos(180 - \delta_1 - \delta_{1'}),\ y_1 = -h'_1 \cos(\delta_{123}) \tag{5.32}$$

$$Leg2 : x_2 = \frac{l_b}{2} \cos(\delta_{13}) + hf_2 \cos(\delta_2),\ y_2 = \frac{w_t}{\sin(\delta_{123})} - h'_2 \cos(\delta_{123}) \tag{5.33}$$

$$Leg3 : x_3 = l_b \cos(\delta_{13}) + hf_3 \cos(\delta_3),\ y_3 = -h'_3 \cos(\delta_{123}) \tag{5.34}$$

$$CoG : x_{CoG} = \frac{l_b}{2} \cos(\delta_{13}),\ y_{CoG} = \frac{w_t - \frac{w_b}{2}}{\sin(\delta_{123})} \tag{5.35}$$

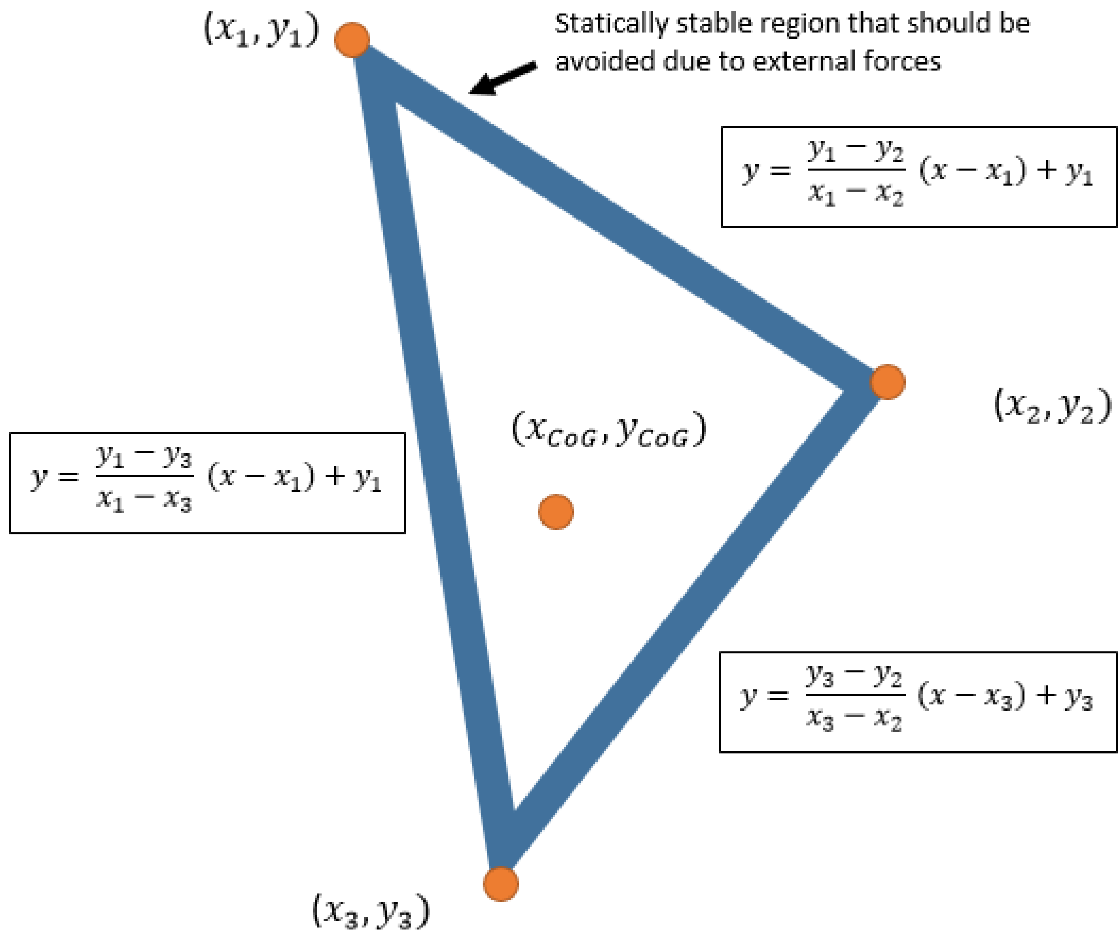

Figure 5.15: Statically stable region and projection of the center of gravity from the top profile

Let the end effectors of each leg in the supporting tripod be defined as (x_1, y_1), (x_2, y_2) and (x_3, y_3) for legs 1, 2 and 3, respectively. By using the point-slope formula, the equation for the line between the end effectors on legs 1 and 2 is

$$y = \frac{y_1 - y_2}{x_1 - x_2}(x - x_1) + y_1 \tag{5.36}$$

Similarly, the equation for the line between legs 1 and 3 is

$$y = \frac{y_1 - y_3}{x_1 - x_3}(x - x_1) + y_1 \tag{5.37}$$

Finally, the equation for the line between legs 2 and 3 is

$$y = \frac{y_3 - y_2}{x_3 - x_2}(x - x_3) + y_3 \tag{5.38}$$

Due to unforeseen external forces, it is important to avoid having the center of gravity of the robot near the edge of the statically stable region. A buffer of size c will be used to ensure that the robot remains upright while walking. c is a value that should be set based on environmental inputs such as wind or other obstacles which could collide with the robot and throw it off balance.

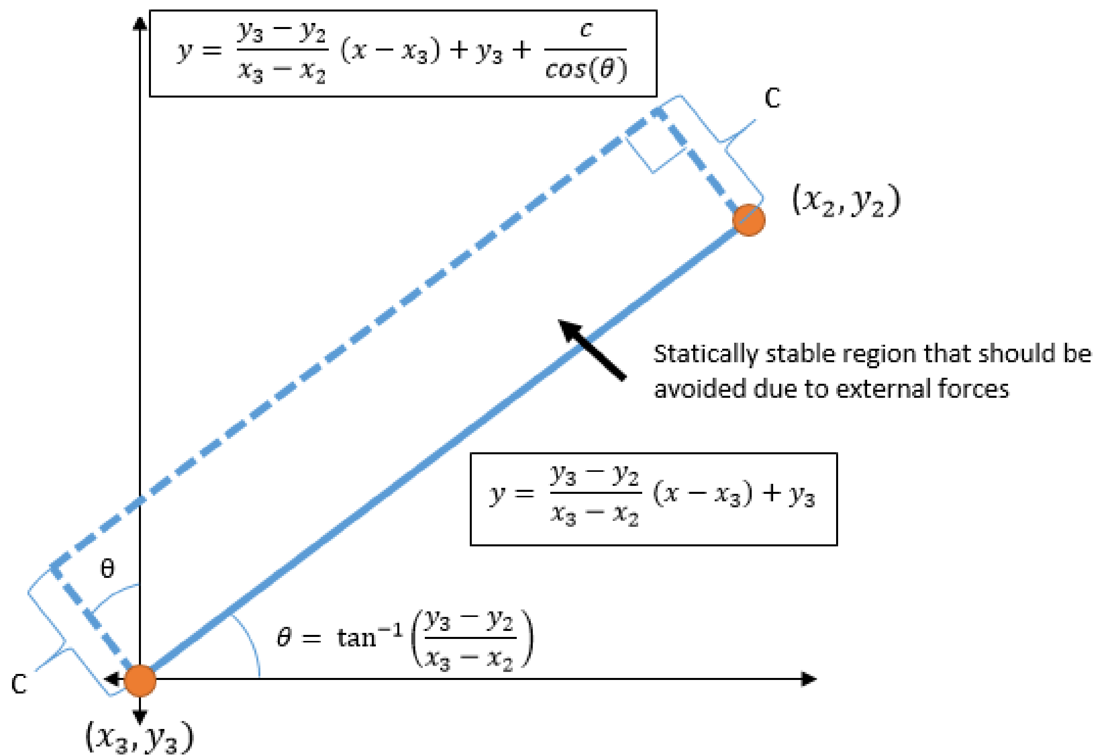

Figure 5.16: Geometry for the line defining the upper limit of the statically stable buffer zone between the end effectors of legs 2 and 3

To define the limits of the buffer region a parallel line is drawn a distance of c from the line between the end effectors. By using the slope of the original line, the y-intercept of the buffer line can be calculated. Using the slope and y-intercept the equation of the buffer line between the end effectors of legs 1 and 2 is

$$y = \frac{y_1 - y_2}{x_1 - x_2}(x - x_1) + y_1 - \frac{c}{\cos\left(\arctan\left(\frac{y_1 - y_2}{x_1 - x_2}\right)\right)} \tag{5.39}$$

Similarly, the buffer line between the end effectors of legs 1 and 3 can be defined as

$$y = \frac{y_1 - y_3}{x_1 - x_3}(x - x_1) + y_1 + \frac{c}{\cos\left(\arctan\left(\frac{y_1 - y_3}{x_1 - x_3}\right)\right)} \tag{5.40}$$

Finally, the buffer line between the end effectors of legs 2 and 3 can be defined as

$$y = \frac{y_3 - y_2}{x_3 - x_2}(x - x_3) + y_3 + \frac{c}{\cos\left(\arctan\left(\frac{y_3 - y_2}{x_3 - x_2}\right)\right)} \tag{5.41}$$

The three buffer equations are used as constraining lines for the projection of the center of gravity to define the stability of the robot. Therefore, if the projection of the center of gravity of the robot in the world systems reference frame satisfies the following equations, while in any supporting position, the stability objective score for use in the genetic algorithm's fitness function is one, otherwise, it is zero.

$$y < \frac{y_1 - y_2}{x_1 - x_2}(x - x_1) + y_1 - \frac{c}{\cos\left(\arctan\left(\frac{y_1 - y_2}{x_1 - x_2}\right)\right)} \tag{5.42}$$

$$y > \frac{y_1 - y_3}{x_1 - x_3}(x - x_1) + y_1 + \frac{c}{\cos\left(\arctan\left(\frac{y_1 - y_3}{x_1 - x_3}\right)\right)} \tag{5.43}$$

$$y > \frac{y_3 - y_2}{x_3 - x_2}(x - x_3) + y_3 + \frac{c}{\cos\left(\arctan\left(\frac{y_3 - y_2}{x_3 - x_2}\right)\right)} \tag{5.44}$$

CHAPTER 6

EXPERIMENTATION

6.1 Experimental Setup

The hexapod that will be used for the experimentation was influenced by [17]. This work describes the configurations of many robots in academia and industry, and their purposes. Based on this research, the representative hexapod being used in this experiment has dimensions as shown in Figure 6.1.

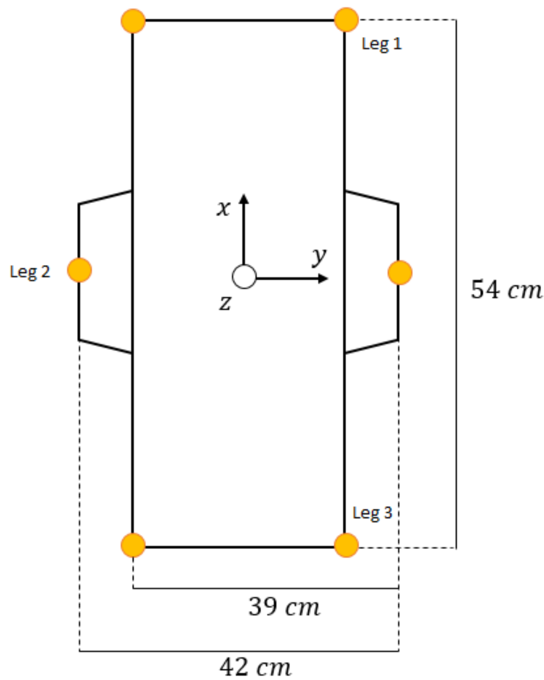

Figure 6.1: Dimensions of the experimental hexapod body

The mass of the robot body is 10 kg. The mass of each leg segment is 1 kg, which includes thet motors and leg segment. The leg segment lengths, l_0 and l_1, shown in Figure 3.5, will be constrained between 6 and 12 cm. The rotational joint angle at the hip, q_0, will be constrained between 0, straight forward in the x-direction, and 135 degrees clockwise, from the aspect of looking at the robot system's xz-plane from the positive y

side. The rotational joint angle at the knee, q_1, will be constrained between 0, straight out from the upper leg, and 135 degrees clockwise, from the aspect of looking at the robot system's xz-plane from the positive y side as shown in Figure 6.2.

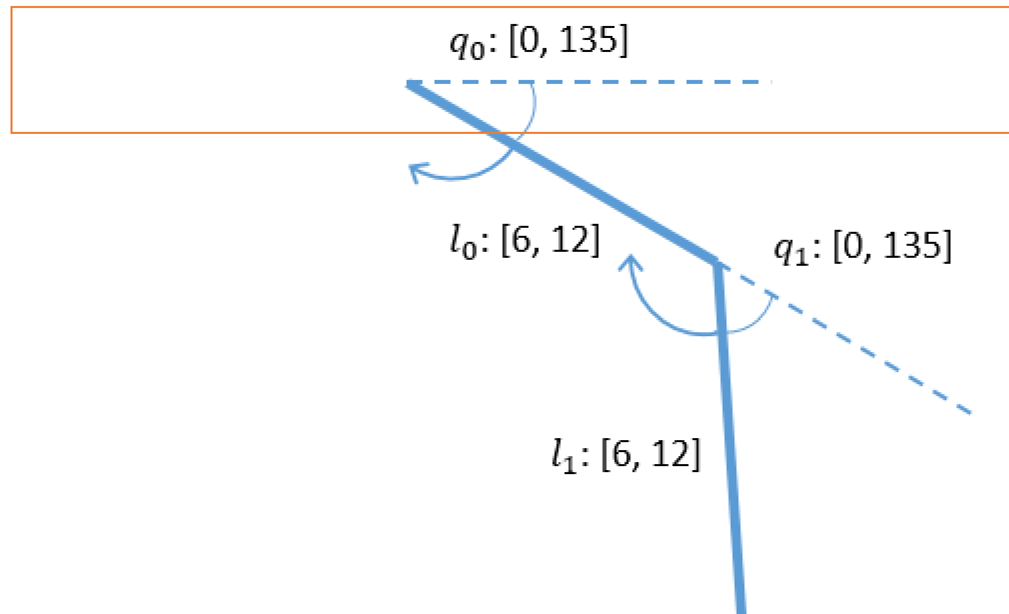

Figure 6.2: Constraints of the experimental hexapod legs

Based on these simulated physical constraints, a GA was defined in order to determine the most efficient, stable gait. Along with the physical constraints of the robot, a number of different factors were also treated as inputs: number of generations, number of parents kept between generations, mutation rate, and weights of the fitness function. To determine the best combination of these inputs, a number of experiments were conducted. Each experiment was conducted by varying the value of a single input at a time holding all others constant. The control value for each input is listed in Table 6.1.

Input parameter	Control value
Number of generations	200
Number of parents kept between generations	20
Mutation rate	0.2
Fitness function weights	0.33, 0.33, 0.33

Table 6.1: Control values for each experiment

6.2 Experimental Results

The initial sets of results were performed to get a baseline of how each input variable effects the overall objective. Each variable was tested over a range of values and each value was tested with ten different random seeds. The individual results of each test are available in Appendix A.

6.2.1 Number of parents kept

The number of parents kept when creating the next generation was varied between 5 and 50, incrementing by 5. After running all of the experiments on this variable the following summary was generated.

	Fitness Score	
Number of Parents Kept	Mean	Standard Deviation
5	0.674228	0.000318
10	0.675601	0.000832
15	0.675565	0.001153
20	0.675397	0.000591
25	0.676746	0.001893
30	0.675871	0.000507
35	0.676442	0.000689
40	0.67609	0.00075
45	0.675972	0.000586
50	0.676089	0.000703

Table 6.2: Results of parent retention experiment

Table 6.2 shows the mean and standard deviation of the fitness score for each set of experiments based on the number of parents kept between generations. Based on the

summary of the results, it can be concluded that no discernible pattern exists. Therefore, in future tests the default value of twenty will be used.

6.2.2 Number of generations

The number of generations was varied between 100 and 1000, incrementing by 100. After running all of the experiments on this variable, the following summary was generated.

	Fitness Score		
Number of Generations	Mean	Standard Deviation	Difference
100	0.674715	0.000611	
200	0.675397	0.000591	0.000683
300	0.676454	0.000647	0.001056
400	0.677302	0.000772	0.000848
500	0.677781	0.000782	0.00048
600	0.678693	0.000943	0.000912
700	0.679294	0.001056	0.000601
800	0.679766	0.001103	0.000472
900	0.680298	0.001631	0.000531
1000	0.680454	0.001518	0.000156

Table 6.3: Results of number of generations experiment

Table 6.3 shows the mean and standard deviation of the fitness score for each set of experiments based on the number of generations. It is clear to see that the mean score is a monotonically increasing function when compared to the number of generations. However, when comparing how much of an effect the next 100 generations has on the score, a diminishing return is present after 600 generations.

6.2.3 Mutation rate

The mutation rate was initially varied between 0.05 and 0.25 incrementing by 0.05. However, after seeing the results of the first few increments, it was clear that the overall score was decreasing and the overall runtime was increasing. Based on these observations, the initial experiment was thrown away and a second experiment was conducted. The

second experiment varied the mutation rate between 0.01 and 0.09 incrementing by 0.02. After running all of the experiments on this variable, the following summary was generated.

	Fitness Score	
Mutation Rate	Mean	Standard Deviation
0.01	0.734306	0.040056
0.03	0.739878	0.025592
0.05	0.708817	0.012818
0.07	0.692769	0.01313
0.09	0.684401	0.004264

Table 6.4: Results of mutation rate experiment

Table 6.5 shows the mean and standard deviation of the fitness score for each set of experiments based on the mutation rate. Based on the summary provided it was concluded that the best overall mutation rate was consistently centered on 0.03.

6.2.4 Weight of efficiency

The weight of the efficiency score was varied initially between 0.33 and 0.73, incrementing by 0.1. After running this experiment, the following summary was generated based on the overall objective score.

	Fitness Score	
Efficiency Weight	Mean	Standard Deviation
0.33	0.675397	0.000591
0.43	0.577982	0.001541
0.53	0.47901	0.001239
0.63	0.381071	0.001393
0.73	0.282983	0.001469

Table 6.5: Results of mutation rate experiment based on overall objective score

It is clear to see that the mean of the overall score decreases as the weight of the efficiency score increases. After realizing the initial implications the efficiency for each experiment was compared, the summary is provided in the following table.

	Efficiency Score	
Efficiency Weight	Mean	Standard Deviation
0.33	0.005397	0.000591
0.43	0.007982	0.001541
0.53	0.00901	0.001239
0.63	0.011071	0.001393
0.73	0.012983	0.001469

Table 6.6: Results of mutation rate experiment based on efficiency

Based on the summary of the efficiency score, it is clear to see that as the efficiency weight increases, the mean of the efficiency score increases. Using this information, a second experiment was conducted varying the weight of the efficiency score between 0.99 and 0.999, incrementing by 0.001, with the addition of data points for 0.95 and 0.97. After running this experiment, the following summary was generated.

	Fitness Score	
Efficiency Weight	Mean	Standard Deviation
0.95	0.066962	0.001273
0.97	0.04942	0.002351
0.99	0.02968	0.003295
0.991	0.030285	0.003719
0.992	0.029906	0.004066
0.993	0.029055	0.001517
0.994	0.032918	0.007089
0.995	0.028146	0.005328
0.996	0.027259	0.004273
0.997	0.026799	0.005292
0.998	0.031505	0.004042
0.999	0.034371	0.008582

Table 6.7: Results of high mutation rate experiment based on overall objective score

Based on the results of using extremely high efficiency weights, there is no discernible pattern to the overall score. After noticing this lack of pattern, the overall score was broken up to compare the efficiency score and the sum of the dragging and stability scores separately.

	Efficiency Score	
Efficiency Weight	Mean	Standard Deviation
0.95	0.017855	0.00134
0.97	0.02002	0.002424
0.99	0.019879	0.003328
0.991	0.022387	0.004082
0.992	0.022082	0.004098
0.993	0.022916	0.001958
0.994	0.027383	0.007265
0.995	0.024016	0.006263
0.996	0.024156	0.004315
0.997	0.025074	0.005572
0.998	0.030767	0.004265
0.999	0.034055	0.00865

Table 6.8: Results of high mutation rate experiment based on efficiency

Using the results from Table 6.8 the efficiency scores show an increasing trend as the efficiency weight increases. So, comparing a summary of the sum of the dragging and stability scores, the randomness of the overall score becomes evident.

Efficiency Weight	Number of Trials without Dragging and Stability
0.95	0
0.97	0
0.99	0
0.991	2
0.992	0
0.993	2
0.994	1
0.995	3
0.996	4
0.997	7
0.998	10
0.999	9

Table 6.9: Results of high mutation rate experiment based on the number of trials without dragging and stability

Based on this summary, the randomness in the overall scores comes from the fact that when the weight of the efficiency score is extremely high the importance of the dragging

and stability objectives is very low. This de-prioritization leads to lower overall scores and infeasible gaits. Therefore, in order to ensure feasibility while maintaining high efficiency, the weight of the efficiency score will be set to 0.97 for future testing.

6.2.5 Best settings

Based on the results of the preceeding sections, two final experiment were conducted. The first experiment combined the best controlled inputs to optimize the gait of the hexapod. The second experiment combined the best controlled inputs but increased the mutation rate to determine if the GA was stuck in a local maximum.

After running the first experiment the following gait was produced with an overall objective score of 1.1.

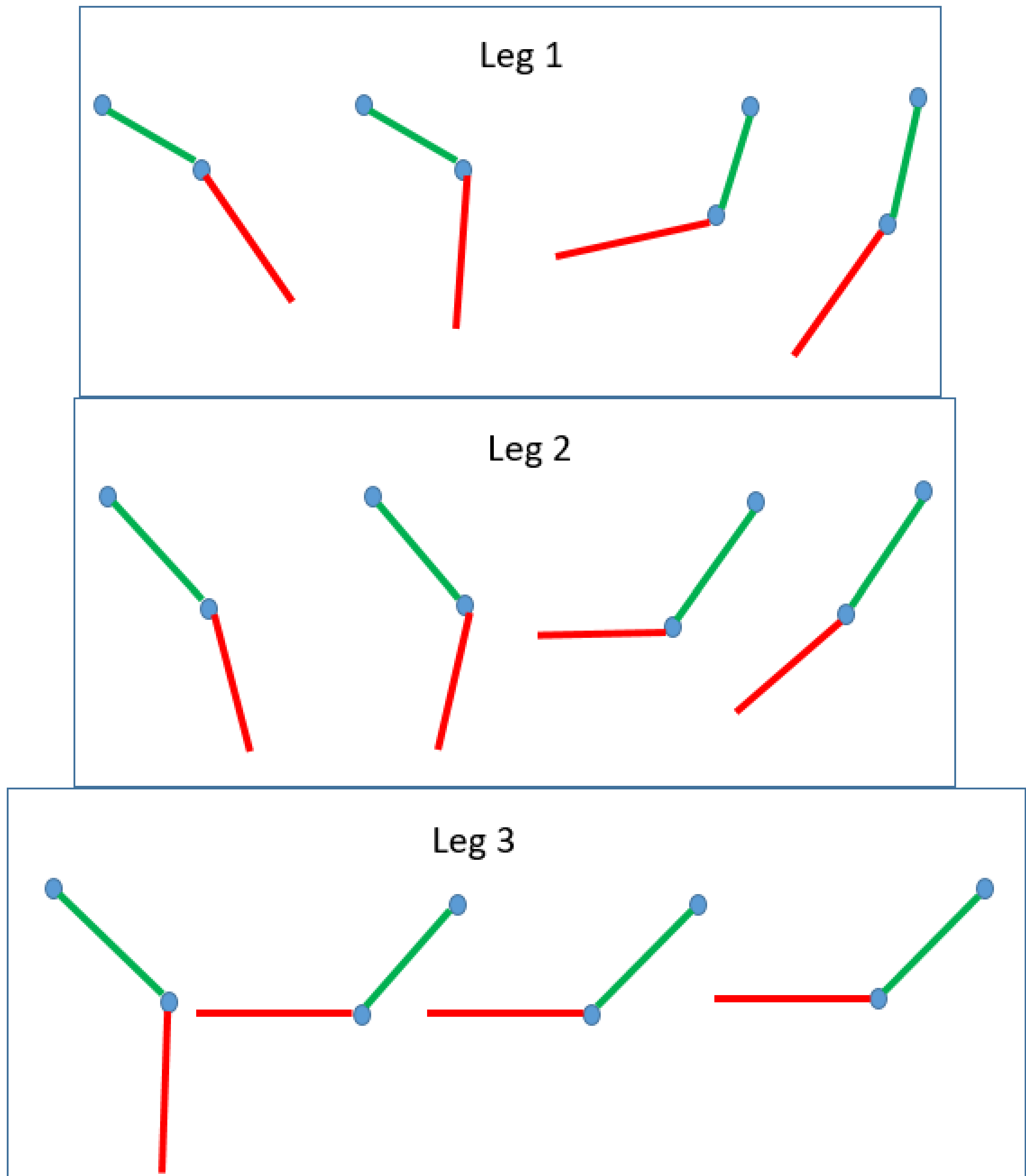

Figure 6.3: Gait for each leg based on the best settings from the controlled experiments

Leg	Phase	q_0	q_1	l_0	l_1
1	1	34.2	20.2	8.2	10.8
1	2	34.5	56.6	8.3	10.8
1	3	105.2	56.6	8.3	10.8
1	4	104.5	20.2	8.3	10.8
2	1	48.1	18.6	9.9	10
2	2	48.2	53.7	9.9	10
2	3	121.9	53.6	9.9	10
2	4	120.8	18.4	9.9	9.9
3	1	45.8	45.1	10.1	11.3
3	2	133.8	45	10.1	11.4
3	3	134.7	43.3	10.1	11.6
3	4	134.1	45.1	10.1	11.8

Table 6.10: Raw results of the gait based on the best settings from the controlled experiments

Based on the raw results in Table 6.10 the leg segment lengths of both the upper and lower segments change very little. These results suggest that based on the setup provided that adding the capability of changing the leg segment length provides little to no additional efficiency to the overall gait.

After running the second experiment, increasing the mutation rate, it was seen that the overall fitness score was reduced to 0.2 over the same number of generations. This result suggests that the increased mutation rate caused the GA to steer away from potentially good scores in order to vary the leg configurations more often.

CHAPTER 7

CONCLUSION

Based on the results of the experiments conducted, using the defined genetic algorithm and constraints did not yield a leg-length configuration such that the energy consumption per unit distance over a flat terrain will decrease with dynamic leg-lengths. A few observations can be made that may account for some of the observed results. As such, exploring resolutions to some of the possible problems and modifying the approach could achieve a gait that would decrease the energy consumption per unit distance over a flat terrain.

First, comparing the leg segment lengths between each phase shows that no change in leg length is used to develop the most efficient gait. This observation suggests that the power consumption of the prismatic motors is too great to make up for the additional distance that could be achieved by extending the leg segments. Looking back at the energy and intuitive dynamics sections, the likely culprit to the high energy consumption in the prismatic motors is the high revolutions per minute (RPM) value. The high RPM value is derived from the number of rotations required to increase or decrease the segment length. The current value of the lead between threads in the worm gear is 6 mm and, with the requirement of every movement taking one second, the RPM values can go as high as 600. Changing the pitch of the threads in the worm gear could significantly impact the RPM value and, therefore, the prismatic motor power requirements.

The second observation is that even though all leg segments remain relatively static, each leg has different segment lengths. Even though this is a valid configuration, when developing a robot for traversing flat terrain it is much more likely that each leg would have similar leg lengths. The reason for having similar leg lengths is to ensure that the robot remains stable as it walks. When the leg lengths are the same, it is more likely that

the center of gravity will remain near the center of the statically stable region, making the robot as stable as possible. Because the stability objective score is binary, there is no incentive for the center of gravity to remain near the center of the statically stable region. To incentivize stability, the stability objective score would need to be updated to give a weighted priority based on distance from the center of the statically stable region.

The third observation is that leg 3 holds its second phase pose for three phases. Based on the constraints put on the GA this is a perfectly feasible gait. Independent of other legs, the stride produced by this leg may be a good solution; however, when comparing the phases of this leg to legs 1 or 2, it is easy to see that similar phases have very different joint spaces. Having different joint spaces doesnt necessarily mean that a gait is infeasible. However, the position of the end effectors do need to be fairly synchronized during supporting phases, otherwise the end effectors will slip across the ground and the statically stable region will be deformed.

When comparing the results here to results from other papers that optimize the hexapod gait based on energy consumption or efficiency, one big difference stands out. The difference is that most of the other papers optimize based on the foothold position; this research optimizes the joint space. For example, [10] uses oscillator functions to define the shape of the foothold position over time. The oscillator functions coefficients are varied in order to determine the optimal gait. The issue with this approach is that after the foothold is defined, inverse kinematics are used to determine the joint space, which is then used to calculate the power consumed for the gait. This is an issue because the calculations used in inverse kinematics produce one of many possible joint spaces that could achieve the same foothold and the one it chooses may not be the optimal, overall joint space. By optimizing the joint space directly, this research avoids this pitfall.

One benefit to the approach taken in [10], as opposed to the approach in this research, was solve time. Based on the limited number of coefficients in the oscillator functions and

the inverse kinematics deterministic nature, the number of joint spaces produced would be levels of magnitude less than the GA search in this research. So, if solve time is a consideration then forgoing some amount of optimality may be required.

7.1 Future Work

Based on the conclusion, modifying the GA to include suggested improvements is the first thrust for future work. The first suggestion from the previous section was to find a way to decrease the prismatic motor power requirement. To do this, the pitch of the threads in the worm gear could be increased to require less revolutions to perform a single prismatic translation. The pitch is; however, constrained by the desire to ensure that the robot weight does not backdrive the motor. Else, the motor would have to remain powered or a locking device would be required for each prismatic motor. Other suggested approaches include looking into different types of linear actuation, such as, pneumatic or hydraulic motors that may consume less power to achieve the desired effect.

The second suggestion from the previous section was to synchronize the leg footholds of the individual leg in a tripod. To account for this behavior two approaches are suggested: (1) controlling part of the joint space and (2) adding a foothold relative position objective to the overall objective function. Controlling part of the joint space, for example, could be achieved by increasing the size of the chromosome to eight alleles, four phases to propel and four phases to lift and descend. The first three phases of each group could contain fixed hip joint positions at 60, 90 and 120 degrees for the propel phase and the opposite for the lift and descend phase. The final phase in each group would allow the hip to extend further in the appropriate direction if it is optimal to do so.

The secondary approach could be applied by using the foothold positions calculated for the stability objective across all supporting phases. If the relative footholds positions dont change, or changes within some tolerance, then the objective would submit a score of one to the overall objective function.

The final consideration mentioned in the conclusion was for solve time. In order to reduce runtime it may be necessary to forgo some amount of optimality, which may have been reached during a GA search, to solve in some fixed time. The first and simplest approach is to set the runtime as one of the GA exit criteria. Depending on how many generations have passed, this may be a reasonable approach; however, without the time to iterate over many generations the optimality gap for this approach could be quite high.

Another approach to consider is a similar approach to [10]. First, a set of oscillator functions would be constructed based on the leg configurations defined. Second, two sets of inverse kinematic equations would need to be produced to translate the footholds in the oscillator function to the joint space, one with static leg segment lengths and one with dynamic leg segment lengths. When calculating the fitness of a given solution both joint spaces would need to be tested to determine which joint space would yield better fitness.

Another factor that was not considered during this research was distance per unit time. It is important to consider the time to complete a task when performing that task. A robot may have an extremely efficient gait but the time it would take to walk a given distance might be beyond the acceptable bounds of the problem. To consider time, a new objective, based on distance per unit time, would be added to the overall fitness function.

The natural extension of this work is to consider dynamic leg length while walking across an inclined plane. The intuition is that allowing the legs at lower elevation to be longer than the legs at higher elevations, (1) the robot will remain statically stable at greater inclines and (2) the statically stable region becomes larger allowing for longer strides.

In addition to walking across an incline, walking on uneven surfaces is another future research area. The hypothesis is that the ability to modify leg-length would allow the robot to maintain improved static stability with minimal modification to the robots footprint when compared to legged robotic vehicles that must tuck knees up or out. The

difficulty with the latter being that (1) varying tuck poses may cause the legs to collide or make walking difficult, and (2) a lack of sensors on the leg segments may cause them to collide with terrain features in jagged terrain.

Another research avenue could change the overall objective. One idea is that some robotics applications may require a robot to carry an object that needs to remain as level as possible. The same approach taken in this research can be applied by updating the objective function to include a measure of level. The intuition here is that when walking on uneven surfaces extending legs whose footholds are lower will allow the robot to remain level while still allowing for full stride length.

REFERENCES

[1] D. Sanz-Merodio. Analyzing energy-efficient configurations in hexapod robots for demining applications. *Ind. Robot Int. J.*, 39(4):357364.

[2] P. Lin. A leg configuration measurement system for full body pose estimates in a hexapod robot. *IEEE Trans. Robot*, 21(3):411422.

[3] M. Li. Free gait generation based on discretization for a hexapod robot. *IEEE ROBIO*.

[4] L. Xu, W. Liu, Z. Wang, and W. Xu. Gait planning method of a hexapod robot based on the central pattern generators: Simulation and experiment. In *2013 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pages 698–703, Dec 2013.

[5] Y.-G. Zhu. Optimal design of hexapod walking robot leg structure based on energy consumption and workspace. *Transactions of the Canadian Society for Mechanical Engineering*, 38:305317.

[6] MicroMo. Dc motor calculations. `https://www.micromo.com/technical-library/dc-motor-tutorials/motor-calculations`, 2017. [Online; accessed 9-November-2017].

[7] T. DeWolf. Jacobian, velocity and force. `https://studywolf.wordpress.com/2013/09/02/robot-control-jacobians-velocity-and-force/`, 2013. [Online; accessed 6-November-2017].

[8] Susan E. Carlson. A general method for handling constraints in genetic algorithms. 03 1996.

[9] R. Kowalczyk. Using constraint satisfaction in genetic algorithms. In *1996 Australian New Zealand Conference on Intelligent Information Systems. Proceedings. ANZIIS 96*, pages 272–275, Nov 1996.

[10] Dariusz Grzelczyk, Bartosz Stanczyk, and Jan Awrejcewicz. Power consumption analysis of different hexapod robot gaits, 12 2015.

[11] P. Gonzalez de Santos. Minimizing energy consumption in hexapod robots. *Adv. Robot*, 23:681704.

[12] Daoxiong Gong. A review of gait optimization based on evolutionary computation. *Applied Computational Intelligence and Soft Computing*, 2010.

[13] Shibendu Shekhar Roy and Dilip Pratihar. Dynamic modeling, stability and energy consumption analysis of a realistic six-legged walking robot. 29:400416, 04 2013.

[14] Jun Nishii. An analytical estimation of the energy cost for legged locomotion. *Journal of Theoretical Biology*, 238(3):636 – 645, 2006.

[15] M. R. Heinen and F. S. Osrio. Morphology and gait control evolution of legged robots. In *2008 IEEE Latin American Robotic Symposium*, pages 111–116, Oct 2008.

[16] A. Manglik, K. Gupta, and S. Bhanot. Adaptive gait generation for hexapod robot using genetic algorithm. In *2016 IEEE 1st International Conference on Power Electronics, Intelligent Control and Energy Systems (ICPEICES)*, pages 1–6, July 2016.

[17] G. Tedeschi. Design issues for hexapod walking robots. *Robotics 3*, 2:181–206.

APPENDIX A

RAW RESULTS

	Fitness Score									
Number of Parents Kept	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10
5	0.674556	0.67397	0.6747	0.673894	0.674359	0.673807	0.673931	0.674151	0.674404	0.674513
10	0.675494	0.675821	0.674989	0.675093	0.675749	0.676451	0.676213	0.674767	0.677078	0.674359
15	0.675339	0.676464	0.674302	0.67514	0.675363	0.674852	0.675498	0.674907	0.678445	0.675341
20	0.675166	0.674282	0.674827	0.67534	0.676133	0.675848	0.675883	0.676074	0.675263	0.675158
25	0.676472	0.681994	0.675773	0.675611	0.676068	0.677026	0.676551	0.675756	0.676	0.676208
30	0.675559	0.675435	0.67609	0.67572	0.675716	0.675166	0.675664	0.676023	0.676887	0.676452
35	0.676448	0.676432	0.677241	0.676591	0.675904	0.677469	0.675963	0.676112	0.677064	0.6752
40	0.675714	0.677428	0.675925	0.675749	0.675103	0.6768	0.67672	0.675407	0.676591	0.675461
45	0.675334	0.675828	0.675937	0.675737	0.676037	0.676076	0.677387	0.676351	0.675391	0.675644
50	0.675922	0.676773	0.676076	0.676189	0.675462	0.675904	0.675416	0.67771	0.675435	0.676001

Table A.1: Fitness scores of parent retention experiment

	Fitness Score									
Number of Generations	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10
100	0.674746	0.674193	0.67458	0.673941	0.676133	0.675021	0.674732	0.674444	0.675056	0.674299
200	0.675166	0.674282	0.674827	0.67534	0.676133	0.675848	0.675883	0.676074	0.675263	0.675158
300	0.676261	0.676063	0.675738	0.676512	0.676453	0.675927	0.67801	0.67614	0.676972	0.676458
400	0.677305	0.67689	0.678019	0.676729	0.678617	0.677758	0.67801	0.676259	0.676972	0.676458
500	0.678498	0.677485	0.678019	0.676729	0.678617	0.67884	0.67801	0.677455	0.677701	0.676458
600	0.679349	0.677485	0.678282	0.680117	0.678617	0.679983	0.678322	0.67768	0.679268	0.677828
700	0.679759	0.679866	0.678282	0.680117	0.678617	0.679983	0.678322	0.678669	0.681286	0.678041
800	0.679759	0.679866	0.68098	0.680117	0.678617	0.680358	0.678322	0.680319	0.681286	0.678041
900	0.682144	0.679866	0.68098	0.680117	0.678617	0.682025	0.678322	0.680319	0.682547	0.678041
1000	0.682144	0.679866	0.68098	0.680812	0.678617	0.682025	0.678352	0.680319	0.682547	0.678878

Table A.2: Fitness scores of number of generations experiment

	Fitness Score									
Efficiency Weight	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10
0.33	0.675166	0.674282	0.674827	0.67534	0.676133	0.675848	0.675883	0.676074	0.675263	0.675158
0.43	0.577231	0.579213	0.575846	0.576686	0.579675	0.57881	0.57866	0.575935	0.577613	0.580155
0.53	0.478643	0.47812	0.481995	0.478653	0.478855	0.477853	0.479083	0.478887	0.480116	0.4779
0.63	0.382454	0.380207	0.379285	0.381036	0.380143	0.379263	0.382282	0.380712	0.382027	0.383303
0.73	0.280948	0.281327	0.285187	0.283604	0.282862	0.285528	0.282627	0.282578	0.28289	0.282279

Table A.3: Results of mutation rate experiment based on overall objective score

	Efficiency Score									
Efficiency Weight	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10
0.33	0.005166	0.004282	0.004827	0.00534	0.006133	0.005848	0.005883	0.006074	0.005263	0.005158
0.43	0.007231	0.009213	0.005846	0.006686	0.009675	0.00881	0.00866	0.005935	0.007613	0.010155
0.53	0.008643	0.00812	0.011995	0.008653	0.008855	0.007853	0.009083	0.008887	0.010116	0.0079
0.63	0.012454	0.010207	0.009285	0.011036	0.010143	0.009263	0.012282	0.010712	0.012027	0.013303
0.73	0.010948	0.011327	0.015187	0.013604	0.012862	0.015528	0.012627	0.012578	0.01289	0.012279

Table A.4: Results of mutation rate experiment based on efficiency

	Fitness Score									
Efficiency Weight	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10
0.95	0.066823	0.068068	0.067719	0.066641	0.065237	0.067471	0.064823	0.067376	0.066408	0.069055
0.97	0.052185	0.05132	0.052177	0.052591	0.046708	0.048508	0.047302	0.048261	0.047626	0.047519
0.99	0.027178	0.030079	0.033983	0.027636	0.025316	0.031139	0.030738	0.027786	0.035731	0.027213
0.991	0.028848	0.029004	0.032982	0.037641	0.025704	0.031343	0.026669	0.031563	0.03283	0.026268
0.992	0.031432	0.02418	0.0309	0.02907	0.02681	0.028163	0.026594	0.029451	0.038259	0.034195
0.993	0.028389	0.029965	0.030247	0.026963	0.031747	0.030268	0.027524	0.029398	0.027616	0.028435
0.994	0.04329	0.026049	0.034411	0.024974	0.033908	0.046478	0.032004	0.027088	0.031119	0.029862
0.995	0.023945	0.023543	0.027892	0.025251	0.025516	0.037379	0.037996	0.029125	0.024177	0.026634
0.996	0.024062	0.022482	0.026383	0.021452	0.031642	0.02648	0.035176	0.027554	0.026429	0.030934
0.997	0.038381	0.025321	0.025211	0.028522	0.031147	0.0214	0.026317	0.019535	0.028095	0.024059
0.998	0.03377	0.036315	0.030888	0.032851	0.029544	0.027903	0.035631	0.03456	0.022982	0.03061
0.999	0.035674	0.030089	0.048379	0.033123	0.028839	0.049448	0.021807	0.029529	0.034818	0.032006

Table A.5: Results of high mutation rate experiment based on overall objective score

	Efficiency Score									
Efficiency Weight	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10
0.95	0.017709	0.019019	0.018652	0.017517	0.016038	0.01839	0.015603	0.01829	0.017271	0.020058
0.97	0.022871	0.02198	0.022862	0.023289	0.017225	0.01908	0.017837	0.018825	0.018171	0.018061
0.99	0.017352	0.020281	0.024225	0.017815	0.01547	0.021353	0.020947	0.017966	0.025991	0.017387
0.991	0.020028	0.020186	0.0242	0.028901	0.016855	0.022546	0.022371	0.022768	0.028587	0.017425
0.992	0.023621	0.016311	0.023084	0.02124	0.018962	0.020326	0.018744	0.021624	0.030503	0.026406
0.993	0.02154	0.023127	0.026936	0.023629	0.024921	0.023432	0.020668	0.022556	0.020761	0.021586
0.994	0.037515	0.02017	0.031601	0.019089	0.028077	0.040722	0.026161	0.021215	0.025271	0.024006
0.995	0.01904	0.018637	0.023007	0.022865	0.020619	0.035055	0.035675	0.024246	0.019273	0.021742
0.996	0.022151	0.018556	0.022473	0.01953	0.027753	0.02257	0.033309	0.023649	0.024527	0.027042
0.997	0.036992	0.022388	0.025287	0.027103	0.029736	0.01996	0.024892	0.016585	0.02517	0.022627
0.998	0.032835	0.035386	0.029948	0.031915	0.028602	0.026957	0.035703	0.034629	0.022026	0.029669
0.999	0.034708	0.029619	0.047926	0.033156	0.028368	0.049497	0.021328	0.029558	0.034853	0.031537

Table A.6: Results of high mutation rate experiment based on efficiency

	Violates Dragging or Stability									
Efficiency Weight	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10
0.95	0	0	0	0	0	0	0	0	0	0
0.97	0	0	0	0	0	0	0	0	0	0
0.99	0	0	0	0	0	0	0	0	0	0
0.991	0	0	0	0	0	0	1	0	1	0
0.992	0	0	0	0	0	0	0	0	0	0
0.993	0	0	1	1	0	0	0	0	0	0
0.994	0	0	1	0	0	0	0	0	0	0
0.995	0	0	0	1	0	1	1	0	0	0
0.996	1	0	0	1	0	0	1	0	1	0
0.997	1	0	1	1	1	1	1	0	0	1
0.998	1	1	1	1	1	1	1	1	1	1
0.999	0	1	1	1	1	1	1	1	1	1

Table A.7: Results of high mutation rate experiment based on the trials without dragging and stability

APPENDIX B

GENETIC ALGORITHM CODE

```
import chromosome.Allele;
import chromosome.Chromosome;
import chromosome.FitnessFunction;
import legVisual.SideView;

import java.io.*;
import java.util.*;

/**
 * Created by rcafarel on 02/25/2017.
 */
public class GeneticAlgorithm {

    private static Random generator;

    private static double MUTATION_PROBABILITY = 0.03;
    private static int KEEP_FROM_EACH_GENERATION = 20;
    private static double efficiencyObjectiveWeight = 0.97;
    private static int CHILDREN_PER_GENERATION = 100;
    private static int GENERATIONS = 600;
    private static List<Chromosome> population =
            new ArrayList<>(CHILDREN_PER_GENERATION);
    private static List<Chromosome> children =
            new ArrayList<>(CHILDREN_PER_GENERATION);
    public static FitnessFunction fitnessFunction;
    public static int randomSeed = 5;

    private static double bestScore = 0;

    public static void main(String[] args) {
        args = new String[3];
        System.out.println(Calendar.getInstance().getTime());
        for (randomSeed=5; randomSeed<=5; randomSeed+=5) {
            for (efficiencyObjectiveWeight=0.97;
                 efficiencyObjectiveWeight<=0.97;
                 efficiencyObjectiveWeight+=0.02) {
                bestScore = 0;
                generator = new Random(randomSeed);
                population.clear();
                children.clear();
                args[0] = String.valueOf(efficiencyObjectiveWeight);
                args[1] = String.valueOf((1 -
                        efficiencyObjectiveWeight) / 2.0);
                args[2] = String.valueOf((1 -
                        efficiencyObjectiveWeight) / 2.0);
                ga(args);
            }
        }
        SideView sideView = new SideView();
        sideView.show(population.get(0));
// show the stride of the best chromosome
    }

    public static void ga(String[] args) {
        if (args.length == 0) {
```

```
            fitnessFunction = new FitnessFunction(0.33, 0.33, 0.33);
        } else if (args.length == 3) {
            fitnessFunction = new FitnessFunction(
                    Double.valueOf(args[0]), Double.valueOf(args[1]),
                       Double.valueOf(args[2]));
        } else if (args.length > 3) {
            fitnessFunction = new FitnessFunction(
                    Double.valueOf(args[0]), Double.valueOf(args[1]),
                        Double.valueOf(args[2]));
            initializePopulation(args[3]);
        }
        randomlyGenerateRemainingPopulation();
        assignCumulativeScores();
        for (int i=0; i<GENERATIONS; i++) {
            if (population.get(0).getScore() > bestScore) {
                bestScore = population.get(0).getScore();
                System.out.println("Generation: " + i);
                population.get(0).outputChromosome();
            }
            performGeneration();
        }
        appendBestScoreToEndOfFile();
        System.out.println("GA complete, best individual score: " +
                 population.get(0).getScore());
    }

    public static void performGeneration() {
        List<Chromosome> bestFromPreviousGeneration =
                 new ArrayList<>();
        children.clear();
        // Keep the best x from each generation
        for(int i=0; i<KEEP_FROM_EACH_GENERATION; i++) {
            bestFromPreviousGeneration.add(population.get(i));
        }

        // Cross-over between parents, performed externally, for the
           //remainder of the members of the next generation.
        while (children.size() <
           CHILDREN_PER_GENERATION-KEEP_FROM_EACH_GENERATION) {
            Chromosome parent1 = selectParent(null);
            Chromosome parent2 = selectParent(parent1);
            Chromosome child = performCrossOver(parent1, parent2);
            if (child != null) {
                // Mutate all children, performed internally
                child.mutate(MUTATION_PROBABILITY);
                if (child.isValidStride()) {
                    children.add(child);
                } else {
                    int a =0;
                }
            }
        }

        // Clear the contents of the population. The next generation
           //will provide all chromosomes for the population.
```

```
        population.clear();

        // add the children to the next generation of the population
        for(Chromosome c: children) {
            population.add(c);
        }
        for (Chromosome c: bestFromPreviousGeneration) {
            population.add(c);
        }

        // At the beginning of every generation sort the population.
        // The sorting criteria is calculated in Chromosome.compareTo
        Collections.sort(population);
        assignCumulativeScores();
    }

    public static void assignCumulativeScores() {
        // This method is called after sorting the population by
           //fitness score.
        double cumulativeRelativeScore = 0;
        double totalScore = 0;

        for (Chromosome c: population) {
            totalScore += c.getScore();
        }
        for (Chromosome c: population) {
            cumulativeRelativeScore += c.getScore()/totalScore;
            c.setCumulativeScore(cumulativeRelativeScore);
        }
    }

    public static Chromosome selectParent(
           Chromosome previouslySelectedParent) {
        // A parent is selected by random draw.
        // The likelihood that a chromosome is selected is based on
           //their relative score
        // with respect to all other chromosomes in the population.
        double randomValue = generator.nextDouble();
        for (Chromosome c: population) {
            if (!c.equals(previouslySelectedParent)) {
                if (c.getCumulativeScore() >= randomValue) {
                    return c;
                }
            }
        }
        return population.get(population.size()-1);
    }


    public static Chromosome performCrossOver(Chromosome p1,
           Chromosome p2) {
        // Select the point at which to cross over then generate the
           //two new children.
        int firstAlleleFirstParent = (int)(generator.nextDouble()*4);
        Chromosome child1 = generateChild(p1, p2,
```

```
            firstAlleleFirstParent);
        if (child1.isValidStride()) {
            return child1;
// currently only generating one child per crossover
        }
        return null;
    }

    public static Chromosome generateChild(Chromosome p1,
            Chromosome p2, int crossOverPoint) {
        // pick two consecutive alleles from the first parent, then the
            //next two from the second parent.
        Allele[] alleles = new Allele[4];
        alleles[crossOverPoint] =
            p1.getAlleleByIndex(crossOverPoint+1);
        alleles[(crossOverPoint+1)%4] =
            p1.getAlleleByIndex((crossOverPoint+1)%4+1);
        alleles[(crossOverPoint+2)%4] =
            p2.getAlleleByIndex((crossOverPoint+2)%4+1);
        alleles[(crossOverPoint+3)%4] =
            p2.getAlleleByIndex((crossOverPoint+3)%4+1);
        return new Chromosome(alleles, p1, p2, generator);
    }

    public static void initializePopulation(String fileName) {
        BufferedReader br = null;
        try {
            br = new BufferedReader(new FileReader(fileName));
            try {
                String line = br.readLine();
                int i = 0, lineNumber = 0;
                Allele[] alleles = new Allele[4];

                while (line != null) {
                    String[] stringValues = line.split(" ");
                    if (stringValues.length == 12) {
                        // assume if there are four values separated by
                       // a single space then we have a good allele,
                        // bad assumption but I will be generating the
                       //initial data
                        alleles[i] = new Allele(stringValues);
                    } else {
                        System.out.println("Malformed allele on line "
                             + lineNumber + " of file " + fileName);
                        break;
                    }
                    if (i==3) {
                       // every four alleles make one chromosome
                        population.add(new Chromosome(alleles, null,
                              null, generator));
                        i = 0;
                    } else {
                        i++;
                    }
                    line = br.readLine();
```

```
                    lineNumber++;
                }
            } catch (IOException ioe) {}
        } catch (FileNotFoundException fnfe) {
        } finally {
            try {
                if (br != null) {
                    br.close();
                }
            } catch (IOException ex) {
                ex.printStackTrace();
            }
        }
        Collections.sort(population);
    }

    public static void appendBestScoreToEndOfFile() {
        try {
            PrintWriter out = new PrintWriter(new BufferedWriter(
                    new FileWriter("C:\\life\\hexapod
\\software\\11-4-17\\ga_homeGrown\\src\\bestScore.txt", true)));
            out.println(efficiencyObjectiveWeight + "," + randomSeed +
                  "," + population.get(0).getScore() + "," +
                    population.get(0).getEfficiencyObjectiveScore());
            out.close();
        } catch (IOException e) {
        }
    }

    public static void randomlyGenerateRemainingPopulation() {
        for (int i=population.size(); i<CHILDREN_PER_GENERATION; i++) {
            population.add(new Chromosome(generator));
        }
    }
}
```

```
package chromosome;

import java.util.Random;

/**
 * Created by rcafarel on 02/25/2017.
 */
public class Chromosome implements Comparable<Chromosome> {

    private static int count = 0;

    public double[] values = new double[48];
    private int id;
    private double score;
    private double efficiencyObjectiveScore;
    private double draggingObjectiveScore;
    private double stabilityObjectiveScore;
    private double cumulativeScore;
    private int numberOfMutations = 0;

    public Random generator;

    private Chromosome parent1, parent2;
// these are just used for debugging

    public Chromosome(Allele[] alleles, Chromosome parent1,
           Chromosome parent2, Random generator) {
        this(alleles);
        updateScore();
        this.parent1 = parent1;
        this.parent2 = parent2;
        this.generator = generator;
        id = count;
        count++;
    }

    public Chromosome(Allele[] alleles) {
        add(alleles[0], 0);  // front propel phase
        add(alleles[1], 12); // back propel phase
        add(alleles[2], 24); // lift phase
        add(alleles[3], 36); // descend phase
    }

    public Chromosome(Random generator) {
        // this constructor is used to initialize the population with
            //random values
        this.generator = generator;
        do {
            for (int i=0; i<12; i+=4)
                randomlyGenerateVerticalShoulder(i);
            for (int i=1; i<48; i+=4)
                values[i] = LegConstraints.randomVK(generator);
            for (int i=2; i<48; i+=4)
                values[i] = LegConstraints.randomUL(generator);
            for (int i=3; i<48; i+=4)
```

```
                values[i] = LegConstraints.randomLL(generator);
            // check to make sure the parents that are randomly
           //generated follow simple constraints
        } while (!isValidStride());
        updateScore();
        id = count;
        count++;
    }

    public void randomlyGenerateVerticalShoulder(int index) {
        values[index] = LegConstraints.randomVH(generator);
        values[index+12] = LegConstraints.randomVH_withMin(generator,
                 values[index]);
        values[index+24] = LegConstraints.randomVH_withMin(generator,
                 values[index+12]);
        values[index+36] = LegConstraints.randomVH_withMax(generator,
                 values[index+24]);
    }

    private void add(Allele a, int startingPoint) {
// used to overwrite any allele in the chromosome
        for (int i=0; i<12; i++)
            values[i+startingPoint] = a.getValues()[i];
    }

    public Allele getAlleleByIndex(int alleleIndex) {
        double[] values = new double[12];
        for (int i=0; i<12; i++) {
            values[i] = this.values[(alleleIndex-1)*12+i];
        }
        return new Allele(values);
    }

    public double[] getLeg_Phase_(int leg, int phase) {
        int phaseIndex = ((phase-1)%4)*12;
        int legOffset = (leg-1)*4;
        double[] legPhase = new double[4];
        legPhase[0] = d2r(values[phaseIndex+legOffset]);
// in radians
        legPhase[1] = d2r(values[phaseIndex+legOffset+1]);
// in radians
        legPhase[2] = values[phaseIndex+legOffset+2]/100.0;
// in meters
        legPhase[3] = values[phaseIndex+legOffset+3]/100.0;
// in meters
        return legPhase;
    }

    public double d2r(double degree) {
        return degree * Math.PI / 180.0;
    }

    public void mutate(double percentage) {
        boolean changed = false;
        for (int i=0; i<48; i++) {
```

```
            if (generator.nextDouble() < percentage) {
                mutate(i);
                changed = true;
            }
        }
        if (changed) {
            updateScore();
        }
    }

    private void mutate(int index) {
        // select a random position within the leg constraints for the
            //appropriate joint of leg segment
        if (index%4 == 0) {
            values[index] = LegConstraints.randomVH(generator);
        } else if (index%4 == 1) {
            values[index] = LegConstraints.randomVK(generator);
        } else if (index%4 == 2) {
            values[index] = LegConstraints.randomUL(generator);
        }else {
            values[index] = LegConstraints.randomLL(generator);
        }
        numberOfMutations++;
    }

    public double[] getValues() {
        return values;
    }

    public double getScore() {
        return score;
    }

    public double getEfficiencyObjectiveScore() {
        return efficiencyObjectiveScore;
    }

    public double getCumulativeScore() {
        return cumulativeScore;
    }

    public void setCumulativeScore(double cumulativeScore) {
        this.cumulativeScore = cumulativeScore;
    }

    public void updateScore() {
    // this is the value of the objective function for the chromosome.
        score = FitnessFunction.calculateTotalFitness(this);
    }

    public int compareTo(Chromosome c) {
        if (c != null) {
            if (Double.isNaN(score)) {
                if (Double.isNaN(c.getScore())) {
                    return 0;
```

```
                } else {
                    return 1;
                }
            }
            if (score < c.getScore()) {
                return 1;
            } else if (score > c.getScore()){
                return -1;
            } else {
                return 0;
                // if they have the same score there isn't a good way
           //to break the tie consistently so just use their
            }
        } else {
            return -1;
        }
    }

    public boolean isValidStride() {
        return !onKnees() && correctDirections();
    }

    public boolean onKnees() {
        // check to see if any leg is supported by the vertical knee
           //during the supported phases
        for (int i=0; i<=32; i+=4) {
            if (onKnee(i)) {
                return true;
            }
        }
        return false;
    }

    public boolean onKnee(int startIndex) {
        return (values[startIndex]+values[startIndex+1] > 180);
    }

    public boolean correctDirections() {
        // we want to check that the vertical shoulder joint is always
           //traveling in the proper direction between phases
        // for all legs
        if (!correctDirections(0)) { // leg 1
            return false;
        } else if (!correctDirections(4)) { // leg 2
            return false;
        } else if (!correctDirections(8)) { // leg 3
            return false;
        }
        return true;
    }

    public boolean correctDirections(int index) {
        // we want to check that the vertical shoulder joint is always
           //traveling in the proper direction between phases
        // for one legs
```

```
        if (values[index] > values[index+12]) {
// leg must rotate backward
            return false;
        } else if (values[index+12] > values[index+24]) {
// leg must rotate backward
            return false;
        } else if (values[index+24] < values[index+36]) {
// leg must rotate forward
            return false;
        } else if (values[index+36] < values[index]) {
// leg must rotate forward
            return false;
        }
        return true;
    }

    public void setScores(double efficiencyObjectiveScore,
                        double draggingObjectiveScore,
                           double stabilityObjectiveScore) {
        this.efficiencyObjectiveScore = efficiencyObjectiveScore;
        this.draggingObjectiveScore = draggingObjectiveScore;
        this.stabilityObjectiveScore = stabilityObjectiveScore;
    }

    public void outputChromosome() {
        System.out.println("Score: " + score);
        System.out.println("A1: " +
           getAlleleByIndex(1).outputAllele());
        System.out.println("A2: " +
           getAlleleByIndex(2).outputAllele());
        System.out.println("A3: " +
           getAlleleByIndex(3).outputAllele());
        System.out.println("A4: " +
           getAlleleByIndex(4).outputAllele());
        System.out.println("Efficieicy objective: " +
           efficiencyObjectiveScore);
        System.out.println("Dragging objective: " +
           draggingObjectiveScore);
        System.out.println("Stability objective: " +
           stabilityObjectiveScore);
        System.out.println("Mutations: " + numberOfMutations);
    }
}
```

```
package chromosome;

/**
 * Created by rcafarel on 02/25/2017.
 */
public class Allele {

    private double[] values = new double[12];
    /** positions in values represent the following
     * hip from leg 1
     * knee from leg 1
     * upper leg from leg 1
     * lower leg from leg 1
     * hip from leg 2
     * knee from leg 2
     * upper leg from leg 2
     * lower leg from leg 2
     * hip from leg 3
     * knee from leg 3
     * upper leg from leg 3
     * lower leg from leg 3
     */

    public Allele(double[] values) {
        this.values = values;
    }

    public Allele(String[] stringValues) {
        for (int i=0; i<12; i++) {
            if (i%4 < 2){
                this.values[i] = Integer.valueOf(stringValues[i]);
            } else {
                this.values[i] = Integer.valueOf(stringValues[i]);
            }
        }
    }

    public double[] getValues() {
        return values;
    }

    public String outputAllele() {
        String output = "";
        output += " h1: " + fixedWidth(values[0]);
        output += " k1: " + fixedWidth(values[1]);
        output += " u1: " + fixedWidth(values[2]);
        output += " l1: " + fixedWidthBetweenLegs(values[3]);
        output += " h2: " + fixedWidth(values[4]);
        output += " k2: " + fixedWidth(values[5]);
        output += " u2: " + fixedWidth(values[6]);
        output += " l2: " + fixedWidthBetweenLegs(values[7]);
        output += " h3: " + fixedWidth(values[8]);
        output += " k3: " + fixedWidth(values[9]);
        output += " u3: " + fixedWidth(values[10]);
        output += " l3: " + fixedWidth(values[11]);
```

```
        return output;
    }

    private String fixedWidth(double val) {
        return String.format("%-3.1f", val);
    }

    private String fixedWidthBetweenLegs(double val) {
        return String.format("%-10.1f", val);
    }
}
```

```
package chromosome;

/** Created by rcafarel on 03/31/2017.
 */
public class FitnessFunction {

    private static double efficiencyObjectiveWeight;
    private static double draggingObjectiveWeight;
    private static double stabilityObjectiveWeight;

    public FitnessFunction(double efficiencyObjectiveWeight,
           double draggingObjectiveWeight,
            double stabilityObjectiveWeight) {
        // the sum of the three weights should be one so that we can
           //standardize the fitness between [0,1].
        this.efficiencyObjectiveWeight = efficiencyObjectiveWeight;
        this.draggingObjectiveWeight = draggingObjectiveWeight;
        this.stabilityObjectiveWeight = stabilityObjectiveWeight;
    }

    public static double calculateTotalFitness(Chromosome c) {
// value of the multi-objective function for the chromosome.
        double efficiencyObjective = efficiencyObjective(c);

        DraggingObjective draggingObjective = new DraggingObjective(c);
        double draggingObjectiveScore =
                 draggingObjective.draggingObjective;

        // we may want to vary the buffer constant to look at different
           //stability factors
        StabilityObjective stabilityObjective =
           new StabilityObjective(draggingObjective, 0.0025);
        double stabilityObjectiveScore =
           stabilityObjective.stabilityObjective;

        c.setScores(efficiencyObjective, draggingObjectiveScore,
           stabilityObjectiveScore);

        return efficiencyObjectiveWeight*efficiencyObjective +
                draggingObjectiveWeight*draggingObjectiveScore +
                stabilityObjectiveWeight*stabilityObjectiveScore;
    }

    public static double efficiencyObjective(Chromosome c) {
        // thesis objective (d/e)
        double totalPower = Power.calculateTotalPower(c);
        double maxLegDistance =  2.0*
           (LegConstraints.MAX_UL+LegConstraints.MAX_LL)/100.0;
        double totalPower2 = Power.calculateTotalPower(c);
        double minDistance = Math.abs(Distance.calculateDistance(c));
        // relative distance divided by total power, calculation to be
           //between 0 and 1.
        return (minDistance/maxLegDistance) / totalPower;
    }
}
```

```
package chromosome;

/**
 * Created by rcafarel on 03/15/2017.
 */
public class Distance {

    public static double calculateDistance(Chromosome c) {
        // assume distance is the minimum distance traveled for a
           //single leg
        // this assumption needs to be changed based on the actual
           //distance the robot body is traveling
        double minDistance =
           calculateDistanceForLegForOnePhaseStartingAtIndex(c, 0) -
           calculateDistanceForLegForOnePhaseStartingAtIndex(c, 24);
           // leg 1
        minDistance = Math.min(minDistance,
           calculateDistanceForLegForOnePhaseStartingAtIndex(c, 4) -
           calculateDistanceForLegForOnePhaseStartingAtIndex(c, 28));
           // leg 2
        minDistance = Math.min(minDistance,
           calculateDistanceForLegForOnePhaseStartingAtIndex(c, 8) -
           calculateDistanceForLegForOnePhaseStartingAtIndex(c, 32));
      // leg 3
        return minDistance;
    }

    public static double
      calculateDistanceForLegForOnePhaseStartingAtIndex(
           Chromosome c, int index) {
        double ulHorizontalDistance =
           c.values[index+2]/100.0*Math.cos(d2r(c.values[index]));
           // ul * cos(vh)
        double llHorizontalDistanceAngle = c.values[index] +
           c.values[index+1]; // 180 - vh - vk
        double llHorizontalDistance =
                c.values[index+3]/100.0*
                 Math.cos(d2r(llHorizontalDistanceAngle));
                 // ll * cos(180 - vh - vk)
        return llHorizontalDistance + ulHorizontalDistance;
    }

    public static double d2r(double degree) {
        return degree * Math.PI / 180.0;
    }

}
```

```
package chromosome;

/**
 * Created by rcafarel on 10/30/2017.
 */
public class Power {

    public static double rad90 = Math.PI/2.0;
    public static double gravity = 9.81; // acceleration due to gravity

    public static double l0Initial = 12.0;
// length of the upper leg segment
    public static double l1Initial = 12.0;
// length of the lower leg segment
    public static double l0Final = 12.0;
// length of the upper leg segment
    public static double l1Final = 12.0;
// length of the lower leg segment
    public static double mass_LegSegment = 1.0;
// mass of one leg segment
    public static double mass_RobotBody = 10.0;
// mass of robot body

    // mass includes robot body and unsupported legs
    public static double mass_RobotAndUnsupportedLegs =
            mass_RobotBody + 6.0*mass_LegSegment;
    // weight includes robot body and unsupported legs
    public static double weight_RobotAndUnsupportedLegs =
            mass_RobotAndUnsupportedLegs*gravity;
    public static double weight_LegSegment = mass_LegSegment*gravity;
// weight of one leg segment

    public static double calculateTotalPower(Chromosome c) {
        double totalPower = 0.0;
// total power is the sum of the power over all phases

        totalPower += calculatePowerInPhase(c, 1);
        totalPower += calculatePowerInPhase(c, 2);
        totalPower += calculatePowerInPhase(c, 3);
        totalPower += calculatePowerInPhase(c, 4);

        return totalPower;
    }

    public static double calculatePowerInPhase(Chromosome c, int phase)
{
        double powerInPhase = 0.0;

        int initialPhaseIndex = phase;
        int finalPhaseIndex = phase+1;
// if phase is 4 then final index will be fixed by chromosome
        boolean supported = (phase < 3);

        double[] legPositionInitial = new double[4];
        double[] legPositionFinal = new double[4];
```

```
        legPositionInitial = c.getLeg_Phase_(1, initialPhaseIndex);
        legPositionFinal = c.getLeg_Phase_(1, finalPhaseIndex);
        powerInPhase += calculatePowerBetween(legPositionInitial,
                                  legPositionFinal, supported);

        legPositionInitial = c.getLeg_Phase_(2, initialPhaseIndex);
        legPositionFinal = c.getLeg_Phase_(2, finalPhaseIndex);
        powerInPhase += calculatePowerBetween(legPositionInitial,
                                  legPositionFinal, supported);

        legPositionInitial = c.getLeg_Phase_(3, initialPhaseIndex);
        legPositionFinal = c.getLeg_Phase_(3, finalPhaseIndex);
        powerInPhase += calculatePowerBetween(legPositionInitial,
                                  legPositionFinal, supported);

        return powerInPhase;
    }

    public static double calculatePowerBetween(
            double[]legPositionInitial, double[] legPositionFinal,
            boolean supported) {
        double q0Initial = legPositionInitial[0];
        double q1Initial = legPositionInitial[1];

        double q0Final = legPositionFinal[0];
        double q1Final = legPositionFinal[1];

        l0Initial = legPositionInitial[2];
        l1Initial = legPositionInitial[3];

        l0Final = legPositionFinal[2];
        l1Final = legPositionFinal[3];

        // linear power based on ratio of difference in leg length over
            //0.37 cm (max change per second) times 4.8 Watts
        // abs because there is no backdrive
        double l0_linearPower_beforeRotational =
                Math.abs(getUpperLegPrismaticPower(q0Final,
                        l0Final - l0Initial, supported));
        double l1_linearPower_beforeRotational =
                Math.abs(getLowerLegPrismaticPower(q0Final, q1Final,
                        l1Final - l1Initial, supported));
        double linearPower_beforeRotational =
                l0_linearPower_beforeRotational +
                     l1_linearPower_beforeRotational;


        double l0_linearPower_afterRotational =
                Math.abs(getUpperLegPrismaticPower(q0Initial,
                     l0Final - l0Initial, supported));
        double l1_linearPower_afterRotational =
                Math.abs(getLowerLegPrismaticPower(q0Initial,
                     q1Initial, l1Final - l1Initial, supported));
        double linearPower_afterRotational =
```

```
            l0_linearPower_afterRotational +
                  l1_linearPower_afterRotational;

    // re-order initial and final to make the loops easier to read
    if (q0Initial > q0Final) {
        double temp = q0Initial;
        q0Initial = q0Final;
        q0Final = temp;
    }
    if (q1Initial > q1Final) {
        double temp = q1Initial;
        q1Initial = q1Final;
        q1Final = temp;
    }

    // pre calculate the range and interval size to save time and
       //make the code more readable
    // 1000 was picked arbitrarily as a good approximation of a
       //true integral
    // -- 10000 was attempted but takes to long to solve and didn't
       //provide significantly better results
    // -- 100 was attempted but the approximation wasn't as good
       //but does run very quickly
    double numberOfIntervals = 100.0;
    double q0Range = q0Final - q0Initial;
    double q0Interval = q0Range / numberOfIntervals;
    double q1Range = q1Final - q1Initial;
    double q1Interval = q1Range / numberOfIntervals;

    double totalRotationalForce_beforePrismatic = 0.0;
    double totalRotationalForce_afterPrismatic = 0.0;

    // loop over the values of q0 and q1 to determine the volume
       //under the curve of the force function
    // q0 is the angle of the hip joint
    // q1 is the angle of the knee joint
    if (q0Initial == q0Final) {
        if (q1Initial == q1Final) {
            return linearPower_afterRotational;
        } else {
            for (double q1 = q1Initial; q1 < q1Final;
                   q1 += q1Interval) {
    // force at a point times the area defined in the q0 q1 plane
                double instantForce_beforePrismatic =
                   Math.abs(force(q0Initial, q1,
                         l0Initial, l1Initial, supported));
                totalRotationalForce_beforePrismatic +=
                   instantForce_beforePrismatic * q1Interval;
                double instantForce_afterPrismatic =
                   Math.abs(force(q0Initial, q1, l0Final, l1Final,
                               supported));
                totalRotationalForce_afterPrismatic +=
                   instantForce_afterPrismatic * q1Interval;
            }
        }
```

```
        } else {
            for (double q0 = q0Initial; q0 < q0Final; q0 += q0Interval)
{
                if (q1Initial == q1Final) {
       // force at a point times the area defined in the q0 q1 plane
                    double instantForce_beforePrismatic =
                       Math.abs(force(q0, q1Initial,
                            l0Initial, l1Initial, supported));
                    totalRotationalForce_beforePrismatic +=
                       instantForce_beforePrismatic * q0Interval;
                    double instantForce_afterPrismatic =
                       Math.abs(force(q0, q1Initial, l0Final, l1Final,
                            supported));
                    totalRotationalForce_afterPrismatic +=
                       instantForce_afterPrismatic * q0Interval;
                } else {
                    for (double q1 = q1Initial; q1 < q1Final;
                            q1 += q1Interval) {
      // force at a point times the area defined in the q0 q1 plane
                        double instantForce_beforePrismatic =
                             Math.abs(force(q0, q1, l0Initial,
                                   l1Initial, supported));
                        totalRotationalForce_beforePrismatic +=
                             instantForce_beforePrismatic *
                                   q0Interval * q1Interval;
                        double instantForce_afterPrismatic =
                             Math.abs(force(q0, q1, l0Final, l1Final,
                                   supported));
                        totalRotationalForce_afterPrismatic +=
                             instantForce_afterPrismatic *
                                   q0Interval * q1Interval;
                    }
                }
            }
        }

        // rotational force is in Nm, we need to convert it to Watts
        double speed = rad2deg(q0Range+q1Range)*60.0/360.0; // RPM

        double rotationalPower_beforePrismatic =
           totalRotationalForce_beforePrismatic * speed * 0.1047;
        double rotationalPower_afterPrismatic =
           totalRotationalForce_afterPrismatic * speed * 0.1047;
        if (rotationalPower_beforePrismatic +
                 linearPower_afterRotational <
                linearPower_beforeRotational +
                 rotationalPower_afterPrismatic) {
            return rotationalPower_beforePrismatic +
                       linearPower_afterRotational;
        } else {
            return rotationalPower_afterPrismatic +
                       linearPower_beforeRotational;
        }
    }
```

```
    public static double rad2deg(double radians) {
        return radians*180.0/Math.PI;
    }

    // get the weight of supported by the linear motor
    // if supported then the weight is 1/3 the weight of the robot body
            //and one of the unsupported legs
    // if unsupported then the weight its own weight plus the lower leg
            //segment
    public static double getUpperLegPrismaticPower(
            double q0, double distance, boolean supported) {
        double weight = weight_RobotAndUnsupportedLegs/3.0;
        if (!supported) {
            weight = 2.0*weight_LegSegment;
        }
        double forceAlongAxis = weight * Math.cos(rad90-q0); // in Nm
        return getPrismaticPower(forceAlongAxis, distance);
    }
    // get the weight of supported by the linear motor
    // if supported then the weight is 1/3 the weight of the robot
      //body,
    // one of the unsupported legs and the upper leg segment
    // if unsupported then the weight its own weight
    public static double getLowerLegPrismaticPower(
            double q0, double q1, double distance, boolean supported) {
        double weight = weight_RobotAndUnsupportedLegs/3.0 +
                              weight_LegSegment;
        if (!supported) {
            weight = weight_LegSegment;
        }
        double forceAlongAxis = weight * Math.cos(q0+q1-rad90);
        return getPrismaticPower(forceAlongAxis, distance);
    }

    // calculate the power required to move the forceAlongAxis up the
      //inclined plane of the worm gear
    // assume d=0.01m, lead=0.006m, mu=0.5, voltage=12
    public static double getPrismaticPower(
                  double forceAlongAxis, double distance) {
        double lead = 0.006, d=0.01, mu=0.5, voltage=12.0;
        double threadAngle_radians = Math.atan2(lead, d*Math.PI);
// in radians
        double angleOfFriction_radians = Math.atan(mu); // in radians
        double torque = forceAlongAxis * d/2.0 *
Math.tan(threadAngle_radians+angleOfFriction_radians); // in Nm
        double speed = (distance/lead)*60.0; // in RPM
        return torque * speed * 0.1047; // in W
    }


    public static double force(double q0, double q1, double l0,
                  double l1, boolean supported) {
        double distance_endEffectorToRobotBody =
                  distance_endEffectorToRobotBody(q1, l0, l1);
        double distance_endEffectorToCenterOfMassOfUpperLeg =
```

```
            distance_endEffectorToCenterOfMassOfUpperLeg(q1, l0, l1);
        // theta is the angle between the lower leg segment and the
            //line between the shoulder and the end effector
        double theta = theta(distance_endEffectorToRobotBody, l0, l1);
        // beta is the angle between the lower leg segment and the line
            //between
        // the center of mass of the upper leg segment and the end
            //effector
double beta = beta(distance_endEffectorToCenterOfMassOfUpperLeg,
            l0, l1);

        // fx is the force in the x-direction due to the weights of the
            //robot body,
        // unsupported legs, and supporting leg segments
        double fx = Math.abs(fx(q0, q1, l0, l1,
                distance_endEffectorToRobotBody,
                 distance_endEffectorToCenterOfMassOfUpperLeg, theta,
                 beta, supported));
        // fz is the force in the z-direction due to the weights of the
            //robot body,
        // unsupported legs, and supporting leg segments
        double fz = Math.abs(fz(q0, q1, l0, l1,
                distance_endEffectorToRobotBody,
                 distance_endEffectorToCenterOfMassOfUpperLeg, theta,
                 beta, supported));
        // omega is the torque caused by the weights of the robot body,
        // unsupported legs, and supporting leg segments about the y-
            //axis
        double omega = Math.abs(omega(q0, q1, l0, l1,
                distance_endEffectorToRobotBody,
                 distance_endEffectorToCenterOfMassOfUpperLeg, theta,
                 beta, supported));

        // jeeT is the jacobian matrix which is used to transform the
            //Fx forces into the Fq forces that we can use to
        // calculate the total forces on the motors
        double jeeT11 = jeeT11(q0, q1, l0, l1);
        double jeeT12 = jeeT12(q0, q1, l0, l1);
        double jeeT21 = jeeT21(q0, q1, l0, l1);
        double jeeT22 = jeeT22(q0, q1, l0, l1);

        // rotational force, for a given q0 and q1, is calculated by
            //multiplying jeeT by Fx and summing the elements
        double rotationalForce = fx*jeeT11 + fz*jeeT12 + omega;
        rotationalForce += fx*jeeT21 + fz*jeeT22 + omega;

        return Math.abs(rotationalForce);
    }

    public static double jeeT11(double q0, double q1, double l0,
                 double l1) {
        return -l1*Math.sin(q0+q1) - l0*Math.sin(q0);
    }

    public static double jeeT12(double q0, double q1, double l0,
```

```
            double l1) {
        return l1*Math.cos(q0+q1) + l0*Math.cos(q0);
    }

    public static double jeeT21(double q0, double q1, double l0,
            double l1) {
        return -l1*Math.sin(q0+q1);
    }

    public static double jeeT22(double q0, double q1, double l0,
            double l1) {
        return l1*Math.cos(q0+q1);
    }

    public static double fx(double q0, double q1, double l0,
            double l1, double distance_endEffectorToRobotBody,
            double distance_endEffectorToCenterOfMassOfUpperLeg,
             double theta, double beta, boolean supported) {
        // sum of (fx for robot body and unsupported legs) + (fx for
            //upper leg segment) + (fx for lower leg segment)
        double robotWeight = supported ?
                weight_RobotAndUnsupportedLegs/3.0 : 0;
        return distance_endEffectorToRobotBody*
                robotWeight*Math.cos(theta+q0)+
                distance_endEffectorToCenterOfMassOfUpperLeg*
                weight_LegSegment * Math.cos(beta+q0)
               + (3.0/4.0*l1)*weight_LegSegment*Math.cos(q0+q1);
    }

    public static double fz(double q0, double q1, double l0, double l1,
                double distance_endEffectorToRobotBody,
                double distance_endEffectorToCenterOfMassOfUpperLeg,
                double theta, double beta, boolean supported) {
        // sum of (fz for robot body and unsupported legs) + (fz for
            //upper leg segment) + (fz for lower leg segment)
        double robotWeight = supported ?
                weight_RobotAndUnsupportedLegs/3.0 : 0;
        return distance_endEffectorToRobotBody*
                robotWeight*Math.sin(theta+q0)+
                distance_endEffectorToCenterOfMassOfUpperLeg*
                weight_LegSegment *Math.sin(beta+q0)
               + (3.0/4.0*l1)*weight_LegSegment*Math.sin(q0+q1);
    }

    public static double omega(double q0, double q1, double l0,
            double l1, double distance_endEffectorToRobotBody,
           double distance_endEffectorToCenterOfMassOfUpperLeg,
           double theta, double beta, boolean supported) {
        // sum of (torque for robot body and unsupported legs) +
      //(torque for upper leg segment) + (torque for lower leg segment)
        // torque is calculated as the perpendicular force to the line
      //between the end effector and the center of mass of the object
      //supplying the torque multiplied by the distance between the two
        double robotWeight = supported ?
            weight_RobotAndUnsupportedLegs/3.0 : 0;
```

```
        return robotWeight*Math.sin(rad90-q0-theta)*
                  distance_endEffectorToRobotBody +
                  weight_LegSegment*Math.sin(rad90-q0-beta)*
                  distance_endEffectorToCenterOfMassOfUpperLeg
                + weight_LegSegment*Math.sin(rad90-q0-q1)*(3.0/4.0*l1);
    }

    public static double distance_endEffectorToRobotBody(double q1,
                  double l0, double l1) {
        return Math.sqrt((l0*l0)+(l1*l1)-2.0*l0*l1*
                        Math.cos(Math.PI-q1));
    }

    public static double distance_endEffectorToCenterOfMassOfUpperLeg(
                  double q1, double l0, double l1) {
        return Math.sqrt((3.0/4.0*l0*3.0/4.0*l0) + (l1*l1) -
                  2.0*(3.0/4.0*l0)*l1*Math.cos(Math.PI-q1));
    }

    // had to add error checking for double precision issues
    public static double theta(double distance_endEffectorToRobotBody,
                  double l0, double l1) {
        double numerator = ((l1*l1) - (l0*l0) -
            (distance_endEffectorToRobotBody*
                  distance_endEffectorToRobotBody));
        double denominator = (-2.0*l0*distance_endEffectorToRobotBody);
        if (numerator/denominator > 1) {
            return Math.acos(1.0);
        } else if (numerator/denominator < -1) {
            return Math.acos(-1.0);
        } else {
            return Math.acos(numerator / denominator);
        }
    }

    // had to add error checking for double precision issues
    public static double beta(
            double distance_endEffectorToCenterOfMassOfUpperLeg,
            double l0, double l1) {
        double numerator = ((l1*l1)-(3.0/4.0*l0*3.0/4.0*l0) -
            (distance_endEffectorToCenterOfMassOfUpperLeg*
            distance_endEffectorToCenterOfMassOfUpperLeg));
        double denominator = (-2.0*(3.0/4.0*l0)*
                  distance_endEffectorToCenterOfMassOfUpperLeg);
        if (numerator/denominator > 1) {
            return Math.acos(1.0);
        } else if (numerator/denominator < -1) {
            return Math.acos(-1.0);
        } else {
            return Math.acos(numerator / denominator);
        }
    }
}
```

```
package chromosome;

/**
 * Created by rcafarel on 11/03/2017.
 */
public class DraggingObjective {
    // constant
    public static final double rad180 = Math.PI;

    // input
    public Chromosome chromosome;

    // calculated values
    public int draggingObjective = 0;

    // -- needed in this class
    public double[] hipFootDistancesSupported = new double[3];
    public double[] hipFootDistancesUnsupported = new double[3];
    public double[] angledHeightOfLegSupported = new double[3];
    public double[] angledHeightOfLegUnsupported = new double[3];
    public double[] heightOfLegSupported = new double[3];
    public double angleFrontToBackInRadians;
    public double angleSideToSideInRadians;

    // -- needed in other classes
    // angle behind leg 1 between hip-foot and upper leg
    public double upperLeg1_hipFoot1_Angle_radians;
    // angle behind leg 1 between hip-foot and robot body
    public double hipFoot1_robotBody_Angle_radians;
    // distance between foot of leg 1 and shoulder of leg 3
    public double foot1_shoulder3_Distance;
    // angle between the line defined by the foot of leg 1 and shoulder
      //of leg 3 and the hip-foot line of leg 1
    public double foot1shoulder3_hipFoot1_Angle_radians;
    // angle between the line defined by the foot of leg 1 and shoulder
      //of leg 3 and the robot body
    public double foot1shoulder3_robotBody_Angle_radians;
    // angle behind leg 1 between hip-foot and upper leg
    public double upperLeg3_hipFoot3_Angle_radians;
    // angle ahead of leg 3 between hip-foot and robot body
    public double hipFoot3_robotBody_Angle_radians;
    // angle between the line defined by the foot of leg 1 and shoulder
      //of leg 3 and the hip-foot line of leg 3
    public double foot1shoulder3_hipFoot3_Angle_radians;
    // distance between foot of leg 1 and foot of leg 3 on the ground
    public double foot1_foot3_Distance;
    // angle between the hip-foot line of leg 3 and the ground
    public double hipFoot3_ground_Angle_radians;
    // angle between the line defined by the foot of leg 1 and shoulder
      //of leg 3 and the hip-foot line of leg 3
    public double foot1shoulder3_ground_Angle_radians;

    public double[] endEffector_x_leg_phase = new double[3];
    public double[] endEffector_y_leg_phase = new double[3];
    public double cog_x, cog_y;
```

```
public DraggingObjective(Chromosome chromosome) {
    this.chromosome = chromosome;
    calculateDraggingObjective();
}

// The dragging objective is a binary objective (0 or 1).
// If the height legs of the supported tripod are less than the
  //legs of the unsupported tripod then return 0 else 1.
// First, find the front-to-back angle of the robot body in the
  //world system reference frame.
// Second, find the side-to-side angle of the robot body in the
  //world system reference frame.
// Third, use the angles to determine thr true height of each leg.
public void calculateDraggingObjective() {
    calculateAngleFrontToBackInRadians();
    calculateAngleSideToSideInRadians();
    calculateHeightOfEachLeg();
    compareMaxAngledHeightOfLegUnsupported();
}

public void compareMaxAngledHeightOfLegUnsupported() {
    // calculate the distance between the hip of the respective leg
  //and the ground in the plane of the robot body
    double maxUnsupportedLeg1 = heightOfLegSupported[0];
    double maxUnsupportedLeg2 = heightOfLegSupported[1];
    double maxUnsupportedLeg3 = heightOfLegSupported[2];

    double tan_angleSideToSideInRadians = Math.tan((rad180/2.0)-
        angleSideToSideInRadians);
    if(tan_angleSideToSideInRadians != 0) {
        double distanceToGroundFromHip1 =
             angledHeightOfLegSupported[0] /
                tan_angleSideToSideInRadians;
        maxUnsupportedLeg1 =
              (distanceToGroundFromHip1 - (2.0 *
                LegConstraints.TRIPOD_WIDTH) +
                LegConstraints.BODY_WIDTH) *
                    tan_angleSideToSideInRadians;

        double distanceToGroundFromHip2 =
             angledHeightOfLegSupported[1] /
                tan_angleSideToSideInRadians;
        maxUnsupportedLeg2 =
              (distanceToGroundFromHip2 +
                LegConstraints.BODY_WIDTH) *
                    tan_angleSideToSideInRadians;

        double distanceToGroundFromHip3 =
             angledHeightOfLegSupported[2] /
                tan_angleSideToSideInRadians;
        maxUnsupportedLeg3 =
              (distanceToGroundFromHip3 - (2.0 *
                LegConstraints.TRIPOD_WIDTH) +
```

```
                    LegConstraints.BODY_WIDTH) *
                         tan_angleSideToSideInRadians;
        }

        if (maxUnsupportedLeg1 > angledHeightOfLegUnsupported[0] &&
                maxUnsupportedLeg2 > angledHeightOfLegUnsupported[1] &&
                maxUnsupportedLeg3 > angledHeightOfLegUnsupported[2]) {
            draggingObjective = 1;
        }
    }

    // First, find the hip foot distance for each leg
    // Second, find the angled height of each leg from the xz-plane
      //perspective of leg 1
    public void calculateAngleFrontToBackInRadians() {
        calculateHipFootDistances();
        calculateAngledHeight();
    }

    // First, calculate mid-point height between legs 1 and 3
    // Second, calculate angle between mid-point height line and the
      //ground
    public void calculateAngleSideToSideInRadians() {
        // average of angled height of legs 1 and 3
        double angledMidpointHeight = (angledHeightOfLegSupported[2] +
            angledHeightOfLegSupported[0]) / 2.0;

        // difference between the height of the angledMidpointHeight
      //and the angled height of leg 2
        double crosssectionalHeightDifference = angledMidpointHeight –
            angledHeightOfLegSupported[1];

        // since the leg are perpendicular to the body of the robot we
      // know that the triangle below the line
        // defined by the crosssectionalHeightDifference is a right
      //triangle so we know that the angle between
        // the ground and the angled midpoint line is the arctan of the
      //opposite over the adjacent.
        if (Math.abs(crosssectionalHeightDifference) < 0.01) {
            angleSideToSideInRadians = rad180 / 2.0;
        } else {
            angleSideToSideInRadians =
                 Math.atan2(LegConstraints.TRIPOD_WIDTH,
                      crosssectionalHeightDifference);
        }
    }

    public void calculateHeightOfEachLeg() {
        // supported legs can be calculated directly using the
            //angleSideToSideInRadians and the angled height
        heightOfLegSupported[0] = angledHeightOfLegSupported[0] *
            Math.sin(angleSideToSideInRadians);
        heightOfLegSupported[1] = angledHeightOfLegSupported[1] *
            Math.sin(angleSideToSideInRadians);
        heightOfLegSupported[2] = angledHeightOfLegSupported[2] *
```

```
            Math.sin(angleSideToSideInRadians);

        // y positions can be calculated using the
//angleSideToSideInRadians with the appropriate offset based on the leg
        endEffector_y_leg_phase[0] = - angledHeightOfLegSupported[0] *
            Math.cos(angleSideToSideInRadians);
        endEffector_y_leg_phase[1] = - angledHeightOfLegSupported[1] *
            Math.cos(angleSideToSideInRadians) +
                 LegConstraints.TRIPOD_WIDTH /
                  Math.sin(angleSideToSideInRadians);
        endEffector_y_leg_phase[2] = - angledHeightOfLegSupported[2] *
            Math.cos(angleSideToSideInRadians);
        cog_y = (LegConstraints.TRIPOD_WIDTH -
            LegConstraints.BODY_WIDTH/2.0) /
                  Math.sin(angleSideToSideInRadians);

    }

    // First, calculate hip-foot distances at the beginning of phase 2
      //and put them into hipFootDistancesSupported
    // Second, calculate hip-foot distances at the beginning of phase 4
      //and put them into hipFootDistancesUnsupported
    public void calculateHipFootDistances() {
        // leg is configured as [q0, q1, l0, l1]
        // phase 2 is for the supported tripod
        double[] leg = chromosome.getLeg_Phase_(1,2);
        hipFootDistancesSupported[0] =
            Geometry.lawOfCosines_length(leg[2], leg[3],
                  rad180-leg[1]);
        leg = chromosome.getLeg_Phase_(2,2);
        hipFootDistancesSupported[1] =
            Geometry.lawOfCosines_length(leg[2], leg[3],
                  rad180-leg[1]);
        leg = chromosome.getLeg_Phase_(3,2);
        hipFootDistancesSupported[2] =
            Geometry.lawOfCosines_length(leg[2], leg[3],
                  rad180-leg[1]);

        // phase 4 is for the unsupported tripod
        leg = chromosome.getLeg_Phase_(1,4);
        hipFootDistancesUnsupported[0] =
            Geometry.lawOfCosines_length(leg[2], leg[3],
                  rad180-leg[1]);
        leg = chromosome.getLeg_Phase_(2,4);
        hipFootDistancesUnsupported[1] =
            Geometry.lawOfCosines_length(leg[2], leg[3],
                  rad180-leg[1]);
        leg = chromosome.getLeg_Phase_(3,4);
        hipFootDistancesUnsupported[2] =
            Geometry.lawOfCosines_length(leg[2], leg[3],
                  rad180-leg[1]);
    }

    // First, find all angles within the quadrilateral that is defined
      //between legs 1 and 3,
```

```
    // the robot body and the ground
    // Second, calculate the angle between the angled robot body and
      //the ground
    // Third, use the angle in Second to calculate the angled heights
      //of all leg
    public void calculateAngledHeight() {
        // leg is configured as [q0, q1, l0, l1], where q0 and q1 are
      //in radians
        // phase 2 is for the supported tripod
        // phase 4 is for the unsupported tripod

        // ----------------------------------------------
        // First, find all angles within the quadrilateral that is
            //defined between legs 1 and 3,
        // the robot body and the ground
        // ----------------------------------------------
        double[] leg1 = chromosome.getLeg_Phase_(1,2);
        // angle behind leg 1 between hip-foot and upper leg
        upperLeg1_hipFoot1_Angle_radians =
            Geometry.lawOfCosines_angle(leg1[3], leg1[2],
                  hipFootDistancesSupported[0]);
        // angle behind leg 1 between hip-foot and robot body
        hipFoot1_robotBody_Angle_radians = rad180 - leg1[0] -
            upperLeg1_hipFoot1_Angle_radians;
        // distance between foot of leg 1 and shoulder of leg 3
        foot1_shoulder3_Distance =
              Geometry.lawOfCosines_length(hipFootDistancesSupported[0],
                         LegConstraints.BODY_LENGTH,
                        hipFoot1_robotBody_Angle_radians);
        // angle between the line defined by the foot of leg 1 and
      //shoulder of leg 3 and the hip-foot line of leg 1
        foot1shoulder3_hipFoot1_Angle_radians =
            Geometry.lawOfCosines_angle(LegConstraints.BODY_LENGTH,
                  hipFootDistancesSupported[0],
                         foot1_shoulder3_Distance);
        // angle between the line defined by the foot of leg 1 and
shoulder of leg 3 and the robot body
        foot1shoulder3_robotBody_Angle_radians =
            Geometry.lawOfCosines_angle(hipFootDistancesSupported[0],
            LegConstraints.BODY_LENGTH,
                         foot1_shoulder3_Distance);

        double[] leg3 = chromosome.getLeg_Phase_(3,2);
        // angle behind leg 1 between hip-foot and upper leg
        upperLeg3_hipFoot3_Angle_radians =
            Geometry.lawOfCosines_angle(leg3[3], leg3[2],
                  hipFootDistancesSupported[2]);
        // angle ahead of leg 3 between hip-foot and robot body
        hipFoot3_robotBody_Angle_radians = leg3[0] +
            upperLeg3_hipFoot3_Angle_radians;
        // angle between the line defined by the foot of leg 1 and
            //shoulder of leg 3 and the hip-foot line of leg 3
        foot1shoulder3_hipFoot3_Angle_radians =
            hipFoot3_robotBody_Angle_radians -
                 foot1shoulder3_robotBody_Angle_radians;
```

```
// distance between foot of leg 1 and foot of leg 3 on the
   //ground
foot1_foot3_Distance =
   Geometry.lawOfCosines_length(hipFootDistancesSupported[2],
         foot1_shoulder3_Distance,
         foot1shoulder3_hipFoot3_Angle_radians);
// angle between the hip-foot line of leg 3 and the ground
hipFoot3_ground_Angle_radians =
   Geometry.lawOfCosines_angle(foot1_shoulder3_Distance,
       hipFootDistancesSupported[2], foot1_foot3_Distance);
// angle between the line defined by the foot of leg 1 and
   //shoulder of leg 3 and the hip-foot line of leg 3
foot1shoulder3_ground_Angle_radians = rad180 -
   foot1shoulder3_hipFoot3_Angle_radians -
       hipFoot3_ground_Angle_radians;

// --------------------------------------------
// Second, calculate the angle between the angled robot body
   //and the ground
// --------------------------------------------

angleFrontToBackInRadians = rad180 -
   hipFoot3_ground_Angle_radians -
   hipFoot3_robotBody_Angle_radians;

// --------------------------------------------
// Third, use the angle in Second to calculate the angled
   //heights of all leg
// --------------------------------------------

// supported legs 1 and 3 an be calculated using previous
   //calculations without angleFrontToBackInRadians
angledHeightOfLegSupported[0] = hipFootDistancesSupported[0] *
       Math.sin(rad180 - foot1shoulder3_ground_Angle_radians -
         foot1shoulder3_hipFoot1_Angle_radians);
angledHeightOfLegSupported[2] = hipFootDistancesSupported[2] *
         Math.sin(hipFoot3_ground_Angle_radians);

endEffector_x_leg_phase[0] = hipFootDistancesSupported[0] *
       Math.cos(rad180 - foot1shoulder3_ground_Angle_radians -
         foot1shoulder3_hipFoot1_Angle_radians);
endEffector_x_leg_phase[2] = hipFootDistancesSupported[2] *
       Math.cos(hipFoot3_ground_Angle_radians) +
         LegConstraints.BODY_LENGTH *
       Math.cos(angleFrontToBackInRadians);

double[] leg2 = chromosome.getLeg_Phase_(2,2);
angledHeightOfLegSupported[1] =
   calculateAngledHeightOfLeg(leg2,
         hipFootDistancesSupported[1], true);

double[] leg1_unsupported = chromosome.getLeg_Phase_(1,4);
angledHeightOfLegUnsupported[0] =
   calculateAngledHeightOfLeg(leg1_unsupported,
       hipFootDistancesUnsupported[0], false);
```

```
        double[] leg2_unsupported = chromosome.getLeg_Phase_(2,4);
        angledHeightOfLegUnsupported[1] =
            calculateAngledHeightOfLeg(leg2_unsupported,
                hipFootDistancesUnsupported[1], false);

        double[] leg3_unsupported = chromosome.getLeg_Phase_(3,4);
        angledHeightOfLegUnsupported[2] =
            calculateAngledHeightOfLeg(leg3_unsupported,
                hipFootDistancesUnsupported[2], false);

    }

    // This method calculates the angled height of a given leg (q0, q1,
      //l0, l1)
    // and a previously calculated hip foot distance
    public double calculateAngledHeightOfLeg(double[] leg,
            double hipFootDistance, boolean calculateEndEffectorLeg2) {
        // angle behind leg between hip-foot and upper leg
        double upperLeg_hipFoot_Angle_radians =
            Geometry.lawOfCosines_angle(leg[3], leg[2],
                  hipFootDistance);
        // angle behind leg between hip-foot and robot body
        double hipFoot_robotBody_Angle_radians = rad180 - leg[0] -
            upperLeg_hipFoot_Angle_radians;
        // angle between the hip-foot line of leg 3 and the ground
        double hipFoot_ground_Angle_radians = rad180 -
            angleFrontToBackInRadians - rad180-
            hipFoot_robotBody_Angle_radians);

        if (calculateEndEffectorLeg2) {
             endEffector_x_leg_phase[1] = hipFootDistancesSupported[1] *
                     Math.cos(hipFoot_ground_Angle_radians) +
                        LegConstraints.BODY_LENGTH/2.0 *
                     Math.cos(angleFrontToBackInRadians);
             cog_x = LegConstraints.BODY_LENGTH/2.0 *
                   Math.cos(angleFrontToBackInRadians);
        }

        // angled height of the given leg is the portion of the hipfoot
            //distance in the z-direction
        return hipFootDistance*Math.sin(hipFoot_ground_Angle_radians);
    }
}
```

```
package chromosome;

/**
 * Created by rcafarel on 11/04/2017.
 */
public class StabilityObjective {
    // input
    public DraggingObjective drag;
    public double bufferConstant;

    // calculated value
    public int stabilityObjective = 0;

    // Compare the lines between the (x, y) coordinates of the end
      //effectors with an offset of the
    // buffer constant. If the CoG is outside the area defined then the
      //stability objective is 0 else 1.
    public StabilityObjective(DraggingObjective drag,
            double bufferConstant) {
        this.drag = drag;
        this.bufferConstant = bufferConstant;
        // the (x, y) coordinates for each end effector and the center
            //of gravity have already been computed in the
        // DraggingObjective class
        // initially the dragging objective just has the end effectors
            //of phase 2, need to check phases 1 and 3 as well
        if (checkStabilityForPhaseDescribedInDrag()) {
            Chromosome chromosome_phase2 = drag.chromosome;
            // set the phase in the dragging objective to 1
            Chromosome chromosome_phase1 =
                  createChromosomeForStability(2, 1, 3, 4,
                        chromosome_phase2);
            this.drag = new DraggingObjective(chromosome_phase1);
            if (checkStabilityForPhaseDescribedInDrag()) {
                // set the phase in the dragging objective to 1
                Chromosome chromosome_phase3 =
                  createChromosomeForStability(1, 3, 2, 4,
                        chromosome_phase2);
                this.drag = new DraggingObjective(chromosome_phase3);
                if (checkStabilityForPhaseDescribedInDrag()) {
                    stabilityObjective = 1;
                }
            }
        }
    }

    public boolean checkStabilityForPhaseDescribedInDrag() {
        if (insideStableRegionBasedOnEndEffectors(0, 1, true)) {
            if (insideStableRegionBasedOnEndEffectors(1, 2, true)) {
                if (insideStableRegionBasedOnEndEffectors(2, 0, false))
{
                    return true;
                }
            }
        }
```

```
        return false;
    }

    public Chromosome createChromosomeForStability(int a1Index,
            int a2Index, int a3Index, int a4Index,
                  Chromosome oldC) {
        Allele[] alleles_phase1 = new Allele[4];
        alleles_phase1[0] = oldC.getAlleleByIndex(a1Index);
        alleles_phase1[1] = oldC.getAlleleByIndex(a2Index);
        alleles_phase1[2] = oldC.getAlleleByIndex(a3Index);
        alleles_phase1[3] = oldC.getAlleleByIndex(a4Index);
        return new Chromosome(alleles_phase1);
    }

    public boolean insideStableRegionBasedOnEndEffectors(int ee1,
            int ee2, boolean lessThan) {
        // calculate the slope between the coordinates for the end
            //effectors defined
        double slope = (drag.endEffector_y_leg_phase[ee1]-
            drag.endEffector_y_leg_phase[ee2]) /
                 (drag.endEffector_x_leg_phase[ee1]-
                        drag.endEffector_x_leg_phase[ee2]);
        double bufferOffset = bufferConstant /
                  Math.cos(Math.atan(slope));
        if (lessThan) {
            double rightSide = slope * (drag.cog_x-
                  drag.endEffector_x_leg_phase[ee1]) +
                     drag.endEffector_y_leg_phase[ee1] - bufferOffset;
            if (drag.cog_y < rightSide) {
                return true;
            }
        } else {
            double rightSide = slope * (drag.cog_x-
                  drag.endEffector_x_leg_phase[ee1]) +
                     drag.endEffector_y_leg_phase[ee1] + bufferOffset;
            if (drag.cog_y > rightSide) {
                return true;
            }
        }
        return false;
    }
}
```

```
package chromosome;

import java.util.Random;

/**
 * Created by rcafarel on 03/15/2017.
 */
public class LegConstraints {

    public static double BODY_LENGTH = 54.0/100.0; // meters
    public static double BODY_WIDTH = 42.0/100.0; // meters
    public static double TRIPOD_WIDTH = 39.0/100.0; // meters

    public static double MIN_VH = 30.0;
    public static double MAX_VH = 145.0;
    public static double randomVH(Random generator) {
      return (generator.nextDouble()*(MAX_VH-MIN_VH) + MIN_VH); }
    public static double randomVH_withMin(
            Random generator, double min) {
        return (generator.nextDouble()*(MAX_VH-min) + min); }
    public static double randomVH_withMax(
            Random generator, double max) {
        return (generator.nextDouble()*(max-MIN_VH) + MIN_VH); }

    public static double MIN_VK = 0.0;
    public static double MAX_VK = 145.0;
    public static double randomVK(Random generator) {
            return (generator.nextDouble()*(MAX_VK-MIN_VK) + MIN_VK); }

    public static double MIN_UL = 6.0;
    public static double MAX_UL = 12.0;
    public static double randomUL(Random generator) {
            return (generator.nextDouble()*(MAX_UL-MIN_UL) + MIN_UL); }

    public static double MIN_LL = 6.0;
    public static double MAX_LL = 12.0;
    public static double randomLL(Random generator) {
            return (generator.nextDouble()*(MAX_LL-MIN_LL) + MIN_LL); }
}
```

```
package chromosome;

/**
 * Created by rcafarel on 11/03/2017.
 */
public class Geometry {

    // used to find the length of the side across from angle
      //oppositeAngle_Radians
    // oppositeAngle_Radians needs to be in radians
    // the order of a and b doesn't matter
    public static double lawOfCosines_length(double a, double b,
            double oppositeAngle_Radians) {
        return Math.sqrt((a*a) + (b*b) -
            (2.0*a*b)*Math.cos(oppositeAngle_Radians));
    }

    // used to find angle across from side oppositeSide
    // angle A is returned in radians
    // the order of b and c doesn't matter
    public static double lawOfCosines_angle(double oppositeSide,
            double b, double c) {
        return Math.acos(((oppositeSide*oppositeSide) - (b*b) -
            (c*c))/(-2.0*b*c));
    }
}
```

```
package legVisual;

import chromosome.Allele;
import chromosome.Chromosome;

import javax.swing.*;
import java.awt.*;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.awt.geom.Ellipse2D;
import java.awt.geom.Line2D;

/**
 * Created by rcafarel on 04/06/2017.
 */
public class SideView extends JPanel {

    public static void main(String[] args) {
        Allele[] alleles = new Allele[4];
        alleles[0] = new Allele(new
double[]{87.6,20.3,10.3,9.6,72.5,26.8,7.9,10.7,64.5,28.4,8.7,11.8});
        alleles[1] = new Allele(new
double[]{94.2,20.8,10.4,6.2,97.6,26.9,6.6,6.2,71.6,27.8,9.2,11.9});
        alleles[2] = new Allele(new
double[]{100.8,20.2,9.2,11,98.5,24.9,10.1,7.2,76.7,29.4,6.2,12});
        alleles[3] = new Allele(new
double[]{94.6,19.7,11.3,6.2,74.1,24.7,6.9,9.8,67,28.3,7.6,11.7});
        Chromosome chromosome = new Chromosome(alleles);

        SideView sideView = new SideView();
        sideView.show(chromosome);
            // show the stride of the best chromosome
    }

    private static Chromosome c;

    public void show(Chromosome c) {
        this.c = c;
        SideView testSideView = new SideView();
        JFrame f = new JFrame("test");
        f.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        f.getContentPane().add("Center", testSideView);
        testSideView.init();
        f.pack();
        f.setSize(new Dimension(1500,750));
        f.setVisible(true);
    }

    public void paint(Graphics g) {
        g.clearRect(0,0,1500,750);
        paintLegs(g);
```

```
    }

    public void paintLegs(Graphics g) {
        Graphics2D g2 = (Graphics2D) g;
        g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                            RenderingHints.VALUE_ANTIALIAS_ON);
        g2.setPaint(Color.gray);

        for (int j = 0; j < 4; j++) { // phase j
            for (int i = 0; i < 3; i++) { // leg i
                drawLeg(g2, j, i, 10, 10, 5,
                            c.values[j * 12 + i * 4 + 0],
                            c.values[j * 12 + i * 4 + 1],
                            c.values[j * 12 + i * 4 + 2],
                            c.values[j * 12 + i * 4 + 3]);
            }
        }
    }

    public void drawLeg(Graphics2D g2, int i, int j,
                            int x, int y, int z,
                            double vh, double vk,
                            double ul, double ll) {
        int tabX = i*400;
        int groundZ = j*200+100;
        int circleRadius = 5;
        int legScaleFactor = 10;

        Leg leg = new Leg(x, y, z, vh, vk, ul, ll);
        // shoulderVertical, fixed location
        g2.setColor(leg.shoulderVertical.getColor());
        g2.draw(new Ellipse2D.Double(
            tabX+leg.shoulderVertical.getY()*
            legScaleFactor-circleRadius,
           groundZ - leg.shoulderVertical.getZ()*
                   legScaleFactor-circleRadius,
              2*circleRadius, 2*circleRadius));
        // upperArm
        g2.setColor(leg.upperArm.getColor());
        g2.draw(new Line2D.Double(tabX+leg.upperArm.getY()*
                   legScaleFactor,
                   groundZ - leg.upperArm.getZ()*
                           legScaleFactor,
                   tabX+leg.upperArm.getEndY()*
                           legScaleFactor,
                   groundZ- leg.upperArm.getEndZ()*
                           legScaleFactor));
        // elbow
        g2.setColor(leg.elbow.getColor());
        g2.draw(new Ellipse2D.Double(tabX+leg.elbow.getY()*
                   legScaleFactor-circleRadius,
                   groundZ - leg.elbow.getZ()*
                           legScaleFactor-circleRadius,
                   2*circleRadius, 2*circleRadius));
        // foreArm
```

```
        g2.setColor(leg.foreArm.getColor());
        g2.draw(new Line2D.Double(tabX+leg.foreArm.getY()*
                                        legScaleFactor,
                 groundZ - leg.foreArm.getZ()*legScaleFactor,
                 tabX+leg.foreArm.getEndY()*legScaleFactor,
                 groundZ- leg.foreArm.getEndZ()*legScaleFactor));

        g2.setColor(Color.GRAY);
    }

    public void init() {
        setBackground(Color.WHITE);
        setForeground(Color.WHITE);
    }
}
```

```
package legVisual;

import legVisual.leg.DOFJoint;
import legVisual.leg.LinearSegment;

import java.awt.*;

/**
 * Created by rcafarel on 09/21/2016.
 */
public class Leg {

    // the leg understands all aspects of an individual leg and
communicates with the body

    public DOFJoint shoulderVertical;
    public LinearSegment upperArm;
    public DOFJoint elbow;
    public LinearSegment foreArm;

    public Leg(int x, int y, int z, double vh, double vk,
           double ul, double ll) {
        shoulderVertical = new DOFJoint(x, y, z, -vh, Color.BLUE);
// assume 0 is straight right and rotates clockwise
        upperArm = new LinearSegment(x, y, z, ul, Color.GREEN);
        elbow = new DOFJoint(x, y, z, -vk, Color.BLUE);
// assume 0 is straight right
        foreArm = new LinearSegment(x, y, z, ll, Color.ORANGE);

        moveShoulderVertical(x,y,z, -vh);
    }

    public void moveShoulderVertical(int x, int y, int z,
           double newAngleVertical) {
        shoulderVertical.changePosition(x, y, z);
        shoulderVertical.changeAngle(newAngleVertical);
        moveUpperArm(x, y, z, newAngleVertical,
                 upperArm.getCurrentLength());
    }

    public void moveUpperArm(int x, int y, int z,
           double newAngleVertical, double newLength) {
        upperArm.changePosition(x, y, z);
        upperArm.changeLength(newLength);
        int eX = upperArm.getX() + (int)(upperArm.getCurrentLength());
        int eY = upperArm.getY() +
            (int)(upperArm.getCurrentLength()*
            Math.cos(newAngleVertical/180.0*Math.PI));
        int eZ = upperArm.getZ() +
            (int)(upperArm.getCurrentLength()*
            Math.sin(newAngleVertical/180.0*Math.PI));
        upperArm.setEndpoint(eX, eY, eZ);
        moveElbow(eX, eY, eZ, elbow.getCurrentAngle());
    }
```

```
    public void moveElbow(int x, int y, int z, double newAngleVertical)
{
        elbow.changePosition(x, y, z);
        elbow.changeAngle(newAngleVertical);
        moveForeArm(x, y, z,
            shoulderVertical.getCurrentAngle()+newAngleVertical,
             foreArm.getCurrentLength());
    }

    public void moveForeArm(int x, int y, int z,
            double newAngleVertical, double newLength) {
        foreArm.changePosition(x, y, z);
        foreArm.changeLength(newLength);
        int fX = foreArm.getX() + (int)(foreArm.getCurrentLength());
        int fY = foreArm.getY() +
            (int)(foreArm.getCurrentLength()*
            Math.cos(newAngleVertical/180.0*Math.PI));
        int fZ = foreArm.getZ() +
            (int)(foreArm.getCurrentLength()*
            Math.sin(newAngleVertical/180.0*Math.PI));
        foreArm.setEndpoint(fX, fY, fZ);
    }
}
```

```
package legVisual.leg;

import java.awt.*;

/**
 * Created by rcafarel on 09/23/2016.
 */
public class DOFJoint {

    private int x, y, z;
    private double currentAngle;
    private Color color;

    public DOFJoint(int x, int y, int z, double currentAngle,
            Color color) {
        this.x = x;
        this.y = y;
        this.z = z;
        this.currentAngle = currentAngle;
        this.color = color;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public int getZ() {
        return z;
    }

    public double getCurrentAngle() {
        return currentAngle;
    }

    public Color getColor() {
        return color;
    }

    public void changePosition(int x, int y, int z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }

    public void changeAngle(double newAngle) {
        this.currentAngle = newAngle;
    }
}
```

```
package legVisual.leg;

import java.awt.*;

/**
 * Created by rcafarel on 09/23/2016.
 */
public class LinearSegment {

    private int x, y, z;
    private int endX, endY, endZ;
    private double currentLength;
    private Color color;

    public LinearSegment(int x, int y, int z, double currentLength,
            Color color) {
        this.x = x;
        this.y = y;
        this.z = z;
        this.currentLength = currentLength;
        this.color = color;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public int getZ() {
        return z;
    }

    public int getEndX() {
        return endX;
    }

    public int getEndY() {
        return endY;
    }

    public int getEndZ() {
        return endZ;
    }

    public double getCurrentLength() {
        return currentLength;
    }

    public Color getColor() {
        return color;
    }
```

```
    public void changePosition(int x, int y, int z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }

    public void changeLength(double newLength) {
        this.currentLength = newLength;
    }

    public void setEndpoint(int x, int y, int z) {
        this.endX = x;
        this.endY = y;
        this.endZ = z;
    }
}
```