ArthroBOT

(Or, A Biomimetic Approach to a Distributed Leg Control Architecture For Gait Generation in Polypedal Mobile Robots)

Dillon MacEwan 211285035 Supervisor: A. Prof Matthew Joordens

Batchelor of Mechatronics and Robotics Engineering

Deakin University

Faculty of Science, Engineering and Built Environment

School of Engineering

Submitted October 2016

Abstract

This thesis describes the development, design and fabrication of a hexapedal walking robotic platform in order to test the performance of a biologically inspired decentralised control system using distributed computing.

The project demonstrates that a distributed control system, based loosely on an ants coupled neural ganglia and drawing on the WalkNET system developed by Holk Cruse, can spontaneously generate variable but robust, adaptable and stable walking gaits analogous to those observed in insects, despite using only minimal signalling between individual legs and with each leg controller operating autonomously. Also, by treating the duty factor as a sliding scale, the different established insect gaits were seen to emerge as a continuum coupled to the gait velocity rather than discreet conditions.

Acknowledgments

First I would like to thank my Supervisor Associate Professor Matthew Joordens for his patience and support. I would also like to thank Trevor Tighe, who was the head engineer at The Creature Technology Company when I worked there and who has provided assistance and advice on various design issues, particularly in control engineering. I would also like to thank Owen Peake, hydraulic engineer with Arcadia International, who provided technical advice for large scale hydraulics, and Matt Denton, the builder and designer of the impressive Mantis hexapod (As well as BB8 from the recent Star Wars movie!), who took the time to discuss various technical and design issues relating to his project, including hydraulics, cost, control approaches and simulation.

I would especially like to thank my wonderful partner, Skadi, who has been tremendously supportive and patient, while also studying and working herself, and my kids, Connor and Tesla, who have all had to put up with my lack of spare time, and often sleep deprived states, piles of study papers and electronics where the dining table is supposedly hiding and a somewhat volatile cash flow!

Contents

i. Table of Figures 6

ii. Tables 7

iii. Table of Graphs 8

1 Introduction 9

1.1 Project Definition 10

2 Literature Review 11

2.1 Mechanical Design 11

2.2 Power 11

2.3 Control 12

2.3.1 Biomimetic Inspired Control Systems 12

2.3.2 Inverse Kinematics and Central Pattern Generation (CPG) 15

2.3.3 Artificial Neural Nets 15

2.3.4 Simulation Vs Physical Platform 15

3 Methodology 17

3.1 Design Criteria & Challenges 17

3.2 A Multi Streamed Approach 17

3.3 Mechanical Development 17

3.4 Control and Software 17

3.5 Sensors 17

3.6 Physical Testing 17

4 Physical Design, Kinematics and Kinetics 18

4.1 Scale And Actuation 18

4.2 Solidworks Modelling 18

4.3 Kinematics 18

4.3.1 Workspace Analysis 18

4.3.2 Inverse Kinematics 19

4.3.3 AEP/PEP 20

4.3.4 Leg Stroke Trajectory 20

4.3.5 Leg Swing Trajectory 20

4.3.6 Duty Factor 21

4.3.7 Rotational Motion 21

4.3.8 Ackerman Steering 22

4.4 Torque Analysis 23

4.4.1 Approximated Body Mass 23

4.4.2 Leg Joint Torques 23

4.4.3 Autodesk Force Effect FBD 26
5 Simulation 27
5.1 Simulation Software 27
5.1.1 Microsoft Robotic Development Suite 27
5.1.2 Morse 27
5.1.3 Gazebo 27
5.1.4 Webots 27
5.2 V-Rep 28
5.2.1 Leg Torques and Foot Position 29
6 Electromechanical Design 31
6.1 Servo Motors 31
6.1.1 Calibration: 31
6.2 Microprocessors 32
6.2.1 Arduino Nano 32
6.2.2 Arduino Mega 33
6.2.3 Raspberry Pi 3 33
6.2.4 Other Microprocessors 34
6.3 Sensors 35
6.3.1 Force Sensor Methods: 35
6.3.2 Bump Sensor 36
6.3.3 Other sensors. 36
6.4 Power System 36
7 Control Design 38
7.1 Control Design Methodology 38
7.2 Leg Controller 38
7.3 Control Network 38
8 Software Development: 40
8.1 PID 40
8.2 Inverse Kinematics 40
8.3 Mapping the Angle and posFoot() 41
8.4 AEP/PEP Swing/ Stride Cycle 41
8.5 Duty Factor 42
8.6 Swing Trajectory 42
8.7 Walk Cycle 42
8.8 Reflex Behaviour 42
8.8.1 Foot Feedback Reflexes 43

8.9 Direction and Speed Control 44
8.9.1 Rotating 44
8.9.2 Ackerman Steering 44
8.10 Further Software Development 45
8.10.1 Processors 46
8.10.2 Network Communications 46
9 Results 48
9.1 Hardware Issues 48
9.2 Running results 48
9.3 Observations 49
9.4 Further Investigation 49
9.4.1 Actuators 49
9.4.2 Increasing Sophistication 50
9.4.3 Motor Cortex 50
9.4.4 Hardware Improvements 50
9.4.5 Expandability and Redundancy 50
9.4.6 Cooperative Robotics 51
9.4.7 Scalability 51
9.4.8 Visual Cortex and Expanded Functionality 51
9.4.9 Proposed Applications 51
9.5 Conclusion 51
References 53
Appendices 56
Appendix I: Development Schedule 56
Appendix II: Control Diagrams 57
Appendix III: Analysis Diagrams 58
Appendix IV: CAD Drawing 61
Appendix V: Calibration Data 62
Appendix VI: Leg Control Program 66
Appendix VII: Calibration Programs 84
Appendix VIII: Force Effect Report 91
Appendix IX: Dynamixel Smart Servos 106

i. Table of Figures

Figure 1: Different leg configurations for Hexapod Walkers (Franco Tedeschi, Giuseppe Carbone 2014) 11
Figure 2: Schematic model of a stick insect leg showing the arrangement of the joints *(Cruse 1998)* 12
Figure 3: Common Insect Gaits (Wilson 1966) 13
Figure 4: WalkNet coordination Influences (Cruse 2013) 13
Figure 5: Leg Corrective Reflexes (Beer et al, 1997). 14
Figure 6: Cruse WalkNet Coordination influences with added bypasses to accommodate leg loss (Gerd and Hirzinger, 2010) 14
Figure 7: WalkNet with additional embedded mapping and navigation Nets (Hoinville et al., 2012) 14
Figure 8: Neural Net control circuit with CPG (Porcino, 1990) 15
Figure 9: HECTOR (Schneider et al, 2014) 16
Figure 10: Hexapod Frame Kit, purchased through AliExpress 18
Figure 11: Solidworks Model 18
Figure 12: Geometric Workspace Analysis 18
Figure 13: 3D Workspace Analysis 19
Figure 14: 2D Workspace Analysis 19
Figure 15: Leg IK analysis 19
Figure 16: : Common Insect Gaits (Wilson 1966) 21
Figure 17: Graphical analysis for Rotational Motion 22
Figure 18: Geometric analysis of Ackerman steering 22
Figure 19: Leg angles 24
Figure 20: Hexapod FBD 24
Figure 21: Proposed spring counter balance design 25
Figure 22: ForceEffect FBD 26
Figure 23: ForceEffect FBD with femur Torque 26
Figure 24: ForceEffect FBD extreme leg position 26
Figure 25: FBD analysis for servo <12kgf.cm 26
Figure 26: : Morse Screenshot (Open Robots 2016) 27
Figure 27: Webots Screenshot (Cyberbotics 2014) 28
Figure 28: Model editing in Meshlabs 28
Figure 29: Screenshot of V Rep Simulation with Leg Torque Output 29
Figure 30: TowerPro MG996R (electronicoscaldas.com 2016) 31
Figure 31: Position feedback modification 31
Figure 32: Physical Calibration Map 32
Figure 33: Physical Calibration 32
Figure 34: Arduino Nano (arduino.cc 2016) 32
Figure 35: Arduino Mega (arduino.cc 2016) 33
Figure 36: Raspberry Pi 3 (raspberrypi.org, 2016) 33
Figure 37: Odroid XU4 *(HardKernel.com 2016)* 34
Figure 38: STM32 Maple Mine (Leaf Labs 2016) 34
Figure 39: Lynxmotion FSR-01 (robotshop.com 2016) 35
Figure 40: Force and Bump foot sensors 36
Figure 41: HC SR04 ultrasonic range finder (robotshop.com 2016) 36
Figure 42: MPU 9150 (sparkfun.com 2016) 36
Figure 43: Mini 360 DC-DC Buck Converter Step Down Module (Ali Express 2016) 37
Figure 44: DC 15A Buck Adjustable 4-32V 12V to 1.2-32V 5V Converter (Ali Express 2016) 37
Figure 45: Leg Controller Diagram 38
Figure 46: WalkNet coordination Influences (Cruse 2013) 39

Figure 47: Control Network Diagram 39
Figure 48: IK Maths Function 40
Figure 49: Angle Mapping Function 41
Figure 50: posFoot() function 41
Figure 51: Function for retrieving AEP, PEP 41
Figure 52: IK trajectory following code 41
Figure 53: Elliptical path code 42
Figure 54: main walk() loop 42
Figure 55: Reflex influence code 43
Figure 56: stepUp() function 43
Figure 57: Leg Corrective Reflexes (Beer et al, 1997). 44
Figure 58: turn() function 44
Figure 59: ackerman() function 45
Figure 60: Assembled prototype control board 48
Figure 61: ArthroBOT, at the time of testing 48
Figure 62: Dynamixel RX 24 smart servo (Robotis 2016) 50
Figure 63: Schedule Gantt Chart 56
Figure 64: ArthroBOT Network Diagram 57
Figure 65: Leg Controller Block Diagram 57
Figure 66: Ackerman Steering Diagram 58
Figure 67: Hexapod FDB 59
Figure 68: Physical Calibration Map 59
Figure 69: Workspace Analysis 60
Figure 70: Hexapod CAD Drawing 61

ii. Tables

Table 1: Component Mass Estimates 23
Table 2: Electronic Servo Calibration 62
Table 3: Leg Data 62
Table 4: Leg Joint Data 62
Table 5: Servo Physical Callibration Data 64
Table 6: Leg Workspace Calibration Data 64
Table 7: FSR Calibration Data 64
Table 8: Dynamixel Specifications 107

iii. Table of Graphs

Graph 1: Geometric Workspace Analysis 19
Graph 2: Proposed Swing Paths 21
Graph 3: LF Foot Position and Velocity 29
Graph 4: LF Joint Torques 29
Graph 5: RM Foot Position and Velocity 30
Graph 6: RM Joint Torques 30
Graph 7: RM Coxia Angular Velocity 30
Graph 8: FSR Resistance/Load 35
Graph 9: FSR Resistance/Load average 35
Graph 10: Workspace Calibration 64
Graph 11: FSR Resistance/Load 65
Graph 12: FSR Resistance/Load Mean 65

1 Introduction

Wheels do not arise anywhere in nature, and much of the natural environment is not conducive to transit by wheeled vehicle. As such there has been a lot of interest over the previous decades in developing walking robots as an adaptable approach to navigation through variable and possibly hostile terrain. Some possible application for such platforms are; extra-planetary exploration, sea floor exploration, volcano exploration, disaster area search and rescue and subterranean research and exploration. The recent and ongoing situation at the Fukushima Nuclear Power Plant presents one situation where mobile robots are needed to navigate through a variable and damaged environment, too toxic for humans, and conduct technical operations. Arthropods offer an enticing avenue of research in the development of walking robotic platforms for a number of reasons. Arthropods represent some of the most successful and adaptable creatures on earth, able to inhabit any environment amenable to life, and make up 75% of all animals on earth (Australian Museum, 2016). They have had 530 million years of evolution to optimise their system design (Australian Museum, 2016), and yet present a much simpler system, both in regards to their biomechanics and neurobiology, than vertebrates. And as such, it has been observed that they seem to adopt a number of simple reflexive and reactive systems in order to overcome the problems of navigating variable terrain. As vertebrates would have also had to have overcome the same issues of legged locomotion in the natural environment, it is assumed that the lessons learned from the examination of arthropods will be equally applicable when designing more complex systems.

However, despite their relative simplicity, arthropods represent systems far more complex than anything currently possible in modern robotics today. Each insect leg contains thousands of specialised sensors, and a cockroach's antennae can have hundreds of thousands of sensors. Each insect leg can have between 5 and 8 DoF (Directions of Freedom), and spiders up to 10 DoF per leg, with additional joints between the body sections of some insects, leading to very complex mechanical system for control (Fichter and Fichter, 1988). Thoracic ganglia in insects contain thousands of neurons and the head ganglia is a complex and sophisticated system of sensory processing, memory banks and motor control, and insect muscles are more efficient than anything currently possible in modern engineering (Ritzmann, Quinn and Fischer 2004). Engineers also need to be aware that some scaling issues may not be applicable to the biomimetic approach when designing robots – certain aspects of insect biomechanics may be advantageous at the centimetre scale, but less applicable at the metre scale – this is certainly indicated by the study of animal biomechanics, when one observes the difference between the tetrapod gait of lower vertebrates such a salamanders and lizards – which has similarities to insect leg arrangements – and mammalian leg geometry, and the mechanical advantage it affords them(Ritzmann et al 2004). Scaling issues are also a factor when considering the materials and their respective properties in robot design. A direct facsimile of biological systems and mechanics may be desirable in a biological scientific enquiry, where one is trying to gain greater understanding of the workings of actual insects, but from an engineering perspective, where one seeks practical outcomes, a choice must be made as to which aspects from biomimetic research are useful, and which to discard.

Even the control of a hexapod, with each leg reduced to three DoF (Cruse 1976), results in a robot with at least 18 DoF operating in a non-Cartesian space resulting in a computationally intensive operation for a traditional centralised controller.

The biomimetic approach to walking machines, with particular attention on arthropods has been approached from a number of directions; from the ground breaking new approach to cognitive systems using subsumption architecture developed by Rodney Brooks at MIT (Brooks, 1986, 1989,1990, 1991) , to the influential work on artificial neurons and insect walking neurobiology by biological cyberneticist Holk Cruse, to the more engineering based approach by Randall Beer, Roger Quinn and their team over the years.

Invertebrate nervous systems consist of distributed clusters of neural ganglia, that are responsible for the articulation and control of separate segments of the body. For the legs, these appear as paired clusters, and it has been observed that the legs act largely autonomously from the brain, which only mediates navigation commands – indeed a cockroach will continue to walk even if it's cerebral ganglion is severed form the central nerve cord (Pearson 1976). This concept will provide the basis of the control system design explored here.

1.1 Project Definition

The objective of this proposed project is to develop, design and build a hexapedal walking robotic platform in order to test the performance of a biologically inspired decentralised control system.

The projects purpose is to demonstrate that a distributed control system, based loosely on an ants coupled neural ganglia and drawing on the WalkNET system developed by Holk Cruse, can spontaneously generate variable but robust, adaptable and stable walking gaits analogous to those observed in insects, despite using only minimal signalling between individual legs and with each leg controller operating autonomously. Also, by treating the duty factor as a sliding scale, the different established insect gaits should emerge as a continuum coupled to the gait velocity rather than discreet conditions.

2 Literature Review

There are a number of reasons motivating research into autonomous walking platforms, as well as numerous pitfalls and challenges that have made progress in this area slow. There is also considerable rationale in looking to nature for inspiration and insight into the design and control of walking robots, as evolution can be the best optimisation tool, as arthropods have had hundreds of millennia to develop the best evolutionary advantages through mechanical and energy efficiencies and through versatility *(Beer, R.D., Chiel, H.J., Quinn, R.D., Ritzmann R E , 1997)*.

Due to the complexity of the task, the designing of a hexapod platform involves a range of trade-offs between design requirements, functionalities, cost restrictions and component specifications *(Franco Tedeschi, Giuseppe Carbone 2014)*.

2.1 Mechanical Design

Mechanically it is worth doing a comparative study between arthropod leg configurations *(Wilson D 1966)*, leg geometries used in other hexapods and the forces acting in similar mechanical systems such as articulated crane arms or backhoes to further understand the strengths and weaknesses of these various configurations in order to come up with more optimal designs for specific requirements *(Kenzo Nonami, Ranjit Kumar Barei, Addie Irawan, Mohd Razali Daud, 2014)*. Factors to consider are the distribution of forces within the legs, the optimal configurations for speed, power efficiency, pose variability, stability and obstacle avoidance. For example a lower centre of mass will increase stability, but decrease the ease with which the robot can traverse obstacle strewn terrain, similarly, a wider span and spread on the hexapods legs may increase stability, maximum achievable velocity and the ability to traverse higher obstacles and wider ditches, but will decrease energy efficiency and increase the stresses in the leg components *(Franco Tedeschi, Giuseppe Carbone 2014,Nonami et al 2014)*. Another factor to be examined is what advantage, if any, is to be gained from a more biologically inspired non-homogeneous approach to the design of the separate leg pairs – i.e. – larger rear leg pairs with stronger actuators for thrust, and lighter more agile front leg pairs for reflexive obstacle detection and mitigation. Also whether there is an advantage to be had through a more radial arraignment of shoulder joints as opposed to a strictly rectilinear one, in regards to both stability and manoeuvrability *(Ritzmann, R.E., Quinn, R.D., Watson, J.T., Zill, S.N., 2000)*.

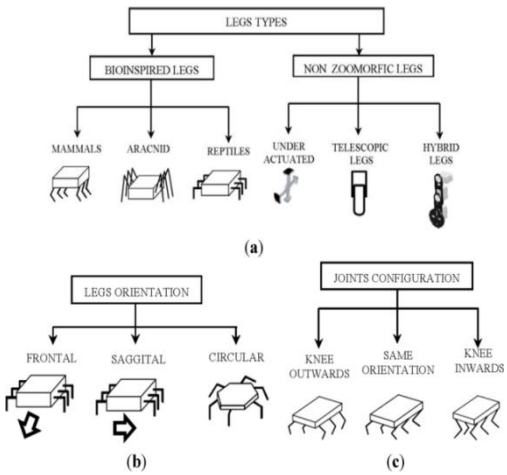


Figure 1: Different leg configurations for Hexapod Walkers (Franco Tedeschi, Giuseppe Carbone 2014)

2.2 Power

There are a number of power and actuation options that can be used in the design of walking robots. Servo motors are the most commonly used in smaller scale robotic platforms, as they are easily controlled, commonly available and relatively cost effective. With considerable improvements to battery technology in recent years, on-board energy storage or generation is becoming less of an issue. Linear motors offer the ability to provide fast reaction times and applied forces, but have not been used so widely in the area of walking robotics, possibly due less availability and greater cost when compared to servo motors *(Franco Tedeschi, Giuseppe Carbone 2014)*. Pneumatic cylinders can be cost effective, but lack stiffness and accuracy due to the compressibility of air, have a relatively low power to weight ratio, and have the additional drawback of having to carry an on board compressed air reservoir, and possibly compressor with additional power requirements. Air muscles or McKibben muscles offer superior power to weight ratio and reaction time, but still have the same controllability issues and power requirements *(Franco Tedeschi, Giuseppe Carbone 2014)*.. Hydraulics offer the best

power to weight ratio, and good accuracy and controllability, and a favoured for larger machines, but the components can be heavy due to the high oil pressures required, the control of hydraulics is more complex than servo motors, and for independent mobility the systems require that the platform carry the additional weight of a hydraulic power pack – consisting of a pump and motor – typically internal combustion for larger systems. Considering this, there will be a lower size and weight threshold at which hydraulics will not be a practical option – it is a peripheral objective of this project to examine this threshold *(Franco Tedeschi, Giuseppe Carbone 2014,Nonami et al 2014)*..

2.3 Control

The control aspect of the project is its main focus, and as such has been the main area of focus for the literature review.

Hexapods have an advantage over quadrupeds and bipedal robots, as they are intrinsically statically stable, and therefor require less continuous processing that the platforms with less limbs require for dynamic stability, but have the disadvantage of having to control a greater number of actuators simultaneously *(Franco Tedeschi, Giuseppe Carbone 2014,Nonami et al 2014)*.. This can be computationally demanding for a centralised controller, so a number of strategies to cope with this issue have been employed by various researchers over the years. Most commonly a decentralised networked control strategy has been employed, with individual leg behaviour and motion determined by feed forward control from the environment and the states of adjacent legs, and feedback from some of the joint positions – these systems are designed to be reactive and to operate in real time *(Beer et al 1996, 1997, Cruse et al 1995, 1998, 2006)* . Some researchers have attempted to build hexapod navigation systems that are wholly reactive and decentralised *(R A Brooks, 1986, 1989, Lewinger and Quinn 2009)*, , while others have opted for maintaining a centralised processor for higher functions such as task planning and as a central pattern generator *(Paolo Arena,, Luigi Fortuna, Mattia Frasca, and Giovanni Sicurella 2004)*, as well as using more computationally demanding processes such as inverse kinematics *(Neslon and Quinn, 2001)*. The latter requires the design to have an increased payload capacity to accommodate the larger processing and associated power requirements.

2.3.1 Biomimetic Inspired Control Systems

2.3.1.1 Genghis

Rodney Brookes first proposed his Subsumption Architecture as an alternative to the at the time traditional approach to AI, which was a top down, centralised approach relying on symbolic world models. Brooks instead took his inspiration from simple animals, viewing how they could successfully survive and navigate in complex and hostile environments, presumably without any higher cognitive functions or world model. From this he developed a bottom up approach to behaviours, driven by sensory input from the immediate environment, and consisting of Augmented Finite State Machines which affect each-others operation not through direct communication but through suppression or inhibition signals. This system was demonstrated with the Hexapod Genghis, and it proved that quite complex and adaptive behaviours could arise from fairly simple sets of rules.

2.3.1.2 WalkNet

Holk Cruse, who has a background in biological cybernetics, and had done extensive research on the locomotion and reflexes of stick insects (Carausius morosus). Cruse had previously identified that while most insects had more DoF per leg, three DoF were enough to describe most of an insects locomotion (Cruse 1976).

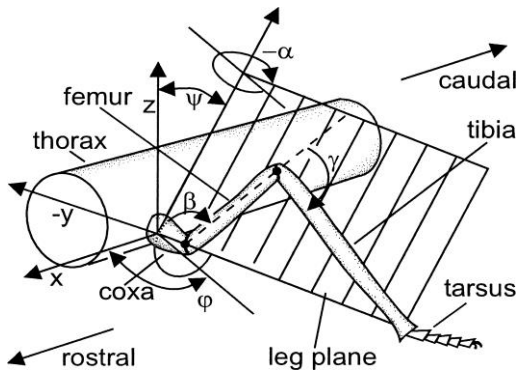


Figure 2: Schematic model of a stick insect leg showing the arrangement of the joints *(Cruse 1998)*

Wilson, Cruse and others have also identified a number of stereotypical gaits for walking insects; pentapedal gait at slow speeds (slow wave gait - only one leg in the air at any time) tetrapedal gait (ripple gait - four legs in contact with the ground at all times) and tripod gait at higher speeds. Two distinct phases in walking motion for each leg are also identified; stance, where the leg is supporting body weight, and swing, where the leg is moving forward through the air to its forward most step position.

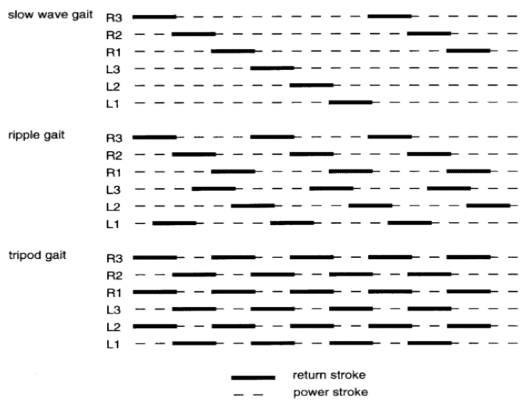


Figure 3: Common Insect Gaits (Wilson 1966)

Cruse took Brooks' subsumption architecture as the basis for developing a basic neural network representative of a reactive control system for walking insects. The network resembles an advanced state machine, with feed-forward controls and suppression/inhibition signals communicating between the different legs, but with no centralised control mechanism. Important control and sensor information for the system included the Anterior and Posterior Extreme Positions for the feet (AEP and PEP), leg joint angular position, relative body height, force feedback on the tarsus, and swing and stance angular velocity. Stability is maintained through a cardinal rule that a leg may only go into swing mode if none of its immediate neighbours are in swing mode. This has the notable affect that as the velocity of the insect increases, and the swing/stance period decreases, the insect gait converges from tetrapedal to tripod.

The other coordination influences identified by Cruse are illustrated in Figure 4. The coordination influences can also be scaled for walking systems greater than 6 legs. Posture – relating to the insect or robots body height, position and orientation, is controlled through a feedback loop from the angular position of the second joint in each leg. These observations and models were combined in a structural program of recurrent neural nets (RNN) called WalkNet. This system and model has been the basis for the majority of hexapod and octapod walking robot control systems subsequently. Even if the entire control architecture has not been used, at the very least the decentralised, feed forward coordination influence model has been used in the majority of multipedal walker control systems.

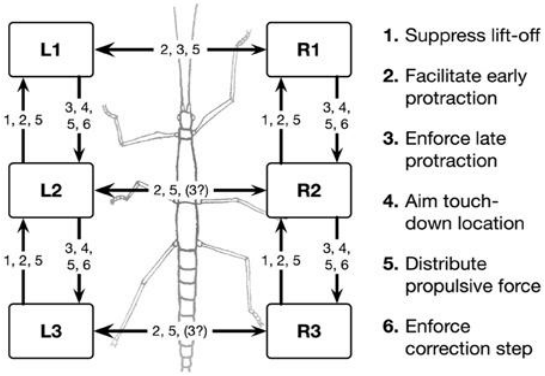


Figure 4: WalkNet coordination Influences (Cruse 2013)

2.3.1.3 WalkNet – A Continued Evolution

The original WalkNet, built and tested in simulation only, had many shortfallings, much of which have subsequently been identified and remedied both by Cruse and by other researchers working with this architecture and applying it practically with a number of walking robotic platforms. Randall Beer, Roger Quinn and their team developed a number of additional functions in order to deal with obstacles or loss of footing in their robots, ROBOT I and ROBOT II – these included a searching reflex when gaps were encountered, an elevator reflex when taller obstacles were encountered, and a stepping reflex for correcting foot slippage (Beer et al, 1997). It should be noted that at the time Beer and their team were developing a similar Neural Network control algorithm to WalkNet, and that these reflexes were also later added to WalkNet and used in the TUM walking machine, MAX (Pfeiffer et al 1994)

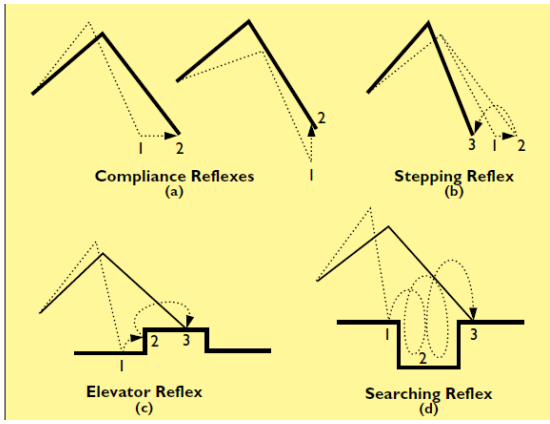


Figure 5: Leg Corrective Reflexes (Beer et al, 1997).

Wait and Goldfarb provided further fine tuning in identifying problems that WalkNet had with real world dynamical situations involving gravity and stability (Wait and Goldfarb 2007) while Martin Gorner and Gerd Hirzinger (2010) and Schilling, Cruse and Arena (2006) have both independently added leg loss tolerance functionality to WalkNet – Gorner and Hirzinger building on Schilling et al earlier work.

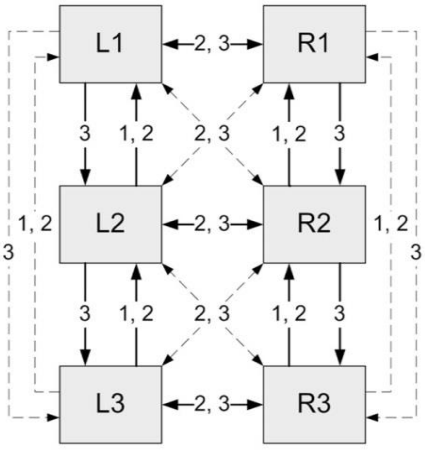


Figure 6: Cruse WalkNet Coordination influences with added bypasses to accommodate leg loss (Gerd and Hirzinger, 2010)

Gorner and Hirzinger also incorporate an omnidirectional navigation function in the application of WalkNet to their DLR Crawler, by defining the AEP and PEP as radii on a circle rather than points on a line parallel to the walker's body.

Higher functions have been also added to WalkNet with the addition of more RNN structures on a higher hierarchical level than the embedded control structure that WalkNet represents. NaviNet provides navigation functionality by influencing steering towards stored waypoints (Cruse and Wehner, 2011; Hoinville et al., 2012), and additional RNN's which model individual legs coupled to an RNN to represent the body in order to provide an internal body model, which aids error compensation during locomotion, and Motivational Network to further influence suppression and inhibition functions, as well as additional procedures for backwards locomotion and turning (Schilling et al 2013).

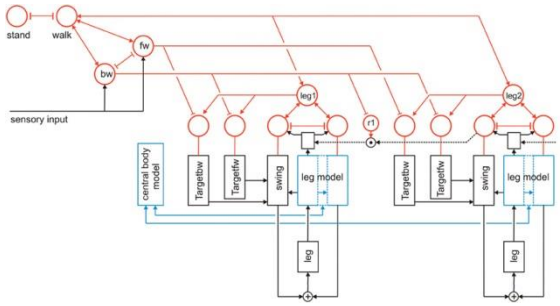


Figure 7: WalkNet with additional embedded mapping and navigation Nets (Hoinville et al., 2012)

2.3.1.4 SCASM and BILL Ant A

Another approach to a neural net based, decentralised sensor driven embedded control system for leg movement is Sensor Coupled Action Switching Leg Modules (SCASM) (Lewinger et al 2006). SCASM presents a further simplified reactive leg control mechanism, which has been used in conjunction with aspects of WalkNet still providing higher functionality such as gait control and navigation functions, and has been demonstrated to be effective on the BILL Ant A platform using low computational capable microcontrollers (Lewinger et al 2009). Further investigations into computationally economical methods for gait generation and hazard and obstacle avoidance are also being developed. This approach then demonstrates a further streamlining of computational demands for small independent systems which have a smaller energy and payload budget than larger platforms. It is unclear, however, whether this simplified system will scale for more complex or larger robotic walking platforms.

2.3.2 Inverse Kinematics and Central Pattern Generation (CPG)

So far this investigation has focussed primarily on WalkNet and similar systems, which seek computational economy by using embedded reactive systems in the form of RNN's, doing away, as much as possible, with any central processing and control. However an alternative and much employed approach to the control of walking robots is the use of Central Pattern Generators (CPG) and Inverse Kinematics with some sort of decentralised control network such as Cellular Nonlinear Networks (CNN). Central Pattern Generators are a centralised oscillator mechanism used to generate rhythmic patterned outputs with no sensory feedback. They are known to be present in biological neural networks, and are used in the context being discussed to provide the rhythmic signal to control the walkers walking gait. Inverse kinematics is a common concept in robotics for the control of non-orthogonal robotic arms and manipulators. It refers to the use of kinematic equations to determine the ideal angular position of the joints of a robot arm (or leg) when the desired location of the robots end effector (or foot) us known. Inverse kinematics can be problematic with the multiple DoF legs used in insect based walkers, due to redundancy, but a number of solutions have been identified (Nelson and Quinn 2001). CNN's or similar are modular networks that implement the CPG signal for locomotion and leg control. This architecture has a number of advantages as it has proven to perform well, and to be robust and reliable, and more versatile when it comes to implementing omnidirectional movement than the WalkNet based systems have tended to. They can also still employ some of Cruse's coordination interfaces at a direct world interface level, in order to maintain stability and gain environmental feedback (Cruse, 2003, 2006, Palmer III and Palunkar 2007, Arena et al 2004, Roennau et al 2010) .

In this project, Inverse Kinematics is the most likely method that will be used for foot trajectory control, but it is the intention of the thesis to demonstrate that the computational complexity is greatly reduced when separate foot controllers handle the IK for each foot independently. Also it is hoped that the project will demonstrate an emergent stale gait generation without the need for a Central Pattern Generator.

2.3.3 Artificial Neural Nets

Artificial neural nets (ANN), and the subset already discussed, RNN, feature frequently in the literature. It is frequently applied to both leg trajectory control and gait generation and is often utilized in research projects with a biomimetic focus and ones which favour a decentralised architecture. They are often applied in conjunction with genetic algorithms. (Porcino, 1990, Beer et al 1992, Silva 2014, Parker and Li, 2003).

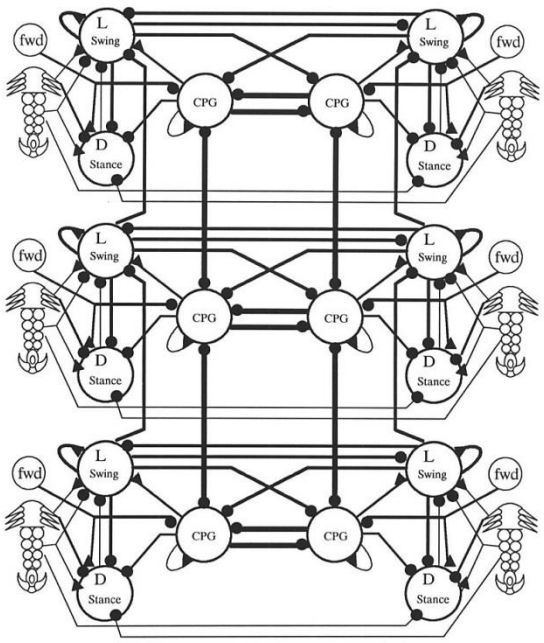


Figure 8: Neural Net control circuit with CPG (Porcino, 1990)

2.3.4 Simulation Vs Physical Platform

Of the research papers exploring the various approaches to the control of mobile walking hexapod robots, a majority of them rely on simulation only as their methodology. Due to the cost and complexity involved in building a fully articulated hexapod this comes as no surprise (Porcino 1990, Rosano-Matchain 2007, De Silva 2014). Of the existing physical platforms, the larger models tend to use more traditional centralised control methods rather than the biomimetic approach discussed here (Mantis 2016, Nonami et

al 2014). Of the functional experimental platforms that take a similar approach to this project, the hexapod HECTOR from the research team at Beilefeld University in Germany is one of the most sophisticated, both mechanically and in control approach (Schneider et al, 2014).

Figure 9: HECTOR (Schneider et al, 2014)

3 Methodology

In order to properly develop and test the proposed control system, a functional physical test platform is proposed. This is considered necessary as a simulation might not give a comprehensive demonstration of the true performance capabilities of the control system when interfacing with physical linkages and actuators, and variable environmental inputs.

3.1 Design Criteria & Challenges

The platform should be of a reasonable scale in order to make observations that may be useful to applying the control system to larger platforms, with each leg having 3DOF, the minimum to allow full articulation. Care has to be taken I sourcing the most cost effective devices and components that are still adequate to meet the criteria of the project, as it has to be achieved on a limited budget, preferably within $500.

One of the biggest challenges will be constructing a fully functioning platform while also developing the control system, electro-mechanical system and software leaving enough time to comprehensively test its performance, with only 12 weeks from the projects commencement to its presentation.

3.2 A Multi Streamed Approach

To achieve these outcomes, the design process will occur concurrent with ordering the necessary components and material and the first stages of the software development. The focus will be on designing, building, developing and testing the leg controller network. In part the aim is to demonstrate the emergence of stable and regular gaits without the need for a central controller, so the inclusion of one is not necessary for this this investigation, and the thesis is better demonstrated without one. However as the leg control network is intended as the first level of a multi-tier control network, the design and interface for these higher level controllers will also be researched, and if time permits, implemented.

3.3 Mechanical Development

For the physical development stage, the hexapod will be modelled in Solidworks, then imported for simulation in V-Rep, with the intention of being able to test code sequences and simulate physical situations whole the electronics for the physical controller is being developed tested and assembled.

3.4 Control and Software

The control system will be developed through analysis of the literature on WalkNet and similar systems and through kinematic analysis of the hexapod legs and desired trajectories. The resultant control network diagram will then inform the development of both the control network circuit and the control software.

3.5 Sensors

As well as basic leg trajectory control, feedback sensors on the legs to provide environmental information to the controller such as detecting leg load and small obstacles will be implemented I order to make the legs more reactive and improve real world performance. To meet the schedule requirements the various development aspects of the project will have to occur concurrently throughout the development.

3.6 Physical Testing

Development and testing of the individual leg trajectory control software can occur as soon as the hexapod body structure and legs are assembled, and servomotors tested and calibrated, and is not dependent on the control network circuitry being fabricated. Full testing of the control network however is dependent on the fabrication of a working prototype board, as with 18 servos, 36 sensors and 6 microcontrollers, the circuit will be too complex for safe breadboard prototyping. It is intended that this stage of the project will occur around week 9 – 10, to allow a reasonable period of testing, tuning and debugging before presentation.

A proposed production schedule Gantt Chart can be found in Appendix I: Development Schedule.

4 Physical Design, Kinematics and Kinetics

4.1 Scale And Actuation

A range of options were initially considered when designing the test platform for this project. Some of the larger existing walking platforms were investigated for feasibility for this project, such as the Comet IV hexapod project (Nonami et al 2014), and the Mantis Walking Machine (Mantis, 2016). After some initial investigation, it was clear that the scale, cost and complexity of such a platform was completely impractical and unfeasible for the parameters of this project. As an indication of the scale of such an endeavour, it would probably have weighed 1 – 4 Tonnes, had material costs of around $80,000 and required considerable time and resources to realise.

As the aim of the project was to investigate the practical implementation of a biologically inspired control system, which should then be both scalable and expandable, and given there is already considerable complexity in building a functioning platform of any scale with 6 legs, at least 6 microcontrollers and 18 DOF as well as numerous sensors, the most expedient and cost effective approach was to base the structure of the platform off an existing, readily available structure (Figure 10).

Figure 10: Hexapod Frame Kit, purchased through AliExpress

4.2 Solidworks Modelling

Once the body type and design was decided on, a 3d model of the hexapod was built using the 3D CAD software, Solidworks (Figure 11). The Solidworks model was then used for further analysis and simulation. The Solidworks CAD drawing can be found in Appendix IV.

Figure 11: Solidworks Model

4.3 Kinematics

4.3.1 Workspace Analysis

A geometric analysis of the proposed hexapod's workspace was conducted using Solidworks (Figure 12). For analysis, the limits for angle between the tibia and the ground were set at 80° away from the body when at its inside extreme, and 45° towards the body at the legs full stretched extreme; The range for the angle between the femur and the tibia was set at 15° - 135°; and the range for the tibia angle was set at 40° - 170° (where 180° would be a vertical up position. The anterior and posterior extreme angle position was determined such that leg collision would be avoided in all positions – this gave a range of 60° for the centre pair of legs and 50° each for the front and rear legs (Graph 1).

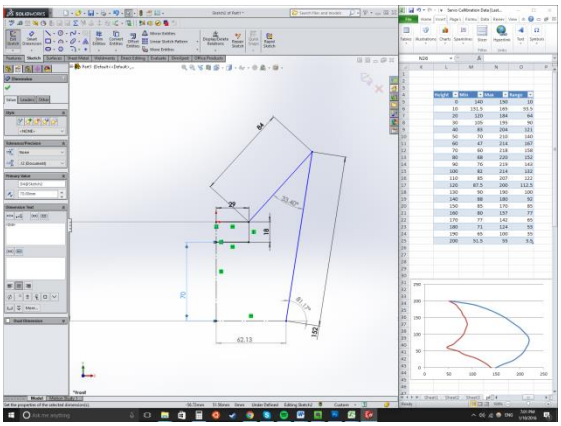
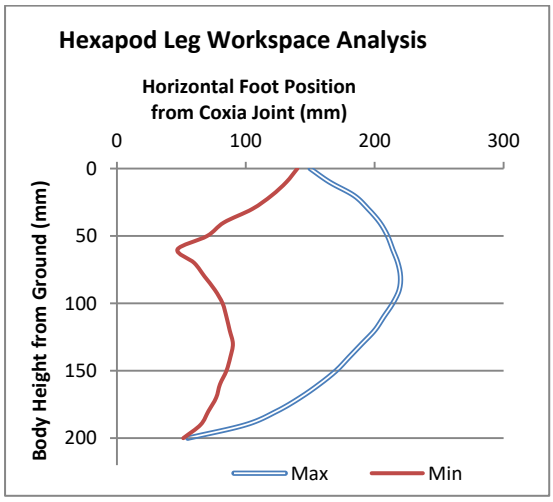


Figure 12: Geometric Workspace Analysis

The geometric analysis data was then used to 3D model the legs workspaces for all legs in Solidworks (Figure 13: 3D Workspace Analysis) and a footprint slices taken from this model in order to determine an optimum operation height for the hexapod (Figure 14).



Graph 1: Geometric Workspace Analysis

An inflection point was noted at a body height of around 60mm, where the overriding limit transitioned from the Femur/Tibia angle, to the Tibia/Ground angle.

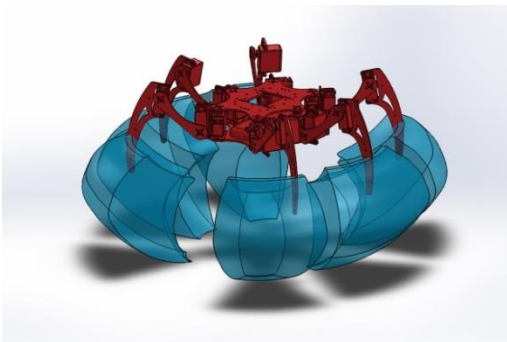


Figure 13: 3D Workspace Analysis

The optimum operations height was chosen in order to maximise the operational footprint while avoiding the inflection point – the foot should also have enough z range in order to be able to comfortably adapt to variations in the ground topography – the body height chosen was 92mm from the foot z position to the underside of the body base. An ideal neutral foot position was then determined in order to maximise leg stroke length both in a forward motion and for omnidirectional motion. Calibration data can be found in Appendix V: Calibration Data.

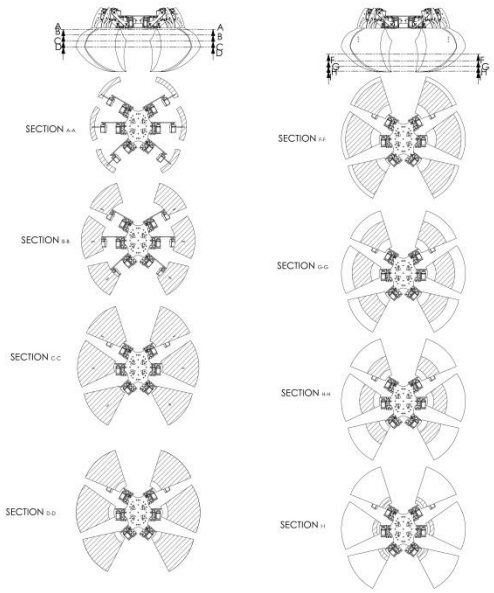


Figure 14: 2D Workspace Analysis

4.3.2 Inverse Kinematics

Calculating the Inverse Kinematics formula for an individual leg was a trivial process, as with only 3DOF on each leg, and necessary range limits on each joint, any chosen Cartesian foot point has only one possible solution for its corresponding leg joint positions (Figure 15).

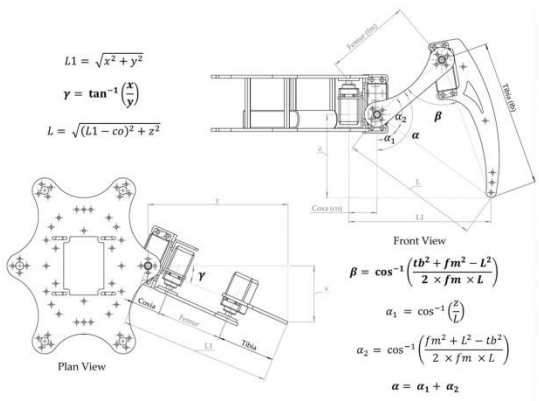


Figure 15: Leg IK analysis

$$L1 = \sqrt{x^2 + y^2} \qquad \text{Eq 1}$$

$$\gamma = \tan^{-1}\left(\frac{x}{y}\right) \qquad \text{Eq 2}$$

$$L = \sqrt{(L1 - co)^2 + z^2}$$ Eq 3

$$\boldsymbol{\beta = \cos^{-1}\left(\frac{tb^2 + fm^2 - L^2}{2 \times fm \times L}\right)}$$ Eq 4

$$\alpha_1 = \cos^{-1}\left(\frac{Z}{L}\right)$$ Eq 5

$$\alpha_2 = \cos^{-1}\left(\frac{fm^2 + L^2 - tb^2}{2 \times fm \times L}\right)$$ Eq 6

$$\boldsymbol{\alpha = \alpha_1 + \alpha_2}$$ Eq 7

The convention chosen for all IK calculations were that z is the vertical distance of the foot from the femur joint – which is 18mm more than the vertical distance of the foot to the underside of the body base – x is the distance from the coxia joint along the forward axis of the body – x positions behind the coxia joint have a negative value – and y is the distance from the coxia joint on the lateral axis of the body. The y axis is flipped on opposite sides of the hexapod body. All angles are in degrees.

4.3.3 AEP/PEP

Once a neutral foot position coordinate (np[X,Y,Z]) for each leg was determined, formulae for other leg trajectory functions are easily derived.

To calculate the foot Anterior and Posterior Extreme Position (AEP, PEP) for any given trajectory, the angle of travel(θ) and the desired stride length(l_s) are also required:

$$AEP[X] = np[X] + \frac{1}{2} l_s \cos(\theta)$$ Eq 8

$$AEP[Y] = np[Y] + \frac{1}{2} l_s \sin(\theta)$$ Eq 9

$$PEP[X] = np[X] - \frac{1}{2} l_s \cos(\theta)$$ Eq 10

$$PEP[Y] = np[Y] - \frac{1}{2} l_s \sin(\theta)$$ Eq 11

$$AEP[Z] = PEP[Z] = np[Z]$$ Eq 12

4.3.4 Leg Stroke Trajectory

For smooth leg motion, and in order to minimise conflicting torques developing in opposing leg joints, it is important that the leg stride trajectory tracks the hexapods line of motion with reasonable accuracy. One method that is outside the scope of this project, but could be worth considering for future application is using force feedback with torque sensors in each leg joint – this method would be more analogous with how an animal might operate and in keeping with the biomimetic approach of this investigation, and would have the advantage of mitigating errors in the software or variation in the terrain topography by using physical feedback inputs, but for the purpose of this project the foot trajectory is determined primarily using IK path planning.

To maintain a straight line trajectory, a number of incremental steps is decided on to ensure both a reasonable velocity of stroke, and a smooth motion – this is then used, together with the AEP and PEP coordinates, to determine the Δx, Δy and Δz for each incremental waypoint along the foot trajectory:

$$\Delta x = \frac{l_s \sin(\theta)}{steps}$$ Eq 13

$$\Delta y = \frac{l_s \cos(\theta)}{steps}$$ Eq 14

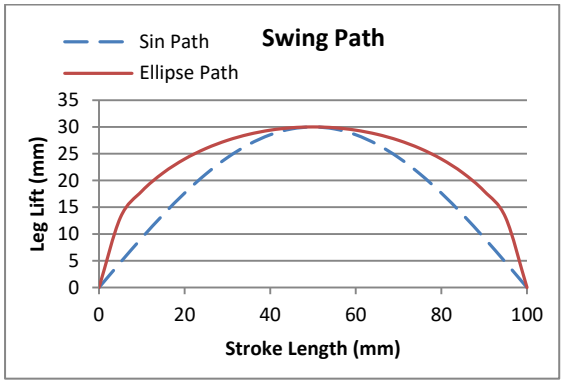
Barring changes necessitated by the foot force feedback, z should remain constant during the stroke phase, or trend towards np[Z].

4.3.5 Leg Swing Trajectory

In determining the leg swing trajectory, a continuous arced motion was desirable rather than a discrete stepped one. Two methods were considered for the swing motion – sinusoidal motion and elliptical motion. Elliptical swing motion was favoured as it demonstrated a more abrupt Δz at the start and end of the motion curve, reducing the possibility of foot drag or premature ground collision (Graph 1).

In these methods, Δz is an absolute difference from np[Z] rather than an incremental change, and the angle for the trig function is in radians rather than

degrees. h_l designates the maximum desired step height.



Graph 2: Proposed Swing Paths

Sinusoidal Motion:

$$\Delta z = h_l \sin\left(\pi \frac{currentStep}{steps}\right) \qquad \text{Eq 15}$$

Elliptical Motion:

$$\Delta z = h_l \sqrt{1 - \left(1 - \frac{2 \times currentStep}{steps}\right)^2} \qquad \text{Eq 16}$$

4.3.6 Duty Factor

Duty Factor (β) determines the proportion of the leg cycle period (T) that the leg is in a stance, or support state (t_s) compared to its swing, or return state(t_r):

$$t_s = \beta T \quad t_r = (1 - \beta)T \qquad \text{Eq 17}$$

Katsuhiko Inagaki also demonstrates that duty factor determines the average number of supporting legs for any given value of β (where n is the number of legs)*(Inagaki, 2009)*:

$$n_{ave} = \beta \cdot n_{tot} \qquad \text{Eq 18}$$

For static stability in a hexapod, at least 3 legs must be on the ground, therefore the operational duty factor range for a hexapod is $0.5 \leq \beta < 1$. It is also apparent that for a Pentapedal gait β must necessarily be 0.833. The duty factor, then, controls the gait of the hexapod, as a continuum of values from a slow pentapedal gait to the fastest tripod gait.

From analysis of the recognised gait patterns in insects, the swing phase on a given side always proceeds sequentially, hind to fore, regardless of duty factor, however the lateral delay between swing phases can clearly be seen as a function of duty factor (Figure 16).

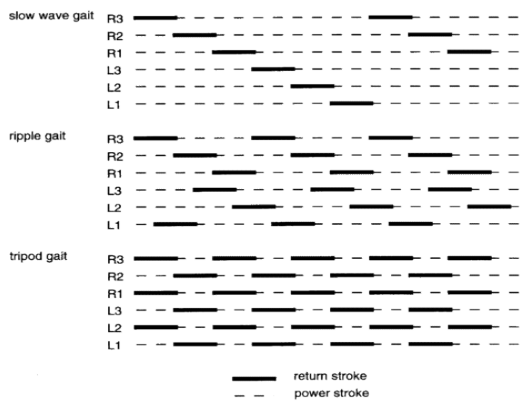


Figure 16: : Common Insect Gaits (Wilson 1966)

The delay between the start of the swing phase between corresponding pairs of legs is 0.5T The delay therefor between one leg landing and it's lateral partner returning is:

$$t_{ld} = (\beta - 0.5)T \qquad \text{Eq 19}$$

Finally, as the limiting factor for leg velocity is the reaction time of the joint actuators, the swing or return phase can be considered as constant, set at the shortest period it takes the leg to complete most likely swing trajectories, with the velocity set by the time each leg takes to complete the stride trajectory. The velocity of the hexapod therefor can also be controlled by varying the duty factor (l_s is stride length).

$$v = \frac{2l_s}{\beta T} \qquad \text{Eq 20}$$

4.3.7 Rotational Motion

The desired angle of trajectory for each foot motion for rotational motion was worked out graphically using Solidworks (Figure 17). It was determined that an travel angle of 48° on the front right and rear left legs, and -48° on the front left and rear right legs would generate the smoothest rotational motion, with opposing sides travelling in opposite directions, determined by whether left or right rotation was desired.

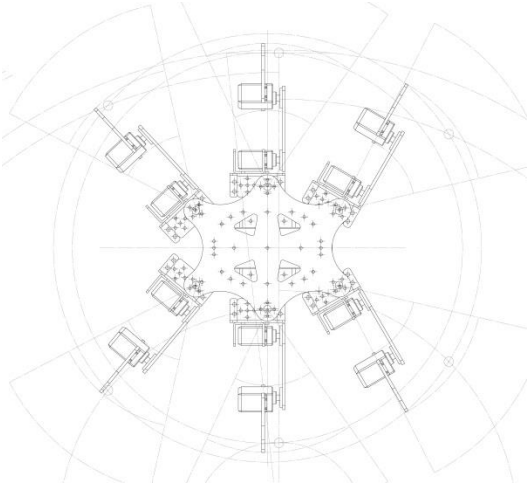


Figure 17: Graphical analysis for Rotational Motion

4.3.8 Ackerman Steering

Implementing Ackerman steering on a hexapod poses particular challenges, as each legs neutral position is located at a different radius from the centre of the turning circle, with the fore and rear legs offset from the radius of the line of travel. Also as all legs are driving, an angle of trajectory and a length of stroke has to be calculated for each separate leg for any given radius.

The governing equations for Ackerman steering were derived from geometric analysis, using the established foot neutral x and y coordinates, and setting the outside middle leg stroke at L (Figure 18).

R	Radius of trajectory to outside centre foot
rI	Radius of trajectory to inside fore foot
r2	Radius of trajectory to inside rear foot
r3	Radius of trajectory to outside fore foot
r4	Radius of trajectory to outside rear foot
A	Stroke Length of inside centre foot
B	Stroke Length of inside rear foot
C	Stroke Length of inside fore foot
D	Stroke Length of outside fore foot
E	Stroke Length of outside rear foot
θ	½ the arc angle of outside centre foot
α	Angle of travel inside fore foot
β	Angle of travel inside rear foot
γ	Angle of travel outside fore foot
δ	Angle of travel outside rear foot

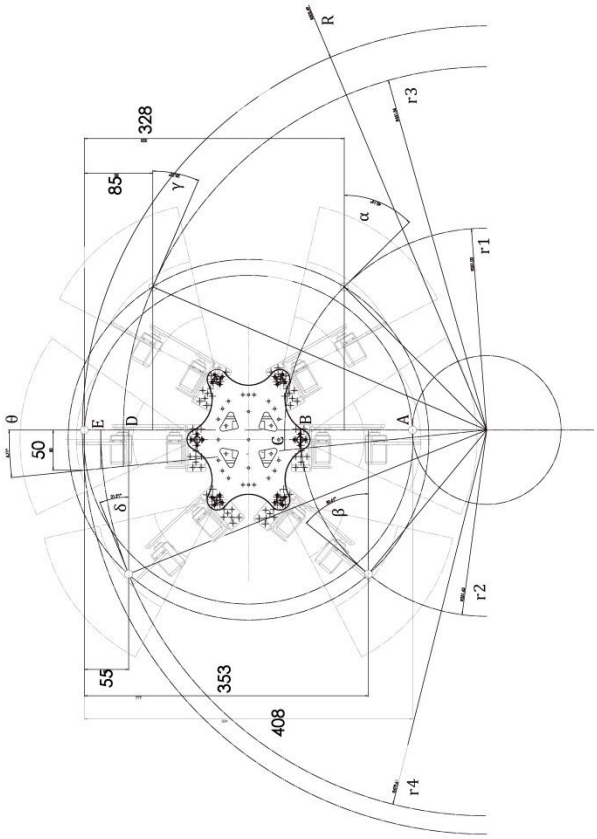


Figure 18: Geometric analysis of Ackerman steering

$$\theta = \tan^{-1}\left(\frac{L}{2R}\right) \qquad \text{Eq 21}$$

$$\frac{L}{R} = \frac{A}{R-408} \therefore A = L\left(1 - \frac{408}{R}\right) \qquad \text{Eq 22}$$

Front and rear neutral points are 178 mm from the centre line of the centre leg neutral points.

Inside Fore

$$r1 = \sqrt{178^2 + (R-328)^2} \qquad \text{Eq 23}$$

$$C = \frac{Lr1}{R} \qquad \text{Eq 24}$$

$$\alpha = \tan^{-1}\left(\frac{178}{R-328}\right) \qquad \text{Eq 25}$$

Inside Rear

$$r2 = \sqrt{178^2 + (R - 353)^2} \quad \text{Eq 26}$$

$$B = \frac{Lr2}{R} \quad \text{Eq 27}$$

$$\beta = \tan^{-1}\left(\frac{178}{R - 353}\right) \quad \text{Eq 28}$$

Outside Fore

$$r3 = \sqrt{178^2 + (R - 85)^2} \quad \text{Eq 29}$$

$$D = \frac{Lr3}{R} \quad \text{Eq 30}$$

$$\gamma = \tan^{-1}\left(\frac{178}{R - 85}\right) \quad \text{Eq 31}$$

Outside Rear

$$r4 = \sqrt{178^2 + (R - 55)^2} \quad \text{Eq 32}$$

$$E = \frac{Lr4}{R} \quad \text{Eq 33}$$

$$\delta = \tan^{-1}\left(\frac{178}{R - 55}\right) \quad \text{Eq 34}$$

These can be reduced to:

$$r = \sqrt{178^2 + (R - b)^2} \quad \text{Eq 35}$$

$$Stroke = \frac{Lr}{R} \quad \text{Eq 36}$$

$$travel\ angle = \tan^{-1}\left(\frac{178}{R - b}\right) \quad \text{Eq 37}$$

Where b is the perpendicular offset of the foot neutral position from the outside centre foot neutral position.

With the formulae derived in this kinematic analysis, it should be possible to derive leg motion algorithms capable of executing smooth forward and reverse motions, omnidirectional travel, Ackerman steering motions and rotations on the spot, while also being able to smoothly and continuously transition through gait sequences and velocities, from pentapedal to tripod gait by adjusting a small number of parameters, such as direction of travel, angle of travel, stroke length and duty factor. Larger copies of analysis diagrams can be seen in Appendix III: Analysis Diagrams.

4.4 Torque Analysis

4.4.1 Approximated Body Mass

Once the fundamentals of the body design and its constituent had been decided on, a reasonable estimate of over-all body mass is possible. The initial estimate gave an expected body mass of approximately 2.445kg (Table 1)

Item	Mass(g)	Quantity	Total Mass(g)
Body	185	1	280
Coxia	30	6	180
Femur	15	6	90
Tibia	20	6	130
Fixings			240
Cabling etc.	100		100
Prototype Board	40	2	80
Battery	200	1	200
Servomotors	55	18	990
Nanos	4	6	24
Mega	35	1	35
Rasp. Pi 2B	42	1	42
Servoshield		1	30
HC SR04	8	3	24
Total			2445

Table 1: Component Mass Estimates

4.4.2 Leg Joint Torques

Using these masses, an estimate of the expected torques on each joint is possible. An analysis of the forces on the legs while static in tripod pose, with the foot at the neutral position and the body height at 92mm follows.

With these parameters, in the following diagram; $\theta_2 = \theta_3 = 39°$ and $\theta_1 = \theta_4 = 72.5°$ (Figure 19)

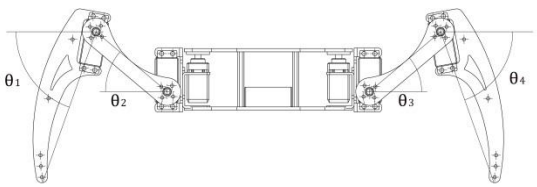


Figure 19: Leg angles

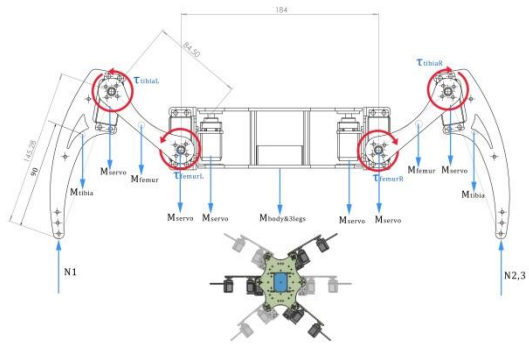


Figure 20: Hexapod FBD

M_0	Body + 3 legs	1845g	18.10N
M_1	Servo Motor	55g	0.54N
M_2	Femur	15g	0.147N
M_3	Tibia	20g	0.196N
L_1	Coxia to Coxia	103mm	0.103m
L_2	Coxia to Femur	29mm	0.029m
L_3	Femur	84mm	0.084m
L_4	Tibia	145mm	0.145m
L_5	Tibia COM from foot	90mm	0.09m

$$N_1 + N_2 + N_3 = Mg = 9.81 \times 2.445 = 24N \qquad \text{Eq 38}$$
$$N_2 = N_3$$

Balancing the net torques around the left foot, N_1, as the hexapod is stationary, the combined torques around the left foot equals zero. (Figure 20)

$$\sum \tau_{left\ foot} = -M_3 L_5 \cos\theta_1 - M_1 L_4 \cos\theta_1 - M_2\left(L_4\cos\theta_1 + \frac{L_3}{2}\cos\theta_2\right) - M_1(L_4\cos\theta_1 + L_3\cos\theta_2) - M_1(L_4\cos\theta_1 + L_3\cos\theta_2 + L_2) - M_0\left(L_4\cos\theta_1 + L_3\cos\theta_2 + L_2 + \frac{L_1}{2}\right) - 2M_1(L_4\cos\theta_1 + L_3\cos\theta_2 + L_2 + L_1) - 2M_1(L_4\cos\theta_1 + L_3\cos\theta_2 + 2L_2 + L_1) - 2M_2\left(L_4\cos\theta_1 + \frac{3L_3}{2}\cos\theta_2 + 2L_2 + L_1\right) - 2M_1(L_4\cos\theta_1 + 2L_3\cos\theta_2 + 2L_2 + L_1) - 2M_3((2L_4 - L_5)\cos\theta_1 + 2L_3\cos\theta_2 + 2L_2 + L_1) + 2N_2(2L_4\cos\theta_1 + 2L_3\cos\theta_2 + 2L_2 + L_1) = 0 \qquad \text{Eq 39}$$

Simplifying and solving for N_2, N_3 :

$$N_2 = \{ M_3((4L_4 - L_5)\cos\theta_1 + 4L_3\cos\theta_2 + 4L_2 + 2L_1) + M_2\left(3L_4\cos\theta_1 + \frac{7L_3}{2}\cos\theta_2 + 4L_2 + 2L_1\right) + M_1(9L_4\cos\theta_1 + 10L_3\cos\theta_2 + 11L_2 + 6L_1) + M_0\left(L_4\cos\theta_1 + L_3\cos\theta_2 + L_2 + \frac{L_1}{2}\right) \} / (4L_4\cos\theta_1 + 4L_3\cos\theta_2 + 4L_2 + 2L_1) \qquad \text{Eq 40}$$

$$N_2 = N_3 = 6.27N : N_1 = 11.46N$$

To calculate the torque around the left tibia joint, in tripod stance, while stationary, it has to balance the normal force at the tarsus, or foot:

$$\tau_{tibia} = N_1 L_4 \cos\theta_1 = 0.499N \cdot m \qquad \text{Eq 41}$$

To do the same for the Femur Joint:

$$\tau_{femur} = N_1(L_4\cos\theta_1 + L_3\cos\theta_2) \qquad \text{Eq 42}$$

$$= 1.25N \cdot m$$

To examine the maximum stationary torque allowed by the limits of range previously stated for the leg joint, θ_1 is set to 45° and θ_2 is set to 0°. With the legs on both sides set at the same angle, the weight distribution remains the same so our normal forces, N_1, N_2 and N_3 remain the same. The results this time then are:

$$\tau_{tibia} = 1.17N.m$$

$$\tau_{femur} = 2.14N.m$$

These results are problematic, as even in the neutral position, the torque on the femur joint exceeds the selected servo motors maximum torque at 6V of 1.08 N.m by 15.7%, and at full extension, the femur experiences almost twice that maximum, and the tibia just exceeds it. Additionally, the overall mass of the hexapod used here is a best estimate, and the finished platform may well be heavier, and that these calculations do not account for any extra load being carried by the hexapod, or extra dynamic torques experienced while in motion.

Ideally, the femur should be actuated with a servo motor that can deliver 3N.m torque and the Tibia and Coxia with servo motors that can deliver 2N.m – this would allow for a hexapod of overall body mass of greater than 3kg. Unfortunately such servos are outside of the limited budget of this project, so the options are limited to reducing the linkage length to reduce the torques, shedding some weight, or reducing the load by supporting the body, either partially or completely during practical runs. The first option would impact on the build time, while the third one is undesirable, as it would adversely affect testing on the force sensors in the legs/feet, and ideally the performance of the control system while travelling freely and underload should be observed and tested. The best options for weight reduction is to remove the batteries and have a tethered power supply, not have the higher performance boards (Arduino Mega and Raspberry Pi) on board for this stage, and to lose the top level of the 3 tier body frame. This saves around 390g.

Recalculating for this mass gives normal reactions at the tarsus of:

$$N_1 = 9.56N\ ;\ \ N_2 = N_3 = 5.28N$$

And torques in the neutral position of:

$$\tau_{tibia} = 0.415N.m$$

$$\tau_{femur} = 1.036N.m$$

And at the extreme position of:

$$\tau_{tibia} = 0.975N.m$$

$$\tau_{femur} = 1.774N.m$$

This puts the torques within the limits of servos around the neutral leg position, but is still considerably above the maximum torque on the femur joint at the extreme position, so a combination of tethered power supply and additional body support may be necessary for the purposes of this investigation.

Another solution under consideration is to reduce the load on the servos by adding spring tensioners at the femur and tibia joints, making the actuators do more work in the swing phase but reducing the load during the stride phase (Figure 21).

Considering the maximum torque on the joint while the leg is lifted will be when it is extended horizontally:

$$\tau_{tibia} = M_3(L_4 - L_5) \qquad \text{Eq 43}$$

$$\tau_{femur} = M_3(L_3 + L_4 - L_5) + M_2\left(\frac{1}{2}L_3\right) + M_1L_3 \qquad \text{Eq 44}$$

$$\tau_{tibia} = -0.0108N.m$$
$$\tau_{femur} = -0.0796N.m$$

Taking the extreme position torques of the untethered hexapod mass, and balancing the counter-spring mid-range gives a required servo torque range of:

$$\tau_{tibia} = \pm 0.590N.m$$
$$\tau_{femur} = \pm 1.110N.m$$

This significantly reduces the maximum load on the servos. Implementation of this solution would take significant design experimentation, fabrication and calibration time, however.

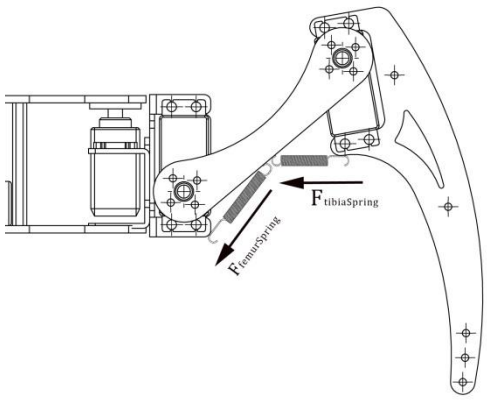


Figure 21: Proposed spring counter balance design

4.4.3 Autodesk Force Effect FBD

A simplified free body diagram was built using Autodesk Force Effect for further analysis, the results of this analysis confirmed the findings of the numerical analysis.

Force Effect was also used to examine other combinations of mass and torque to examine which ones would work (Figure 22 - 16). The conclusions of these enquiries supported the conclusions of the numerical analysis. Additionally, it indicated that the maximum body mass that would work with this geometry, with the previously defined extreme leg position, using femur servos with a torque < 12kgf.cm, was less than the combined mass of the legs, so unsolvable. Due to budget and resource constraints, for the purpose of investigating the proposed control system, the project proceeded with the underpowered servos, opting for the supported and tethered solution. The Force Effect generated report can be found in Appendix VIII: Force Effect Report.

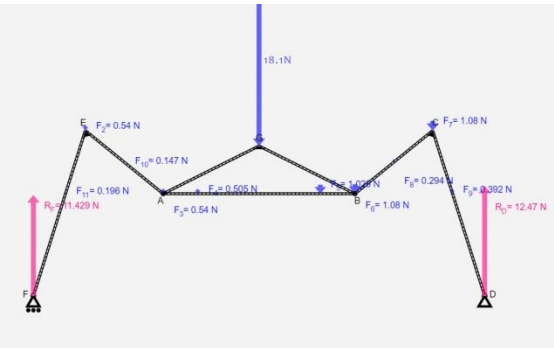


Figure 22: ForceEffect FBD

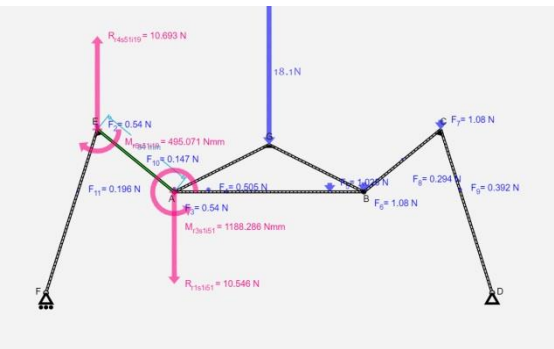


Figure 23: ForceEffect FBD with femur Torque

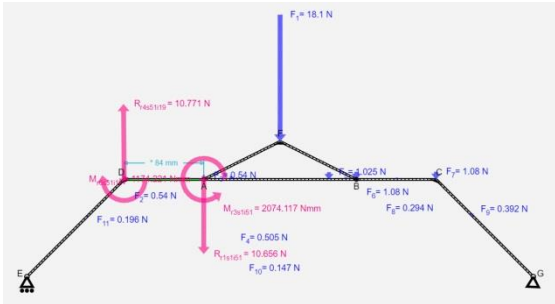


Figure 24: ForceEffect FBD extreme leg position

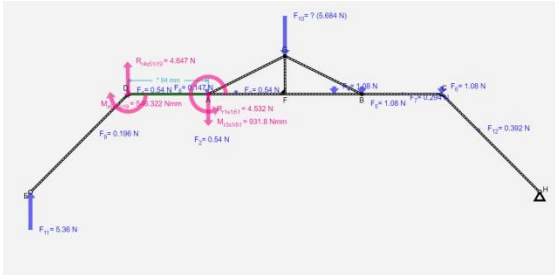


Figure 25: FBD analysis for servo <12kgf.cm

5 Simulation

5.1 Simulation Software

A range of options were investigated for simulating the platform prior to construction. The final short list of simulation packages which were considered are MRDS, Morse, Gazebo, Webots and V-Rep.

5.1.1 Microsoft Robotic Development Suite

Microsoft Robotic Development Suite (MRDS) is a free, full featured 3D Robotic Simulation environments developed by Microsoft. It is available for Windows, and is compatible with C#, Java, or their own, proprietorial Visual Programming Language. It is designed with an easy to use GUI and is meant to be accessible for hobbyists with limited programming experience. For simulation, MRDS uses NVIDIA PHYSX technology.

While MRDS presents itself as accessible and easy to use, it is limited by its dependencies on other Microsoft products – does not seem to offer compatibility with ROS and Python, which were still being considered as integral to the project at the simulation phase.

5.1.2 Morse

The Modular Open Robotic Simulation Engine, or MORSE is built on Blender, the Bullet physics engine and Python, and is controlled almost entirely from the Command Line (Figure 26). Morse comes with a number of predefined environments from small to large, indoor and outdoor, a range of standard sensors and a selection of prebuilt robot platforms, with the ability to import others. Morse is only available on Linux Ubuntu, and supports the Robot Operating System (ROS). Morse does not have any embedded advanced algorithms, such as path planning, and its producers state that it cannot be considered to be physically accurate for fine motor functions in its current version (1.2.1) (Open Robots 2016).

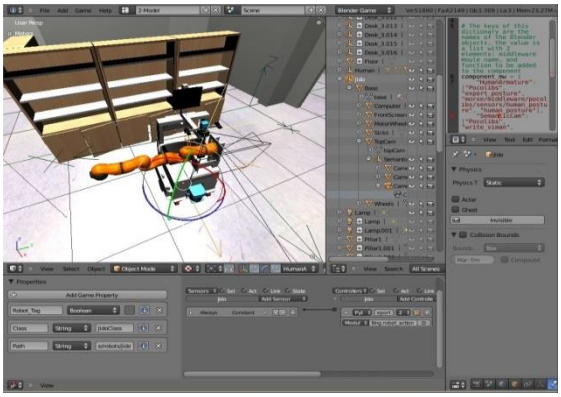


Figure 26: : Morse Screenshot (Open Robots 2016)

5.1.3 Gazebo

Gazebo is a free, open source robotic 3D simulation platform that is heavily integrated with the Robot Operating System (ROS), and is currently only available for Linux Debian. The current version of Gazebo (4.0.0) utilizes the Open Graphics Rendering Engine (OGRE) for 3D modelling and rendering, as access to a number of physics engines – namely ODE, Bullet, Simbody and Dynamic Animation Robotic Toolkit (DART) – all of which are high performance open source physics engines. Gazebo allows you to simulate sensor data from a wide range of sensors, comes with a number of pre packed robot models or allows you to import your own model, and allows for programming in C, C++, Java, ROS and Python, with some plugins available for other languages such as Java. Gazebo has a GUI, but seems to favour its Command Line Tool, making for a less intuitive environment for the first time user (Open Source Robotics Foundation, 2016). Gazebo is the simulation Software recommended in conjunction with ROS in the Darpa Robotic Challenge (Ackerman 2012).

5.1.4 Webots

Webots is a popular, full featured, Swiss developed 3D simulation platform used by over 1000 universities and research institutes globally (Figure 27). It features a library of prebuilt examples and environment, as well as extensive sensor and actuator libraries and an intuitive GUI and programming interface. Webots uses the Open Dynamics Engine (ODE) physics library, and the professional release allows for additional physics engine plug ins. Webots is available for Window,

Mac OSX and Linux, and allows for programming robots in C, C++, Python, MATLAB, Java and ROS (Cyberbotics, 2016).

Webots was the simulator used by Matt Denton, who designed the Mantis Walking Machine – currently one of the largest functioning hexapod walking platforms (Mantis, 2016). However the full featured version of Webots is not cheap, at $US2300 for the Academic discounted Pro version, and the Educational version, which is lacking the additional Physics plugin, Supervisorial programming and Fast Simulator Mode, is still $US350 for a licence (Cyberbotics, 2016).

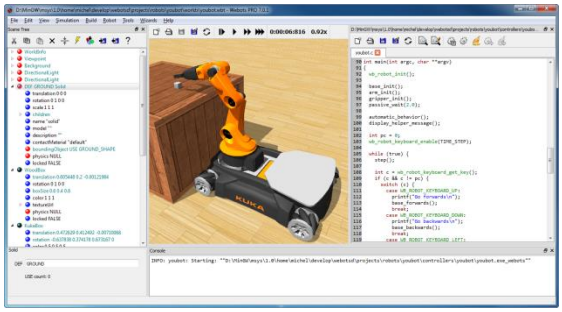


Figure 27: Webots Screenshot (Cyberbotics 2014)

5.2 V-Rep

Virtual Robot Experimentation Platform, or V-Rep is a commercial 3D robotic simulation package that uses some open source modules. V-Rep offers a full featured Educational licence, which differs only from the Professional version in that it is watermarked. It is available for Windows, Linux and OSX. V-Rep comes with an easy to use, intuitive GUI, and a large range of pre packed models, as well being able to easily import and define 3D models from other CAD programs. V-Rep has a range of configurable sensors and actuators, and uses the open source ODE, Bullet and Vortex Physics Engines. V-Rep has a very versatile API, allowing for six different programming approaches – embedded scripts, add-ons, plugins, remote API, ROS nodes and Custom Client/Server – and is compatible with a range of programming languages; C, C++, Java, Python, Lua, Matlab, ROS, Octave and Urbi. V-Rep also has comprehensive motion planning, Data and Visualisation recording and exporting, inverse and forward kinematic calculations, and allows for easy editing of models within the simulator, and many other features besides.

After reviewing the available platforms, V-Rep was chosen as the software of choice for simulation as it was the best choice for versatility, range of tools, language compatibility, ease of use and cost effectiveness.

In order to simulate the Hexapod in V-Rep, the model had to be first imported from Solidworks, then the model had to be configured for simulation.

STL is the compatible file format between these two programs, however STL exports from Solidworks have a very high polygon count, so first the model had to be edited in Meshlabs (a free, open source 3d model editing software) where the polygon count was reduced from around 180,000 to a more manageable 50,000. This took some experimenting in order to achieve the desired polygon count with minimal distortion to the model (Figure 28).

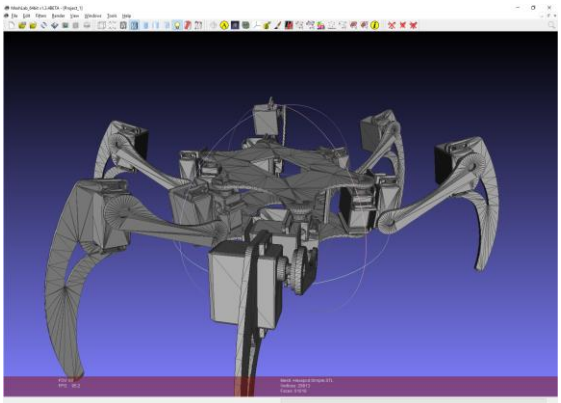


Figure 28: Model editing in Meshlabs

Once imported into V-Rep, the model had to be then configured for simulation. This involved separating the various body components, building primitive shape versions for each component and allocating their physical properties such as mass for physics analysis, and allocating and positioning all of the active joints on the model.

In retrospect, the whole process would have been made easier, quicker and more accurate if each component part had been imported into V-Rep

separately rather than configuring the model as a whole.

Once the model was configured and ready for simulation, a piece of code was modified for each leg in order to run a sequence analogous to movement expected from the real world hexapod. Similar to the proposed control system, a code controller was linked to each individual leg, however the leg sequence was controlled by sequential timing rather than a reflex signal network, and the code used V-Reps inbuilt IK function for foot trajectory. From this, the simulator could be used to generate various outputs such as leg joint torques and velocities, foot position and velocity, and be used to test various code and control approaches (Figure 29).

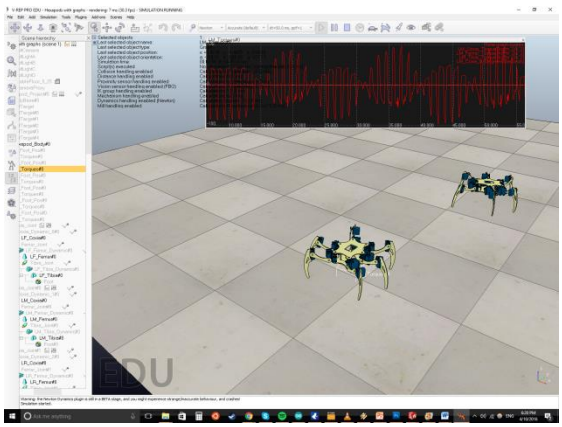
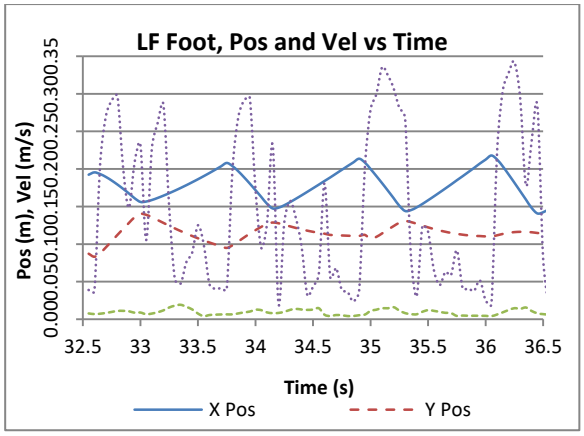


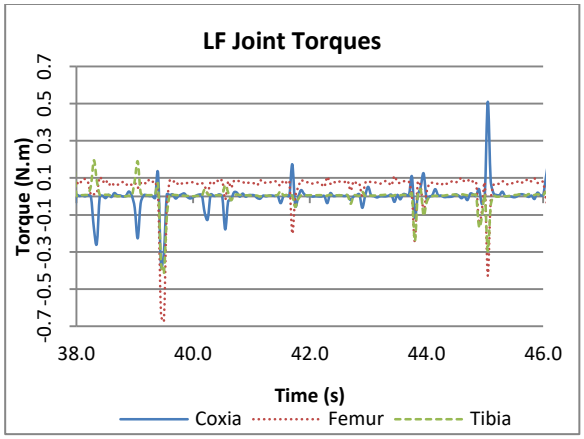
Figure 29: Screenshot of V Rep Simulation with Leg Torque Output

5.2.1 Leg Torques and Foot Position

A sample of some of the data exported from the simulator is presented here. If the simulator is to be taken as producing a reliable kinetic model of the hexapod, the leg torques appear to peak around 1.2N.m, which is above the stated torque on the servo motors being considered for this project. The numerical analysis discussed in the previous section already indicated that the servos on this project are underpowered, particularly for the femur actuation, the simulation further verifies this and reinforces the need for the proposed tethered and supported solution for demonstration.



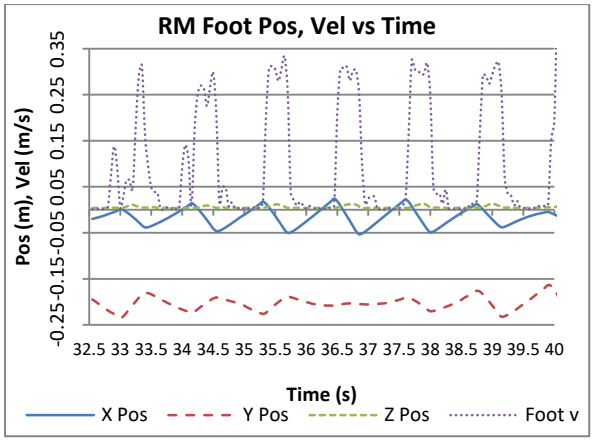
Graph 3: LF Foot Position and Velocity



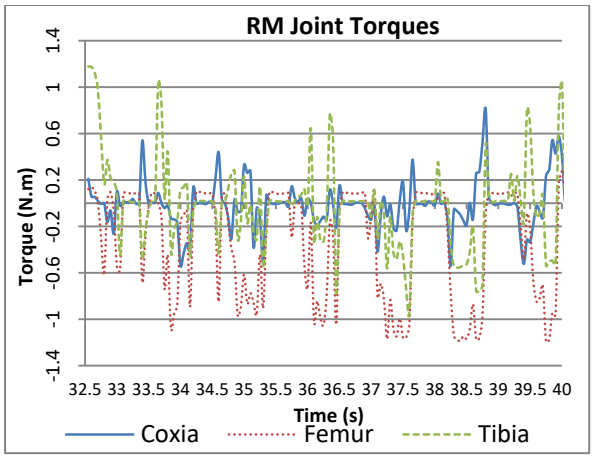
Graph 4: LF Joint Torques

In the graphs exported for the Right Middle leg (Graph 5 -7) the correlation between foot position, the torques on the various joints and the angular velocity of the coxia joint is clearly evident.

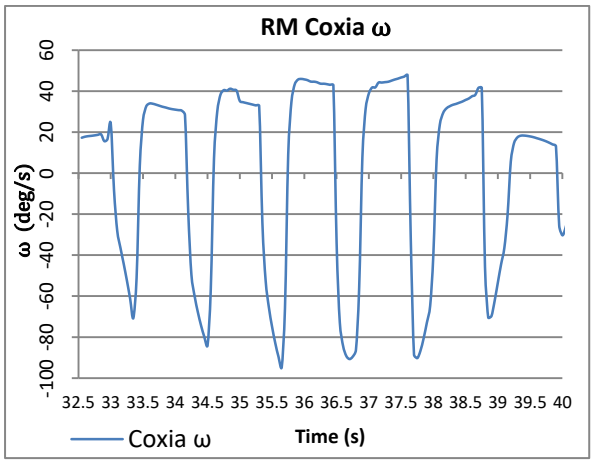
The process of learning how to use the software, importing and calibrating the model, building a primitives physics simulation model, modifying the code to emulate some of the desired motions, fine tuning the model and program until it was running smoothly, and exporting and processing the data consumed a considerably larger portion of the schedule than originally allocated to it. V-Rep seems to be a very versatile simulation tool with a lot of features and applications, it would be good for future projects to be able to more fully exploit this software in the design phase.



Graph 5: RM Foot Position and Velocity



Graph 6: RM Joint Torques



Graph 7: RM Coxia Angular Velocity

6 Electromechanical Design

6.1 Servo Motors

The servos used in this project are the hobby servo TowerPro MG996R (Figure 30). These servos were chosen as they are reasonably robust, with metal gears and reasonably high torque (11kgf.cm stall torque at 6V), and they are very low priced for their size. With 18DOF to actuate, the servo motors make up the largest proportion of production cost, as the project, so economy was important when choosing the servos. Unfortunately, analysis indicated that these servos were inadequate to actuate a fully functional hexapod platform of the design being proposed.

For the purposes of investigating and demonstrating the effectiveness of the control system proposed, the platform was built with these servos anyway, with the intention of replacing them with more adequate servo motors when budget allowed.

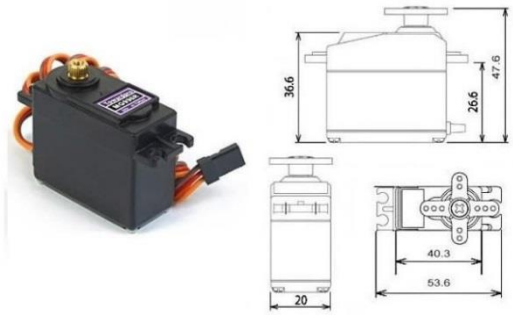


Figure 30: TowerPro MG996R (electronicoscaldas.com 2016)

Weight: 55 g
Dimension: 40.7 x 19.7 x 42.9 mm approx.
Stall torque: 9.4 kgf·cm (4.8V),
11 kgf·cm (6 V)
Operating speed: 0.17 s/60° (4.8 V),
0.14 s/60° (6 V)
Operating voltage: 4.8 V a 7.2 V
Running Current 500 mA –900 mA (6V)
Stall Current 2.5 A (6V)
Dead band width: 5 µs
Stable and shock proof double ball bearing design
Temperature range: 0 °C –55 °C

Initial testing of the stock servos revealed wide variation between individual servos performance in relation to pulse range, centring pulse and angle change to pulse width increment.

Initially, in order to mitigate these factors, and to build a more reliable control system, using position feedback from the servo potentiometer, closing the loop back to the microcontroller, and allowing the implementation of PID control using real world position data, the servos were modified with an additional signal wire attached to the servo motor potentiometer (Figure 31).

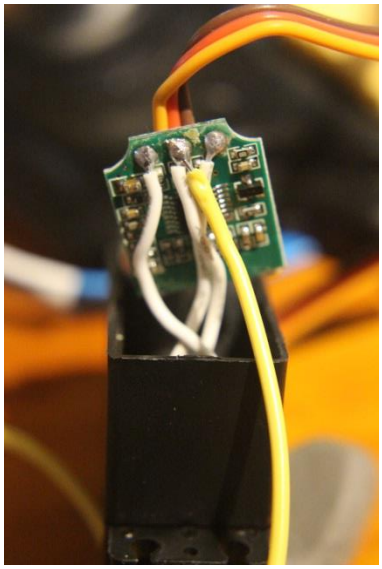


Figure 31: Position feedback modification

6.1.1 Calibration:

6.1.1.1 Electronic Calibration

The position feedback on the servo was also initially used to calibrate each servo, with a calibration program written which tested the full range of each servo motor, outputting the true high, low and centre pulse, then running a routine to move to each extreme position so that it's true angle range could also be measured (see Appendix VII: Calibration Programs).

During the calibration process it became obvious that this method of position feedback was not going to be practical for real time operation when the walk routine was running, as in order to get a reliable reading when the motor was starting from stationary, changing direction or moving through a larger swing, the input signal appeared to have a

settling time of up to 200ms, with significant overshoot in the first 50ms.

6.1.1.2 Physical Calibration

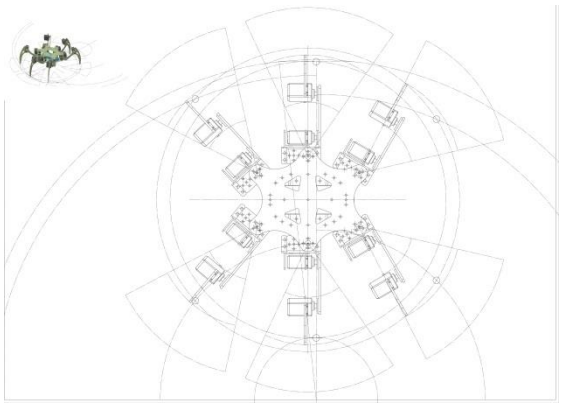


Figure 32: Physical Calibration Map

When the calibration data obtained in the first method was implemented in the IK code to set the individual servo parameters, there was significant inaccuracy in some of the legs, so clearly the calibration data for some of the servos was not accurate enough for the IK controller to work accurately and reliably. A second calibration process was devised, where the foot was positioned at set points on a calibration map (Figure 32) using a servo controller, from which the leg linkage angles could be accurately determined and the pulse signal for all three leg servos outputted. From these outputs, the individual servo calibration parameters could be calculated and implemented in the leg controller code. This method yielded much more reliable results (Figure 33). All calibration data can be found in Appendix V: Calibration Data.

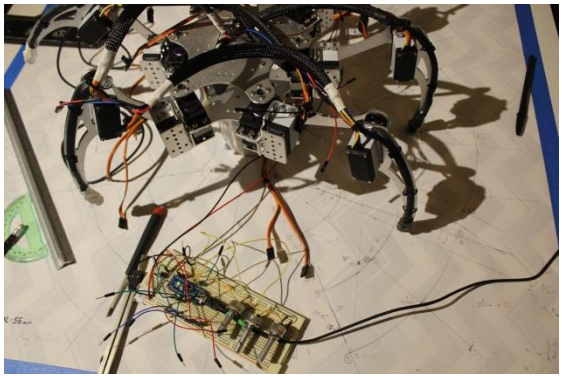


Figure 33: Physical Calibration

6.2 Microprocessors

Several microprocessors were researched for this project.

6.2.1 Arduino Nano

Microcontroller	Atmel ATmega328
Operating Voltage (logic level)	5 V
Input Voltage (recommended)	7-12 V
Input Voltage (limits)	6-20 V
Digital I/O Pins	14 (6 PWM output)
Analog Input Pins:	8
DC Current per I/O Pin	40 mA
Flash Memory 32 KB)	2 KB used by bootloader
SRAM	2 KB
EEPROM	1 KB
Clock Speed	16 MHz

The Arduino Nano was selected as the individual leg controller due to the compact size of the development board and its low cost (Figure 34). It uses an ATmega 328 processor, with 8 ADC pins, two of which also function as the I2C clock and data pins, and 14 Digital IO pins – 6 of which can be used for PWM (three are needed on each chip in this project), it has just enough pins to suit the requirements of the proposed system. With 32kB of flash memory, the code needs to be kept efficient.

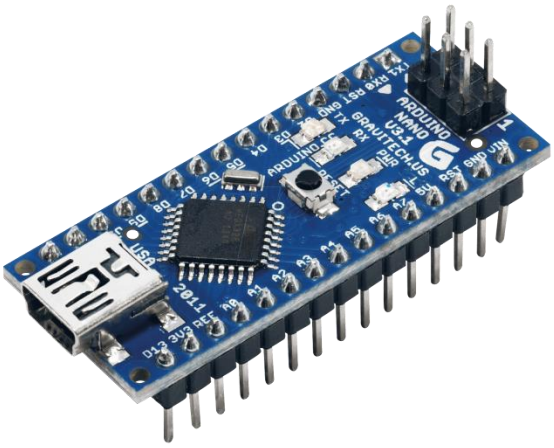


Figure 34: Arduino Nano (arduino.cc 2016)

6.2.2 Arduino Mega

Arduino Mega

Microcontroller ATmega2560
Operating Voltage 5V
Input Voltage (recommended) 7-12V
Input Voltage (limit) 6-20V
Digital I/O Pins 54 (15 PWM output)
Analog Input Pins 16
DC Current per I/O Pin 20 mA
DC Current for 3.3V Pin 50 mA
Flash Memory 256 KB , 8 KB bootloader
SRAM 8 KB
EEPROM 4 KB
Clock Speed 16 MHz
LED_BUILTIN 13

For the "Motor Cortex" controller, initially using a microcomputer based board such as the Raspberry Pi was considered, but the ATmega2560 based Arduino Mega was instead favoured for this application. Microcontroller based boards such as this one are better suited for real time interfacing and implementation of sensor inputs and actuator outputs, and connecting it to, and communication with the rest of the leg microcontrollers is simpler to implement (Figure 35).

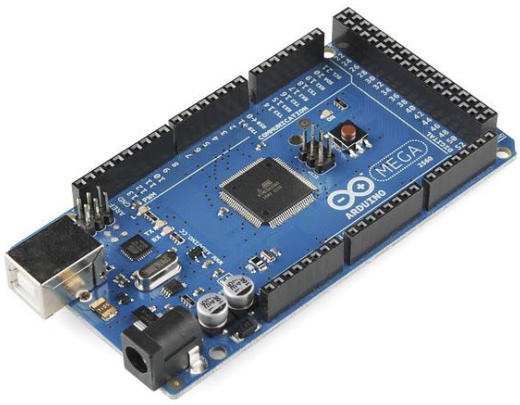


Figure 35: Arduino Mega (arduino.cc 2016)

6.2.3 Raspberry Pi 3

The raspberry Pi 3 is a microcomputer on a breakout board targeted a the STEM education market. It uses a A 1.2GHz 64-bit quad-core ARMv8 CPU as its processor, and Videocore IV 3d graphics core, which can run OpenGL, and is Bluetooth and wireless enabled (Figure 36).

Figure 36: Raspberry Pi 3 (raspberrypi.org, 2016)

This board is less efficient than the dedicated microcontrollers when it comes to interfacing with sensors and actuators, but with a parallel processors and a dedicated GPU, it is being considered for running more complex tasks such as odometry, whole body simulation and computer vision, as well as map building and path planning. Robot Operating System (ROS) was investigated as an option for networking with the other processors, though the Raspberry Pi wasn't capable of real time execution of functions that used the ROS 3d visualisation tool, R-Vis, which includes most of its computer vision functions such as visual odometry and SLAM, as well as 3D body modelling.

Raspberry Pi 3

SoC: Broadcom BCM2837
CPU: 4× ARM Cortex-A53, 1.2GHz
GPU: Broadcom VideoCore IV
RAM: 1GB LPDDR2 (900 MHz)
Networking: 10/100 Ethernet, 2.4GHz 802.11n wireless
Bluetooth: Bluetooth 4.1 Classic, Bluetooth Low Energy
Storage: microSD
GPIO: 40-pin header, populated
Ports: HDMI, 3.5mm analogue audio-video jack, 4× USB 2.0, Ethernet, Camera Serial Interface (CSI), Display Serial Interface (DSI)

The Raspberry Pi requires a 5V 2A power supply, and uses 3.3V on its GPIO pins, so some voltage protection would also be required when interfacing with the ATmega processors and any other devices that use 5V for their digital HIGH.

6.2.4 Other Microprocessors

6.2.4.1 Odroid XU4

Odroid XU4
Samsung Exynos5422 Cortex™-A15 2Ghz andCortex™-A7 Octa core CPUs Mali-T628 MP6(OpenGL ES 3.0/2.0/1.1 and OpenCL 1.1 Full profile) 2Gbyte LPDDR3 RAM PoP stacked eMMC5.0 HS400 Flash Storage 2 x USB 3.0 Host, 1 x USB 2.0 Host Gigabit Ethernet port HDMI 1.4a for display

When the limitations of the Raspberry Pi became apparent, other options were also investigated. The Odroid series from HardKernel was looked at as a good option for running visual processing applications – in particular the Odroid XU4– it is a 64 bit, 2GHz Octacore built off the ARM big.LITTLE processing, which uses ARM A15 and ARM A7 processors optimised for processing performance and power efficiency specifically for mobile devices. The development board is very similar to that of the Pi series, but it has significantly higher performance capabilities (Figure 37).

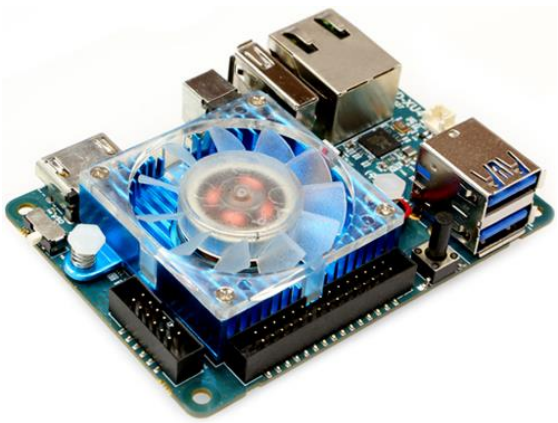


Figure 37: Odroid XU4 *(HardKernel.com 2016)*

A couple of the differences with what the development board offers with the Raspberry Pi is the absence of on-board Wi-Fi or Bluetooth capabilities, though it does have an in built ADC, which the Raspberry Pi lacks. Like the Raspberry Pi, it uses an external Flash memory such as an SD card, and has twice the on-board RAM at 2Gb It also uses the superior performance Mali-T628 MP6 GPU. It has been used as the on-board processor for visual processing and autonomy in a number of existing mobile robot projects (Nardi, 2014) but consumes more power when in operation – it needs a 4A 5V power supply compared to 2A 5V with the Pi. Adding such performance capabilities ended up being outside the scope of this project, but this board is certainly of interest for future development.

6.2.4.2 STM32 ARM Cortex M3

Another microprocessor options that was considered for the leg controller was to use STM32 32bit ARM Cortex M3 based boards such as the Maple Mini from Leaflabs (Figure 38).

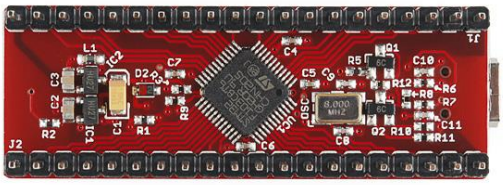


Figure 38: STM32 Maple Mine (Leaf Labs 2016)

These are low cost chips that are available on a breadboard compatible development board not much larger than the Nano, but offer much greater performance and capabilities. With 32 bit processing compared to the Nano's 8 bit, a clock speed 4.5 times that of the Nano at 72MHz, 4 times the Flash Memory (128Kb) and 10 times the SRAM (20Kb), as well as more Digital pins with 34 GPIO's with 12 PWM pins.

STM32 F103RCBT6:
a 32-bit ARM Cortex M3 microprocessor:

Clock Speed: 72 MHz
Input Voltage (recommended): 3.0V-12V
34 Digital I/O Pins (GPIO)
9 Analog Input Pins
12-bit ADC resolution
12 PWM pins at 16-bit resolution
Dedicated USB port for programming and communications
128 KB Flash and 20 KB SRAM
Integrated SPI (2) and I2C (2)
7 Channels of Direct Memory Access
3 USART devices
Four 4-channel Timers

A number of these were ordered for testing, but due to postage delays, they did not arrive in time to use for this project.

6.3 Sensors

As well as the position feedback signal input attempted with the servomotors, the desired functionality for the leg controller also required force feedback from the legs and a bump sensor in the foot for detecting and avoiding small obstacles.

6.3.1 Force Sensor Methods:

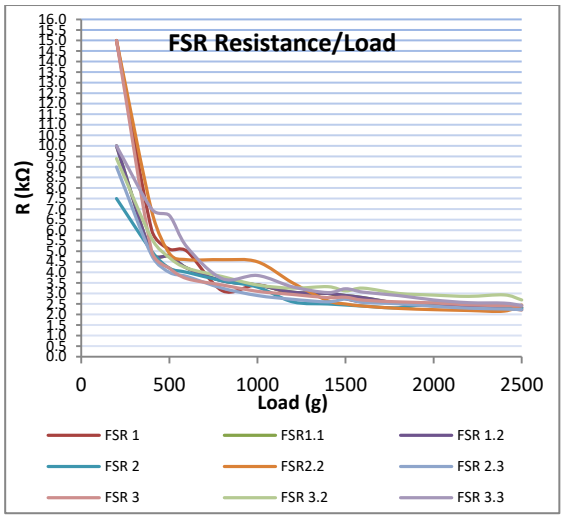
6.3.1.1 Hall Effect Sensor

Initially custom stress sensors were attempted, using 1.2mm neodymium magnets embedded in the top of the tibia linkage, a linear hall effect sensor mounted above the magnet but attached to the tibia servo, and the tibial linkage fastened to the tibial servo with rubber bushes to allow for a small amount of flex between the mounted servo and the tibia when under load. When under load, the position of the magnet relative to the sensor would shift, giving an indication of the load. This approach gave good resolution for the variation in tibial load, but the assembly process required a high level of precision to ensure consistent performance across all 6 legs, and the initial assembly process was not robust enough for continuous use.

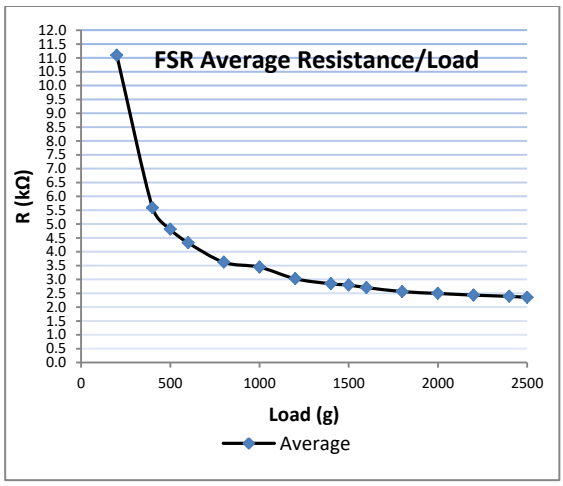
6.3.1.2 FSR

Figure 39: Lynxmotion FSR-01 (robotshop.com 2016)

The second approach was to use 8mm diameter force sensitive resistors on the end of tarsus (Figure 39), with a silicon pad "foot". This method proved to be much more robust, but lacked the resolution of the first method. For detecting when the leg was under load it was quite adequate though.



Graph 8: FSR Resistance/Load



Graph 9: FSR Resistance/Load average

From the numerical torque analysis, the load range experienced at the tarsus is 600g – 2200 g. This load translates to a (nonlinear) resistance range of 4.5kΩ – 2.5kΩ, according to test results. A 1kΩ resistor was used as a voltage divider when reading the FSR range with the microcontroller. Calibration data and larger graphs can be found in Appendix V: Calibration Data.

6.3.2 Bump Sensor

To detect collisions with small objects, steps and rocks, a bump sensor was constructed using a simple momentary micro switch with a silicon bumper. The switch and force sensor were mounted on the same "tarsus" component and attached to the tibia (Figure 40).

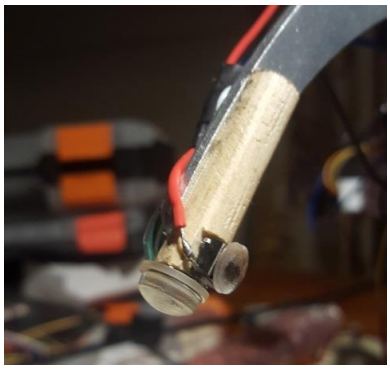


Figure 40: Force and Bump foot sensors

6.3.3 Other sensors.

Figure 41: HC SR04 ultrasonic range finder (robotshop.com 2016)

Other sensors were considered for implementing the higher functionality to the robot platform. In order to detect obstacles and implement obstacle avoidance methods, the Motor Cortex controller is to interface with low cost ultrasonic range finders, HC-SR04 (Figure 41). For pose control, tip detection and sensor based tracking the IMU – MPU 9150, which is a combined accelerometer, gyroscope and magnetometer (Figure 42).

Figure 42: MPU 9150 (sparkfun.com 2016)

For computer vision, visual odometry and other visual processes, the graphical processing board, or "Visual Cortex" would, of course, have to interface with a camera and possible some sort of RGB-D sensor. As this was beyond the possible scope of this project not further research was done in this area.

6.4 Power System

To work out the requirements of the power supply, a number of metrics were applied. Running 18 high torque servomotors, under load, is the largest demand on the system.

With a stall current of 2.5A, the maximum load on the system could be up to 45A, but this would be in the extremely unlikely event that all motors stalled simultaneously. The running current at 6V is stated as 500mA – 900mA. This gives a theoretical running range of 9 – 16.2A.

When testing the individual legs, when running but not under load, the current draw for one leg peaked at around 1.85A, giving an overall load, with all 6 legs operating but not supporting any weight, of 11.1A

Applying a numerical analysis, calculating the probable load from some generalised calculations of the torques experienced by all of the motors during normal running, which 3 of the legs in swing mode and the other three with a spread of the body load evenly shared. To simplify the calculations, all leg angles are set at 45° for stance, and horizontal for swing. Torque analysis has not been done for the Coxia servo, so it well be estimated being close to but less than the femur torque.

Torques for legs in swing phase:

$$\tau_{tibia} = M_3(L_4 - L_5)$$

$$\tau_{femur} = M_3(L_3 + L_4 - L_5) + M_2\left(\frac{1}{2}L_3\right) + M_1L_3$$

$$\tau_{tibia} = -0.0108N.m$$
$$\tau_{femur} = -0.0796N.m$$
$$\tau_{coxia} = -0.07N.m$$

Torques for leg in stride:

$$\tau_{tibia} = N_1L_4\cos\theta_1$$

$$\tau_{femur} = N_1(L_4\cos\theta_1 + L_3\cos\theta_2)$$

$$\tau_{tibia} = 0.82N.m$$
$$\tau_{femur} = 0.9N.m$$
$$\tau_{coxia} = 0.85N.m$$

To calculate the power needed for this configurations:

$$P = 2\pi\frac{rpm}{60}\tau \qquad \text{Eq 45}$$

Where P is power in Watts, and rpm/60, according to the MG996R specs is 0.9 at 6V

No efficiency data is available for these servomotors, but based on other studies and documentation *(micromo.com, 2016; Sawicz, 2016)*, a generalised assumption of 40% efficiency will be made:

$$P = 46.3W \times \frac{100}{40} = 115.8W \qquad \text{Eq 46}$$

$$I = \frac{P}{V} = 19.3A \qquad \text{Eq 47}$$

The microcontrollers running the legs individually do not consume that much power:

According to the datasheet, the pins on the ATmega 328 can provide up to 200mA, as well as additional power needed by the processor and leds, with a 25% safety factor, each processor will need a maximum 250mA supply, making 1.5A for all 6 leg controllers.

Applying the same calculations to the Arduino Mega, the maximum current it would need is also around 1.5A.

As the power demands of the actuators and the microcontrollers is quite different, and to protect the microcontrollers from any potentially damaging current spikes, the voltage supply circuits for the servomotors and the microcontrollers will be isolated from each other as much as possible, using a single power source. The proposed power supply uses an 11.1V 5Ah, 20C LiPo battery as its source, with separate DC-DC step down variable switching power supplies for each circuit.

A 20C 5Ah battery should be able to supply at least 4 times the expected current demand, and should also have capacity for 10 – 15 minutes of continuous operation at the expected current demand. The switching power supplies enable the separate voltages to be set with minimum power loss though conversion.

Figure 43: Mini 360 DC-DC Buck Converter Step Down Module (Ali Express 2016)

The servomotors are supplied by two 15A capable buck converters (Figure 44), one for each side of the hexapod, and the microcontrollers have their own 3A power supply (Figure 43).

Figure 44: DC 15A Buck Adjustable 4-32V 12V to 1.2-32V 5V Converter (Ali Express 2016)

7 Control Design

7.1 Control Design Methodology

The control system design is the key component of this thesis. To meet the goals of the project, it should be decentralised, using distributed computing. Each leg controller should be essentially autonomous, using minimal inter-controller signalling based on coordination rules derived from observing insect behaviour. The foundation of the control network design is the WalkNET system originally developed by Holk Cruse. This approach has been used in a number of other research projects since its original development, particularly in the development of behaviour based reactive robotic control systems that favour a distributed architecture. Many of these projects develop quite complicated networks between the leg controllers, relying on a range of suppression and exciter signals, often using Neural Network programming for gait optimisation, and a Central Pattern Generator to maintain a stable and regular gait rhythm. Other designs rely entirely on Neural Network programming and force and position feedback from the legs in concert with the coordination signals from the other legs to generate all aspects of the leg trajectory including AEP and PEP.

The purpose of this investigation is to design the control system to be as robust, stable and as adaptable as possible, but with the simplest architecture necessary to achieve these goals. The distributed computation approach should reduce the computational complexity, and free up computational resources for the higher level controllers. It was also proposed that a continuum of stable and regular gaits would emerge from this system without the need for a CPG or any other central controller, but only through varying the duty factor and through configuring the coordination rule reflex signals correctly.

7.2 Leg Controller

As such, the controller was designed so that the individual leg behaviours were built starting with the basic functionality of being able to accurately track a desired step path, with additional functionality built of this foundation using force and position feedback from the leg sensors, and reflex signals from the control network. A PID controller using Inverse Kinematics, with both force and position was proposed as the primary leg control method employed in the individual leg control code (Figure 45). A Neural Network based approach, though popular in other works, was considered to be adding unnecessary complexity for this investigation. The approach for this investigation is to get the best performance and the widest range of behaviours while finding the simplest network to achieve those aims, from this solid foundation, further sophistication can be built on the network, so Neural Networks may be a valid direction for further development.

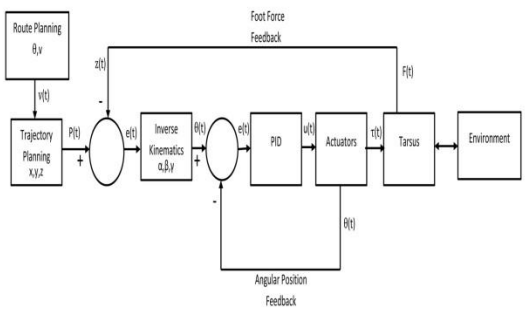


Figure 45: Leg Controller Diagram

7.3 Control Network

For the inter-controller coordination reflex network, the WalkNET coordination rules were analysed and simplified. In this design (Figure 46), it was determined that coordination influence 1 & 2 (return stroke inhibits start of return stroke, and start of power stroke excites start of return stroke) and could be combined, as both signalled in the same direction and both were functions of the triggering legs swing or return phase. Further it was decided that the lateral reflex was only necessary between the rear legs, as the step sequence would then iterate forward independently on both sides.

The third influence – Caudal position excites start of return stroke, which iterates front to back – was dropped from this design as it was felt it added unnecessary complexity at this stage of the investigation, and didn't seem to contribute a great deal to the functionality already handled by the first

two influences. The fifth influence is the "trading on tarsus" reflex, and is a corrective reflex observed in insects when they trip over their own feet, This is dealt with in the software, with limits set on the servo range ensuring such collisions are avoided. The fourth influence in the WalkNET model deals with spreading the lead evenly between supporting legs. This method is worth investigating for future development, but to implement it at this stage was beyond the scope of this project. It requires the sharing of analogue inputs from the legs force sensors between the legs, and the implementation of some sort of differential balancing function to evenly share the load. The force sensitive resistors used in the feet of this platform didn't provide enough resolution or repeatability to implement this method, but were adequate for sensing when the leg was under load, and for adjusting the foots target z coordinate.

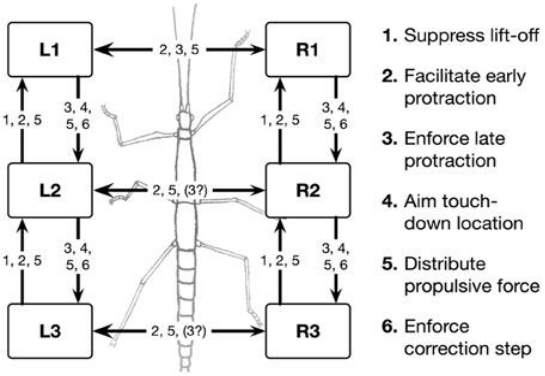


Figure 46: WalkNet coordination Influences (Cruse 2013)

The leg control network is intended to provide a foundation for the locomotion control of a mobile robotic platform which would have other levels of functionality, such as steering control and obstacle avoidance, computer vision, path planning, and AI operations. Continuing the distributed computing model inspired animal nervous system structures, these higher functions would also be managed by separate, dedicated microprocessors analogous to an animal's visual and motor cortex. From these requirements then, the ArthroBOT control network architecture was designed (Figure 47). The control network diagram informed both the circuit design and the software development. Larger versions of the Control Diagrams can be found in Appendix II: Control Diagrams.

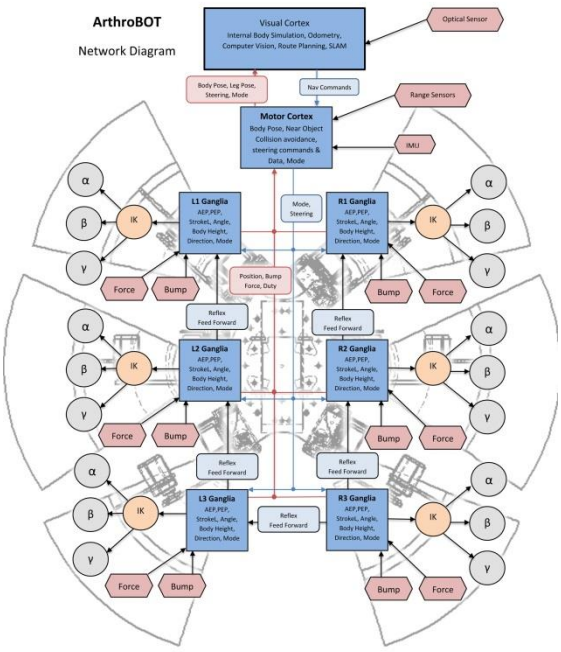


Figure 47: Control Network Diagram

8 Software Development:

The individual leg control software was developed primarily from the formulae derived in section 4. A number of modules were programmed and tested independently, as well as various sketches used for testing and calibration of components.

As the same program had to be implemented and tested on 6 separate controllers, with all debugging and updating processes having to be iterated at least 6 times, it was important that the program operate as universally as possible. There are a number of factors that complicate the implementation of this functionality, primarily, the wide variation in calibration settings between the servos meant that a method had to be implemented that facilitated easy integration of the various servo parameters. Secondly, the different geometric configurations for each leg had to be accounted for – for example the various relative neutral positions for each foot, and the reversal of relative servo direction for the same motion left to right.

Initially the servo parameters were stored in the EEPROM of each chip, with a function written to retrieve them on start-up – this method was later dropped, as the unreliability of the servos meant that they would occasionally have to be replaced and recalibrated, with the new data needing to be stored at the right address on the EEPROM, which became a cumbersome process as most of the parameters took up 2 bytes of data, which had to be split, stored, retrieved and concatenated. It also meant that the control chips themselves were not easily interchangeable.

Now each leg controller just needs one integer changed to indicate which leg it is to control. This then controls it's I2C address, how various angle and direction variables are dealt with in the motion functions, how it handles reflex signals, and the servo parameters are stored and installed in a switch/case function - getParams() - that is called during set up.

Various software sketches were written in order to electronically and physically calibrate and test the leg servos. These sketches are in Appendix VII: Calibration Programs.

8.1 PID

The initial design proposal incorporated position feedback from the joint actuators, with the intention that this could be used for PID control, either directly with the desired servo angle or mathematically using the Inverse Kinematic function and the desired foot position. Some time was spent developing a PID controller as part of the leg control program before it became apparent that the settling time on the position signal meant that this method of position feedback would be impractical for real time operation of the leg.

8.2 Inverse Kinematics

The main foot position controller uses an inverse kinematics function that uses the equations derived in chapter 4. To avoid rounding errors, the maths is calculated using floats. The leg dimensions are constants, and the function returns the ideal joint angle in degrees.

```
//IK maths Function

void IKtrig(int x, int y, int z)
{
  float x1 = x;
  float y1 = y;
  float z1 = z;
  L1 = hypot(x1, y1);

  gama = atan2(x1, y1) * dg;

  L = hypot((L1 - CL), z1);

  beta = acos((sq(TL) + sq(FL) - sq(L)) / (
2 * TL * FL)) * dg;

  alpha1 = acos(z1 / L) * dg;
  alpha2 = acos((sq(FL) + sq(L) - sq(TL)) /
(2 * FL * L)) * dg;
  alpha = alpha1 + alpha2;

  ta[ALPHA] = alpha;
  ta[BETA] = beta;
  ta[GAMA] = gama;

  for (int i = 0; i < numServos; i++)
  {
    if (ta[i] < minA[i])
    {
      ta[i] = minA[i];
    }
    if (ta[i] > maxA[i])
    {
      ta[i] = maxA[i];
    }
  }
```

Figure 48: IK Maths Function

The IK function also limits the target angle so the servos stay within their stated limits.

8.3 Mapping the Angle and posFoot()

As the actual position for each servo differed from the ideal angle returned by IKtrig(), and depending on the leg orientation, the angle range is often the reverse of the convention used in the function, the angle then had to be mapped using the target servos defining parameters:

```
// function converts IK angles to real angles and writes them to servo

void mapWriteServo (int s)//s = servo
{
  int x = map(ta[s], minA[s], maxA[s], realMinA[s], realMaxA[s]);
  servo[s].write(x);
}
```

Figure 49: Angle Mapping Function

These functions are combined in a position foot function, which takes the Cartesian coordinates for the desired foot position, calculates the ideal joint angles, then maps the real angles to the servo motors.

```
/* posFoot moved the foot position to the desired x,y,z coordinates by retrieving the servo angles using IKtrig() then mapping to the true servo angle values */

void posFoot(int x, int y, int z)
{
  tp[X] = x;
  tp[Y] = y;
  tp[Z] = z;
  IKtrig(x, y, z); //retrieve servo positions
  for(int i = 0; i < numServos; i++)
  {
    mapWriteServo(i); //map and write true servo positions
  }
}
```

Figure 50: posFoot() function

8.4 AEP/PEP Swing/ Stride Cycle

To implement the trajectory control for each leg, first the Cartesian coordinates (relative to the Coxia joint) for start and finish point of the stride motion are retrieved. This function takes the stride length and the angle of travel, and calculates the AEP and PEP as per equation 8 – 12. The variable dir is defined as 1 for Forward and -1 for reverse – this has the effect of inverting the AEP and PEP positions, and functions as the direction trigger.

```
//function retrieves AEP and PEP x and y values fom direction angle

void getAEP PEP(int l, int a) //l - length of stroke - a - angle of direction in degrees
{
  a2 = a * rd; // convert a to Radians for trig
  aep[X] = np[X] + dir * (0.5 * l * cos(a2));
  aep[Y] = np[Y] + dir * (side * 0.5 * l * sin(a2));
  aep[Z] = np[Z];
  pep[X] = np[X] - dir * (0.5 * l * cos(a2));
  pep[Y] = np[Y] - dir * (side * 0.5 * l * sin(a2));
  pep[Z] = np[Z];
}
```

Figure 51: Function for retrieving AEP, PEP

For normal walking motion, a standard stride length of 100mm was chosen, as it allowed ±20% in most direction for step correction, and an IK step count of 30 allowed for reasonably smooth motion, while still executing the functions faster than the servo reaction time. A delay of 8 – 10 ms was experimentally determined to allowed the leg to sweep through the target range while keeping up with the function execution time. This IKdelay is kept constant for the swing phase.

To increment along the desired foot trajectory, Δx and Δy are calculated using equations 13 – 14. In order to avoid rounding errors, all the calculations in this process had to be done with floats.

```
  dx = (aep[X] - pep[X]) / float(steps);
  dy = (aep[Y] - pep[Y]) / float(steps);

  float cx = aep[X]; //reduce rounding errors
  float cy = aep[Y];
...
while (striding)
  {
    st++; //count steps used

    posFoot(tp[X], tp[Y], tp[Z]);

    cx -= dx;
    cy -= dy;
    tp[X] = cx ;
    tp[Y] = cy ;
    delay(del);
...
```

Figure 52: IK trajectory following code

8.5 Duty Factor

The Duty factor was set as an integer between 0 and 4 * the IK delay value. The IK delay for the stride cycle is IKdelay + Duty Factor. The delay on the lateral reflex signal is defined using the duty factor as a percentage of the stride path: Lateral Delay = steps * Duty Factor / 100

So as discussed in chapter 4, the duty factor defines the velocity, the gait, and the leg sequence of the hexapod.

8.6 Swing Trajectory

For the swing leg lift trajectory, an elliptical path was used, with Δz calculated using equation 16. After experimenting with a number of methods, a swingTo(int x, int y) method was settled on. Rather than relying on the AEP and PEP coordinates, this function has more universality in its application as it will step along an elliptical path from the current foot position to any designated target – it means that it will adapt the path if the return stroke is excited early, or if it is delayed.

```
void swingTo(int x, int y)
{
…
  int hlfS = stp / 2;
  float EllipseLift;
  float a;
  lift = 30;
…
  for(int i = 0; i < stp; i++)
  {
    posFoot(tp[X], tp[Y], tp[Z]);

    a = (hlfS - float(i + 1)) / hlfS;
    a = a * a;
    if (a > 1)
    {
      a = 1;
    }

    EllipseLift =  sqrt(1 - a) * lift;

    cx += dx ;
    cy += dy ;
    tp[X] = cx;
    tp[Y] = cy;
    tp[Z] = np[Z] - EllipseLift;
    delay(IKdelay);
```

Figure 53: Elliptical path code

8.7 Walk Cycle

walk() is the function called while the hexapod is in motion. The start position and state is set by a Switch/Case operation determined by the leg identity. It then sets the "walking" flag to true and cycles through swing() and stride() phases until the process is terminated.

At the start of each cycle, it checks for new steering and duty factor values and recalculates AEP and PEP accordingly, so the leg is capable of following steering function changes without interruption. The leg can also gain "confidence" in its gait by decrementing the duty factor – with all the legs doing this simultaneously, the hexapod transitions in a continuous fashion from pentapedal to tripod gait and from a slow cautious walk to a fast trot.

When walk() is terminated, the leg returns to its neutral position and all steering variables are reset.

```
 while (walking)
  {

    getAEP PEP(strokeL, travelAngle);
    run = true;
    swingTo(aep[X], aep[Y]);


    if (walking) //in case a halt is called
in swing
    {
      stride();
    }
 }
  run = false;
  delay(leg * 100);
  swingTo(np[X], np[Y]);
  delay(100);
```

Figure 54: main walk() loop

8.8 Reflex Behaviour

For the legs to operate largely autonomously from each other, while still generating stable and fluid gaits, the control system relies on feedforward signals between the legs according to their walk cycle states. Examination of Wilson's diagram of common insect gaits *(Wilson, 1966)*, the step sequence is propagated from the rear leg forward, as each leg completes its swing or return phase and takes load, the next leg forward commences its return phase. The step sequence is also propagated laterally between pairs of legs with a delay of: $t_{ld} = (\beta - 0.5)T$. These observations of gait sequence propagation are also reflected in the coordination influences used in WalkNET, where the influences propagate forward and laterally. In the code, this is approximated with a integer Duty Factor value of $0 \leq \beta \leq 4 * IKdelay$, and is

used as a percentage of transit through the stride phase.

When designing the control system, it was decided that the desired outcomes could be achieved using only the first to coordination rules from the WalkNet system, and that these rules could be combined into a single signal. To implement these then, the rear right leg was designated as the driving leg, with a lateral and forward reflex signal out, which would be set High for the inhibition influence, while the leg was in its swing phase, with the start of the power phase excited on a falling signal on the input pin. Ideally, the start of the power stroke will be when the leg reads that it is load supporting from the force feedback sensor, at the time of writing, it is determined only in the software. As it's important that the reflex actions are executed as soon as they are signalled, and that they not be missed due to the software loop cycle or delays, the "excite" influence is set as an interrupt function. When triggered, it sets the "striding" flag to false, ending the striding loop and triggering the swing function, so long as the leg is in walk() mode.

```
  pinMode (LAT IN, INPUT);
  pinMode (FOR IN, INPUT);

   PCintPort::attachInterrupt(FOR_IN,
&reflex, FALLING);
   PCintPort::attachInterrupt(LAT IN,
&reflex,                          FALLING);
...

void swingTo(int x, int y)
{
...
 digitalWrite(FOR_OUT, HIGH);
  if (leg == 6)
  {
    digitalWrite(LAT OUT, HIGH);
  }
...

void stride()
{
...
  int   ls   =   steps   *   duty   /   100;
  //time for lateral signal from R3
...
     if (st >= 2)
      {
        digitalWrite(FOR OUT, LOW);
      }
    }
    if (leg == 6) //driving leg R3
    {
      if (st >= ls)
      {
        digitalWrite(LAT OUT, LOW);
```

```
      }
...
//Interrupt Pin Functions

void reflex()
{
    striding = false;
}
```

Figure 55: Reflex influence code

8.8.1 Foot Feedback Reflexes

As well as responding to the feed forward signals from other legs, the leg also responds to external feedback from sensors in the foot (Figure 57). Each foot has a pressure sensor - a force sensitive resistor mounted on the end of the foot – and a bump sensor – a momentary switch on the front of the foot.

When the bump sensor is triggered during the swing phase, the leg controller calls the "step over" reflex function – moving the foot position reverse and up, then forward and down – in order to step over small obstacles or steps – the function adjusts the step height up 20mm, then readjusts.

```
void stepUp()
{
  int exs;
  int wiy;
  int zed;

  zed = tp[Z] - 15;
  exs = tp[X];
  wiy = tp[Y];
  exs -= 6 * dx;
  wiy -= 6 * dy;

  posFoot(exs, wiy, zed);
  delay(IKdelay);
  delay(IKdelay);

  zed = zed - 5;
  exs += 7 * dx;
  wiy += 7 * dy;

  posFoot(exs, wiy, zed);

  delay(IKdelay);
  delay(IKdelay);

  tp[X] += 4 * dx;
  tp[Y] += 4 * dy;
  tp[Z] = zed;
  lift += 20;
}
```

Figure 56: stepUp() function

The force feedback sensor in the foot allows the leg to sense when it had made contact with the ground and is load supporting, triggering the end of the

swing phase, it also allows the z positions to be adjusted downwards or upwards to adapt to topographical variation when traversing unstructured terrain. It also enables a pothole avoidance reflex method to be implemented, as if the foot fails to find firm ground after a reset increase to z, the leg steps up to np[Z] – 10, then calls swingTo() a point 15 – 20mm ahead of its AEP coordinate. If the foot fails to find solid ground after 2 iterations of this reflex, it then calls a halt to the walk() cycle, returns to its neutral position and awaits new navigation instruction from the motor cortex.

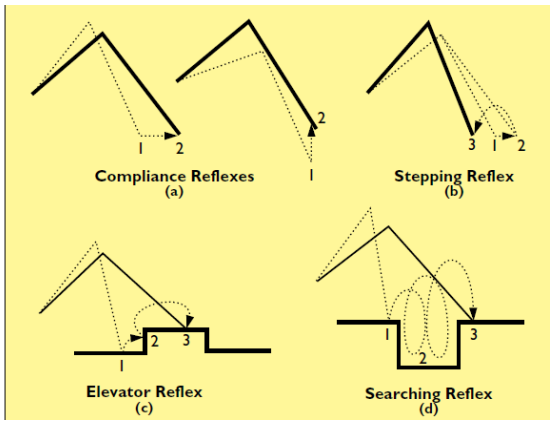


Figure 57: Leg Corrective Reflexes (Beer et al, 1997).

At the time of writing, a sketch for this function has been written but not fully tested or implemented.

8.9 Direction and Speed Control

As already discussed, the leg controller is able to implement directional control with the dir variable, speed control by adjusting the duty factor variable, and omnidirectional travel by adjusting the travelAngle variable, and is able to adjust all of these incrementally while executing the walk() function.

In addition to these, the leg controller is also able to implement Ackerman steering while in a forward motion, and on the spot rotation. The Ackerman steering method can be implemented while the walk() loop is running, the rotation method is a separate state called while stationary, which calls walk() within the function.

8.9.1 Rotating

When rotating, the leg controller starts from stationary, and sets the appropriate leg direction of travel and travel angle, as derived in chapter 4. The rotating motion can be set for a discrete number of steps, or continuous until an override signal is sent from the motor cortex.

```
void turn(int dr, int cyc)
//direction,steps
{
 mode = 4;
  cycles = cyc;

  switch(leg)
  {
    case 1: //L1
    {

      travelAngle = -48;
      dir = dr * (-1);
    }
    break;
…
    break;
    case 5: //R2
    {
      travelAngle = 0;
      dir = dr;
    }
    break;
    case 6: //R3
    {
      travelAngle = -48;
      dir = dr;
    }
    break;
  }
  walk();
}
```

Figure 58: turn() function

8.9.2 Ackerman Steering

The final steering function implemented by the individual leg is Ackerman steering. The leg can adjust its AEP and PEP to accommodate for a turning circle radius range of 500mm to 2500mm, controlled by an Ackerman variable with a range of -100 to 100, where -100 represents a turning circle of radius 500 in a clockwise direction. The Ackerman function is only called if the Ackerman variable does not equal zero. At the time of writing the Ackerman code has been written, but not fully implemented and tested. The function is primarily built using equations 35 – 37.

```
void ackerman()
{
  int radius; //radius of travel of outer
centre foot
  int r;    //radius of travel of this foot
  int d;    //perpendicular offset of this
foot
  int dr = 1; //+ or - direction of foot
travel
  float b;  //holder for offset for maths
  float a;  //holder for trig angle
  int L = setStride; //outside centre leg
stroke length
  radius = 480 + (abs(acker) * 20); // find
desired turning raduis

  switch(leg)
  {
    case 1: //L1
    {
      if (acker > 0)
      {
        d = 328;
        dr = 1;
      }
      else
      {
        d = 85;
        dr = -1;
      }
    }
    break;
    case 2: //L2
    {
      if (acker > 0)
      {
        strokeL = L * (1 - (408 / radius));
        travelAngle = 0;
      }
      else
      {
        strokeL = setStride;
        travelAngle = 0;
      }
    }
    break;
    case 3: //L3
    {
      if (acker > 0)
      {
        d = 353;
        dr = -1;
      }
      else
      {
        d = 55;
        dr = 1;
      }
    }
    break;
    case 4: //R1
    {
      if (acker > 0)
      {
        d = 85;
        dr = 1;
      }
      else
      {
        d = 328;
        dr = -1;
      }
    }
    break;
    case 5: //R2
    {
      if (acker > 0)
      {
        strokeL = setStride;
        travelAngle = 0;
      }
      else
      {
        strokeL = L * (1 - (408 / radius));
        travelAngle = 0;
      }
    }
    break;
    case 6: //R3
    {
      if (acker > 0)
      {
        d = 55;
        dr = -1;
      }
      else
      {
        d = 353;
        dr = 1;
      }
    }
    break;

    if ((leg != 2) && (leg != 5)) // if not
centre legs
    {
      b = radius - d;
      r = hypot(178, b);
      strokeL = L * r / radius;
      a = atan2 (178.0, b);
      travelAngle = a * dg * dr;
    }
  }
}
```

Figure 59: ackerman() function

8.10 Further Software Development

While the focus of this investigation was to design and test a biologically inspired distributed control system, and sought to demonstrate that a polypedal robot with this type of controller running on each leg would display stable and versatile gait generation despite the absence of a central controller and each leg acting largely independently, the purpose of the control system was to be eventually integrated with higher performance micro controller(s) and processor(s) handling navigation, route planning, steering control, body pose, obstacle avoidance, and even more computationally complex functions such as odometry, internal body simulation, Computer Vision, and Simultaneous Localisation and Mapping (SLAM).

Some preliminary research, development and testing was conducted in these area, laying the groundwork

for future expansion, but not complete code was compiled at this stage

8.10.1 Processors

Initially it was thought that the route planning, steering control, odometry, obstacle avoidance and internal body model simulation could all be managed by a microcomputer such as a Raspberry Pi, which would interface with IMU and range sensors, collect leg position data from the leg controllers, and send steering control information to the leg controllers, possibly using a second microcomputer to handle the more computationally demanding graphical processes such as visual SLAM. Robot Operating System (ROS) was investigated as a networking solution for this configuration, with the on board software for the Raspberry Pi written in Python.

A Raspberry Pi 3 was configured for this purpose, and minimal version of ROS indigo was successfully installed on the Pi. After spending some time familiarising with the ROS environment, and testing some basic Publisher/Subscriber communication between the Pi and an Arduino Mega, several issues became apparent that resulted in this aspect of the project being shelved for now. ROS is quite a complex development environment, with the option of implementing a wide range of high end function, but as such it is much more time consuming to learn and configure, and this was impacting adversely on the project schedule. It also became apparent that the Pi was not as suited to interfacing with sensors and actuators as micro controllers such as the AVR based Arduino boards, also ROS and the Pi microcomputer were less well suited for real-time reactive systems.

This configuration still has a lot of promise for later integrating more complex functionality to the hexapod, but a different approach for implementing basic navigation, sensor reading, body pose and obstacle avoidance functionality was then pursued. It was decided that a more efficient approach was to use the ATmega 2560 based Arduino Mega as the robotic platform's "Motor Cortex", handling input from range sensors and an IMU, sending steering variables to the leg controller and retrieving position and state information from the leg controllers. Other functionality investigated, but yet to be implemented, was Bluetooth capability for manual RC control from a mobile device, and leg controller override for more complex "deliberated" manoeuvres and for body pose operations.

8.10.2 Network Communications

For managing data communication between the Motor Cortex and the leg controllers, a couple of options were considered. I2C communication was investigated at the easiest to implement and most practical for multiple devices, with some test code written and tested for communication between the Arduino Mega and the Nano leg controllers. The main drawback with this approach was that it uses a Master/Slave configuration, meaning that for the leg controllers to return data to the Motor Cortex, it has to be requested by the Motor Cortex acting as Master. To get around this, the leg controllers flag the motor cortex with a digital signal pin when it wants to send data.

ROS was also considered as possible good networking option for exchanging control, position and state data. The advantage of using ROS is that it is designed for distributed computing for just this type of application, and utilises a number of different communications, including synchronous Remote Procedure Call (RPC) -style communication over services, asynchronous streaming of data over topics, and storage of data on a Parameter Server. As such it is much more versatile for sending and receiving state and position data and control variables throughout the control network. It's main disadvantage at this stage of the project development is it needs a processor at least as capable as the Raspberry Pi to act as a Master server, and to run the main ROS program, and that ROS is not a realtime platform, which could be a problem for a reflex oriented control system such as this one. Also the complexity of the ROS environment meant that the learning and configuring time needed to implement it was impractical for the restrictions on this project.

ROS was also being investigated as an option for implementing SLAM and computer vision capabilities at a later date. It is apparent that the

ROS visual processing suite, R-Vis, is too demanding to run on the Raspberry Pi 3. It may be possible to run it on a higher performance board such as the Odriod XU4, though this then puts increased demand on the power requirements. One approach was to research whether a simplified application of these functions could be custom built for the Pi using OpenCV and OpenGL – such an approach is beyond the scope of the current project however.

Finally, odometry poses a particular set of problems for a legged walker, particularly one with a decentralised and largely decoupled control system, as it is impossible to accurately track position, velocity and orientation using a purely geometric approach relying on the servo control data. A suggested approach would be to combine geometric odometry, sensor data from the on-board IMU and visual odometry. Both the visual odometry and geometric odometry are computationally demanding, and may need the capabilities of multicore 32 or 64 bit system with a GPU such as the Pi 3 or the Odroid boards. The full leg controller code can be found in Appendix VI: Leg Control Program.

9 Results

9.1 Hardware Issues

The majority of the development time on this project was taken up with the technical demands of building a functioning platform in order to conduct the experiments, and in fine tuning the control code. The decision to use budget servos compounded this factor, as a lot of time was spent calibrating then replacing and recalibrating failed servomotors. There were some issues earlier in the prototyping stage with a couple of the Nano controllers developing internal short circuits, probably due to a stray jumper wire during breadboard testing.

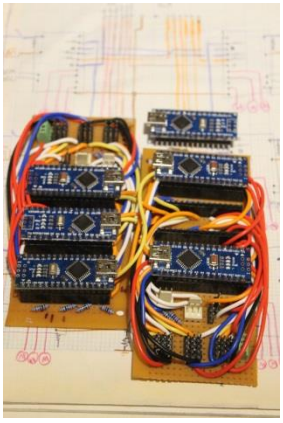


Figure 60: Assembled prototype control board

One of the major factors to impact the production schedule was the multiplying factor of using 6 controllers to control 18 servo motors, which meant that any software testing or hardware calibration had to be repeated at least 6 and often 18 or more times.

9.2 Running results

At the time of writing, only a few full cycles of the complete control system have been completed, but the results achieved have been very promising, and the issues that have been observed appear to be easily remediable through fine tuning of the code parameters. Building the functionality of the control system will also improve its robustness.

A couple of tests were executed with the robot untethered and supporting its own weight, including power supply. As anticipated from the previous analysis, the servomotors struggled to support the load, resulting in 3 servo motors shorting out after 5 walk cycles.

All subsequent tests were conducted with the hexapod supported, the body height was set just short of ground level, so that the legs could move freely, but their IK tracking could also be accurately observed, using the physical calibration map as a guide. This also meant that the force feedback algorithms could not be properly tested, as the legs were not under load. The bump sensor code was tested by holding various small obstacles in the bath of the leg swing trajectory. The stepUp() function appeared to work very effectively.

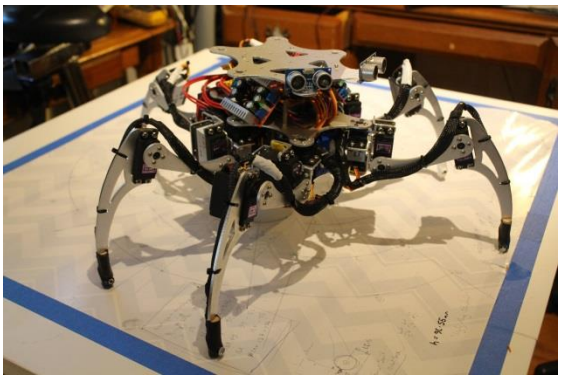


Figure 61: ArthroBOT, at the time of testing

In order to test the control system and its effectiveness in spontaneous gait generation, a number of sequences were written into the leg control code running independently on each leg. The sequences tested: a straight walking routine with the Duty Factor set at ~75%; A walking routine with the Duty Factor cycling between ~90% to 50% in increments of 4%; an omnidirectional test, with the angle of travel incrementing from 50° to -50°; and a routine to test the turn() function, turning 8 cycles to the left, then 8 cycles to the right.

As well as being unable to effectively test the force feedback functions, the Ackerman steering functional also remains to be tested, and with no central controller attached, and without it running independently on the ground a more comprehensive test of how the control system performs while navigating, receiving steering commands and reacting to the terrain still has to be conducted.

9.3 Observations

Overall, and from the limited number of test cycles conducted, the results were very positive, and in some cases exceeded expectations.

With all the walk cycles, the legs would achieve a regular stable gait pattern within 2 complete walk cycles, with the sequence largely established after one. When incrementing the Duty Factor, the hexapod transitioned in a smooth, continuous motion from a pentapedal gait through to a tripod gait and back again, with the contralateral timing between the back legs readjusting itself to suit the slower gaits. The Omnidirectional sequence also appeared to transition smoothly, with the foot trajectory tracking IK seeming to perform well, and the legs coordinating their direction of travel despite not having a central controller coordinating their steering parameters. Turning on the spot also appeared to be functioning correctly, though with the last two direction dependent functions, the true performance can only be tested with a self-supported locomotive test.

During the straight walking cycles, the bump sensor reflex was also tested, with various obstacles put in the leg swing trajectory path of several of the legs. The foot appeared to step successfully over the obstacle and continue its cycle, and the disruption to the leg would cause some noticeable but minimal disruption to the overall gait sequence, but this disruption would correct usually within two walk cycles.

The gait sequence would at times become unstable when at its fasted, tripod gait, the disruption would usually originate in the fore legs, and would take 2 – 3 walk cycles to correct itself. Also when the bump sensor was triggered at these faster gaits, the disruption to the gait sequence would be more pronounced. These disruptions did not appear to be fatal in the initial testing, though they also need to be observed when the legs are under load, and there are likely conditions when they become a more pronounced problem. They do indicate that the control system can be improved still, and that these disturbances could be mitigated either by adding extra functionality to the code to recognise and adapt to the conditions that cause the leg sequence to slip, or by adding extra reflex signalling between the legs, perhaps with additional contralateral signals between the middle and for legs, or a return signal from the fore legs to the rear legs.

During earlier tests, an issue would sometimes occur at the termination of a walk cycle, where the fore legs would miss the termination signal, and get stuck on a recursive loop at the end of the stride phase, this was seemingly fixed by adjusting the behaviour and parameters of how the legs handled and sent their reflex signals, and in adding some code to deal with certain corner conditions, but it is likely that undiscovered corner conditions still exist and will only become obvious with more extensive testing. adjustment and stepping over potholes are also not tested.

9.4 Further Investigation

More comprehensive testing needs to be done to get a true measure of the overall performance of the control system and its algorithms. The robots performance needs to be tested when mobile and underload, the more complex steering and navigation algorithms, particularly the Ackerman steering function are untested to date, and the functions linked with the force feedback sensor, such as foot height adjustment and stepping over potholes are also not tested.

The next stage of testing then requires that the robot be run in a number of environments and scenarios, such as detecting and navigating around larger obstacles, adapting to variations in unstructured terrain, and pursuing goal oriented tasks that involve using combinations of its control functions.

9.4.1 Actuators

To achieve this, first the underpowered and underperforming budget servomotors need to be replaced with servos that meet the desired specifications. The minimum requirement would be ≥ 15 kgf.cm for the tibia servo, ≥ 25 kgf.cm for the femur servo and ≥ 20 kgf.cm for the coxia servo. Using lower cost servos, this could be done for around $400 (Ali Express 2016)

Figure 62: Dynamixel RX 24 smart servo (Robotis 2016)

The preferred approach would be to replace the existing servos with thigh performance servo motors such as the Dynamixel series from Robotis. These are fast acting robust servos, custom designed for robotic application, and is able to give speed, temperature, shaft position, voltage and load feedback, as well as having a range of additional control options such as adjusting the servo motors response speed and strength. This range of extra functionality would permit significant expansion of the control system to include PID control, more accurate position tracking, and load balancing across the legs. For this application, the preferred configuration would be the AX-18A for the Tibia servo, the RX-28 model for the Femur, and the RX-24 for the Coxia (Figure 62). High end actuators also means a high end cost, the cost for this configuration of Servo motors USD$1,870.75. (Robotis, 2016). The Dynamixel range specifications can be seen in Appendix IX: Dynamixel Smart Servos.

9.4.2 Increasing Sophistication

Once the basic control architecture has been thoroughly tested and tuned, further complexity can be added to it to seek further performance optimisations. The load distribution coordination rule would improve efficiency of navigation on unstructured terrain, and its implementation would be facilitated with the use of smart servos such as a the Dynamixel range. Neural Networks may add further advantage to the systems adaptability and optimisation, so this is another line of enquiry worth pursuing.

9.4.3 Motor Cortex

In order to be able to test more complex steering and navigation sequences, it is necessary to finish implementing the motor cortex, with the proposed obstacle detection using ultrasonic range finders, and body pose using the MPU 9150. For manual control a Bluetooth controller using an android device was instigated, this would be finished for further testing. Methods for handling cul-de-sacs and edges also need to be developed, as well as a leg control override by the Motor Cortex for more complex "deliberated" operations.

9.4.4 Hardware Improvements

Once the control circuit has been tested and finalised, a PCB design should be developed and manufactured. This would make for much neater and therefor more reliable wiring, and also allow for implementation on multiple platforms. To improve the performance and capabilities of the system, the ATmega 328 microcontrollers should be replaced with STM32 32bit ARM Cortex M3 micro-controllers.

9.4.5 Expandability and Redundancy

One of the key performance features of this control system is its ability to achieve a range of stable and coordinated gaits, while also synchronising steering behaviours and executing some obstacle mitigation methods, spontaneously, with no central controller, and minimal necessary coordinated communication between the separate leg controllers. Each leg is operating largely autonomously, running nearly identical code to the other five legs. The leg gait coordination also appears to automatically correct itself if it is disrupted or of a leg is hindered. This indicates that the control network could be configured such that if a single leg was to fail, the reflex signal can bypass that leg, allowing locomotion to continue even in the event of leg loss.

Further, it should be possible to add additional pairs of legs without significantly adding to the computational complexity of the controller, as the leg controllers would need no additional code, only communicating with their immediate neighbours, and the motor cortex sends the same steering parameters to all the legs regardless of their position. An "expansion" PCB housing two leg controllers with forward connections to the rest of the control network as well as connectors for the legs and sensors should be designed and assembled.

9.4.6 Cooperative Robotics

Expanding on this functionality, multiple hexapod units using this control network should be able to operate independently, then be able to couple together in a chain, handing steering control over to the forward most unit. This could have useful application in transporting multiple units between locations, where they would decouple and perform their intended operations, or they could use their combined body mass and locomotive capabilities to traverse larger obstacles such as larger steps or bridging gaps they would be incapable of crossing individually.

9.4.7 Scalability

This control network should work equally as effectively on larger scale robotic platforms such as the Comet IV Hexapod, as essentially the leg behaviours would remain the same. An experimental investigation into whether this control method could provide significant performance and efficiency gains at a larger scale. There may even be compelling reasons to experiment on a larger scale, as the power to rate ratio for the body and actuators may become less of a complicated issue as the scale increases.

9.4.8 Visual Cortex and Expanded Functionality

Once the leg controller network and steering and navigation control has been thoroughly tested, designed, configured and implemented, the ArthroBOT platform should function like any other programmable mobile robot platform, and be easily expandable in its functionality adding a higher performance microcomputer to handle functions such as AI operations, computer vision, visual odometry and SLAM. For these operations, the addition of an Odroid XU4 using ROS for inter processor communication should be implemented.

Odometry for a walking robot is particularly problematic, as it is both mathematically complicated to track absolute changes in body position and orientation through the movement of 6 legs, and physically improbable to be able to continuously and accurately log each relative foot position and velocity in order to make such calculations. A combination of visual odometry and geometric odometry coupled with body pose sensor date from the IMU might be a solution that could be investigated.

The distributed computing approach of this control network is another of its key efficiencies, so the Visual Cortex board would be able to dedicate its resources to necessary graphical processes only.

9.4.9 Proposed Applications

As the ArthroBOT network is low cost, lightweight, power and computationally efficient, adaptable to multiple geometries with minimal modification, and relatively easy to set up and easily expandable, while also demonstrating convincing robustness and adaptability in its performance, there would be numerous application for its use.

As a low cost lightweight controller, it could be applied to products for the hobbyist or RC toy market. As an expandable, adaptable and programmable platform it could be the basis for a range of robotic research and development platforms. Additionally with its scalability, robust performance, adaptability to leg loss, leg interruption and variable terrain, and the efficiencies it provides through distributed computing, and the potential to implement cooperative robotic arrays, it would be a good basis for commercial applications such as exploring and operating in dangerous and hard to navigate unstructured environments where wheeled mobile robots are not practical, such as natural disaster sites, hazardous zones such as Fukushima, and extreme environments such as exoplanet exploration or subterranean or volcanic zone surveying.

9.5 Conclusion

In one of his MIT lectures, the famous roboticist and AI pioneer, Marvin Minsky stated that the problem with AI and robotics researchers is that they waste all their time trying to build working robots (Minsky, 2011). With all of the technical challenges, numerous mechanical and electronic failures and many uncertainties in the development process, this has often seemed like one of the projects he described, requiring an inordinate amount of preparation, manufacturing, set up and

development before the core hypothesis of this project could be tested, all under restrictive time and budget constraints.

While the final platform was less than adequate, and the results obtained less than comprehensive, the results that were observed showed great potential for this biomimetically inspired, decentralised, distributed computation approach to polypedal mobile robot leg control and gait generation.

The project set out to demonstrate that a distributed control system, based loosely on an ants couple neural ganglia and drawing on the WalkNET system developed by Holk Cruse, could spontaneously generate variable but robust, adaptable and stable walking gaits analogous to those observed in insects, despite using only minimal signalling between individual legs and with each leg controller operating autonomously. Also, by treating the duty factor as a sliding scale, the different established insect gaits would emerge as a continuum coupled to the gait velocity rather than discreet conditions. The results obtained in some aspects the observed performance exceeded initial expectations, and demonstrated performance attributes that had not been entirely anticipated. The experimental and testing phase also indicated a number of possible research directions that further testing and development could be applied that also had not been apparent during the conceptual stages of the project.

References

Ackerman, Evan, 2012 "DARPA Awards Simulation Software Contract to Open Source Robotics Foundation" IEEE Spectrum 17th April 2012

Ali Express 2016, aliexpress,com accessed 12/10/16

Arduino 2016, http://arduino.cc/en/Main/arduinoBoardMega2560 accessed 12/10/16

Arena, Paolo, Luigi Fortuna, Mattia Frasca, and Giovanni Sicurella 2004"An Adaptive, Self-Organizing Dynamical System for Hierarchical Control of Bio-Inspired Locomotion" IEEE Transactions On Systems, Man, And Cybernetics—Part B: Cybernetics, Vol. 34, No. 4, August 2004 pp. 1823 – 1837

Australian Museum 2016 "What are arthropods" http://australianmuseum.net.au/What-are-arthropods viewed 16th April 2014

Beer, R.D., Chiel, H.J., Quinn, R.D., Espenschied, K.S., Larsson, P 1996 "A distributed neural network architecture for hexapod robot locomotion". Neural Computation 4, pp 356–365

Beer, R.D., Chiel, H.J., Quinn, R.D., Ritzmann R E , 1997 "Biologically Inspired Approaches to Robotics" Communications Of The Acm March 1997/Vol. 40, No. 3 pp. 30 - 38

Brooks, R.A .1986 "A robust layered control system for a mobile robot". IEEE Journal of Robotics and Automation RA-2, pp. 14–23

Brooks, R.A 1989 "A robot that walks; emergent behaviors from a carefully evolved network". A.I. memo 1091, Massachusetts Institute of Technology

Brooks, R.A. (1991)." Intelligence without reason." IJCAI-91, Sydney, Australia (pp. 569-595).

Cruse and Wehner, 2011 "No Need for a Cognitive Map: Decentralized Memory for Insect Navigation" PLoS Computational Biology 2011

Cruse H 2001 "The evolution of cognition—hypothesis" Elsevier, Cognitive Science 27 (2003) pp. 135–155

Cruse H, Durr V And Schmitz J 2006"Insect walking is based on a decentralized architecture revealing a simple and robust controller" Phil. Trans. R. Soc. A (2007) 365, pp. 221–250

Cruse, H 1976 "The function of the legs in the free walking stick insect, Carausius morosus". J. Comp. Physiol. A 112, pp. 235–262

Cruse, H., Brunn, D., Bartling, C., Dean, J., Dreifert, M., Schmitz, J.: 1995 "Walking – a complex behavior controlled by simple networks". Adaptive Behavior 3(4), pp. 385–418 (1995)

Cruse, H., Kindermann, T., Schumm, M., Dean, J., Schmitz, J.: Walknet - a biologically inspired network to control six-legged walking. Neural Networks 11(7-8), 1435–1447 (1998)

Cyberbotics 2016 http://www.cyberbotics.com/overview accessed 3/10/16

De Silva, Shalutha (2014) Force controlled hexapod walking. PhD thesis, Queensland University of Technology.

Dürr, V., Schmitz, J., and Cruse, H. (2004). "Behaviour-based modelling of hexapod locomotion: linking biology and technical application." Arthropod Struct. Dev. 33, pp 237–250.

electronicoscaldas.com, 2016, http://www.electronicoscaldas.com/datasheet/MG996R_Tower-Pro.pdf accessed 3/10/16

Fitcher, E. F. and Fichter, B. L. 1988 "A survey of legs of insects and spiders from a kinematic perspective," Robotics and Automation, 1988. Proceedings., 1988 IEEE International Conference on, Philadelphia, PA, 1988, pp. 984-986 vol.2.

Gorner, Martin and Gerd Hirzinger 2010 "Analysis and Evaluation of the Stability of a Biologically Inspired, Leg Loss Tolerant Gait for Six and Eight Legged Walking Robots" 2010

IEEE International Conference on Robotics and Automation

HardKernel.com 2016, http://www.hardkernel.com/main/products/prdt_info.php?g_code=G143452239825 accessed 3/10/16

Hoinville, T., Wehner, R., and Cruse, H. (2012). "Learning and retrieval of memory elements in a navigation task," Living Machines Conference (Barcelona), pp 120–131.

Kumar, Kishy 2011 "Analysis of Contemporary Robotics Simulators" IEEE Proceedings of ICETECT 2011

Leaf Labs 2016 http://www.leaflabs.com/ accessed 3/10/16

Lewinger, W.A. B.L. Rutter, M. Blümel, A. Büschges and R.D. Quinn, "Sensory Coupled Action Switching Modules generate robust, adaptive stepping in legged robots," in Proceedings of the International Conference on Climbing and Walking Robots (CLAWAR'06), pp. 661–671, Brussels, Belgium, Sept 12–14, 2006.

Lewinger, William A., Quinn, Roger D. 2009 "A Small, Autonomous, Agile Robot with an On-board, Neurobiologically-based Control System" The 2009 IEEE/RSJ International Conference on Intelligent Robots and Systems

Mantis Robot 2016 http://mantisrobot.blogspot.co.uk/ accessed 25/9/16

Micromo, 2016, http://www.micromo.com/technical-library/dc-motor-tutorials/motor-calculations accessed 18/10/16

Minsky, Marvin 2011, "The Society of Mind" MIT opencourseware lecture series, MIT 2011

Nardi, Luigi 2014 "SLAMBench, a performance and accuracy benchmarking methodology for SLAM" Imperial College of London, 2014

Nelson, G.M., Quinn, R.D., 2001. "A Numerical Solution to Inverse Kinematics for Swing Control of a Cockroach-like Robot" Proceedings of Climbing and Walking Robots Conference (CLAWAR' 01), Karlsruhe, Germany, September.

Nonami, Kenzo, Ranjit Kumar Barei, Addie Irawan, Mohd Razali Daud, (2014) "Hydraulically Actuated Hexapod Robots; Design, Implementation and Control" Springer, Intelligent Systems, Control and Automation: Science and Engineering 2014

Open Robots 2016 http://www.openrobots.org/morse/doc/1.2/what_is_morse.html accessed 3/10/16

Open Source Robotics Foundation, 2016 http://gazebosim.org accessed 2/10/16

Palmer III, Luther and Mayur Palankar 2011 "Blind Hexapod Walking Over Uneven Terrain Using Only Local Feedback" Proceedings of the 2011 IEEE International Conference on Robotics and Biomimetics

Parker G.B. and Li, Zhiyi, "Evolving neural networks for hexapod leg controllers," Intelligent Robots and Systems, 2003. (IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on, 2003, pp. 1376-1381 vol.2.

Pearson, Keir. The Control of Walking. Scientific American, vol 276. Dec. 1976

Pfeiffer, F., Eltze, J., & Weidemann, H.J. (1995). "Six-legged technical walking considering biological principle"s. Robotics and Autonomous Systems, 14, 223–232.

Pfeiffer, F., Eltze, J., Weidemann, 1994. "The TUM walking machine". Intelligent Automation and Soft Computing 2, TSI Press, Albuquerque, NM.

Porcino, N. 1990 "Hexapod gait control by a neural network," Neural Networks, 1990., 1990 IJCNN International Joint Conference on, San Diego, CA, USA, 1990, pp. 189-194 vol.1.

Raspberrypi.org 2016 http://www.raspberrypi.org/help/what-is-a-raspberry-pi/ accessed 12/10/16

Ritzmann, R.E., Quinn, R.D., Watson, J.T., Zill, S.N., 2000. Insect walking and biorobotics: a relationship with mutual benefits. BioScience 50(1), 23–33.

Robotis, 2016 http://robotis-shop.com/DYNAMIXEL-Smart-Robot-Actuator accessed 12/10/16

robotshop.com 2016, http://www.robotshop.com accessed 12/10/16

Roennau, A, T. Kerscher and R. Dillmann 2010 "Design and Kinematics of a Biologically-Inspired Leg for a Six-Legged Walking Machine" Proceedings of the 2010 3rd IEEE RAS & EMBS International Conference on Biomedical Robotics and Biomechatronics

Rosano-Matchain, Hugo Leonardo 2007, "Decentralised Compliant Control for Hexapod Robots: A Stick Insect Based Walking Model" PHD Thesis, University of Edinburgh HOBBY

Sawicz, Darren 2016 "Servo Fundamentals" Princeton University no published date, document accessed 10/10/2016

Schilling, M., and Cruse, H. (2008). "The evolution of cognition— from first order to second order embodiment," in Modelling Communication with Robots and Virtual Humans, eds I. Wachsmuth and G. Knoblich (Berlin: Springer), pp. 77–108

Schilling, M., Cruse, H., and Arena, P. (2007). "Hexapod Walking: an expansion to Walknet dealing with leg amputations and force "oscillations. Biol. Cybern. 96, pp 323–340.

Schilling, Malte , Jan Paskarbeit, Thierry Hoinville , Arne Hüffmeier, Axel Schneider, Josef Schmitz and Holk Cruse 2013 "A hexapod walker using a heterarchical architecture for action selection" Frontiers in Computational Neuroscience 2013

sparkfun.com 2016 www. sparkfun.com accessed 12/10/16

Tedeschi, Franco , Giuseppe Carbone (2014) "Design Issues for Hexapod Walking Robots" Robotics 2014 Vol 3 pp 181 – 206

Wait K W, Dalley S A, Goldfarb M 2008 "Design and Control of a Biomimetic Hexapedal Walker" IEEE/RAS- International Conference on Biomedical Robotics and Biomechatronics EMBS, 2008

Wilson D 1966 "Insect Walking" Annual Review of Entomology 1966

Appendices

Appendix I: Development Schedule

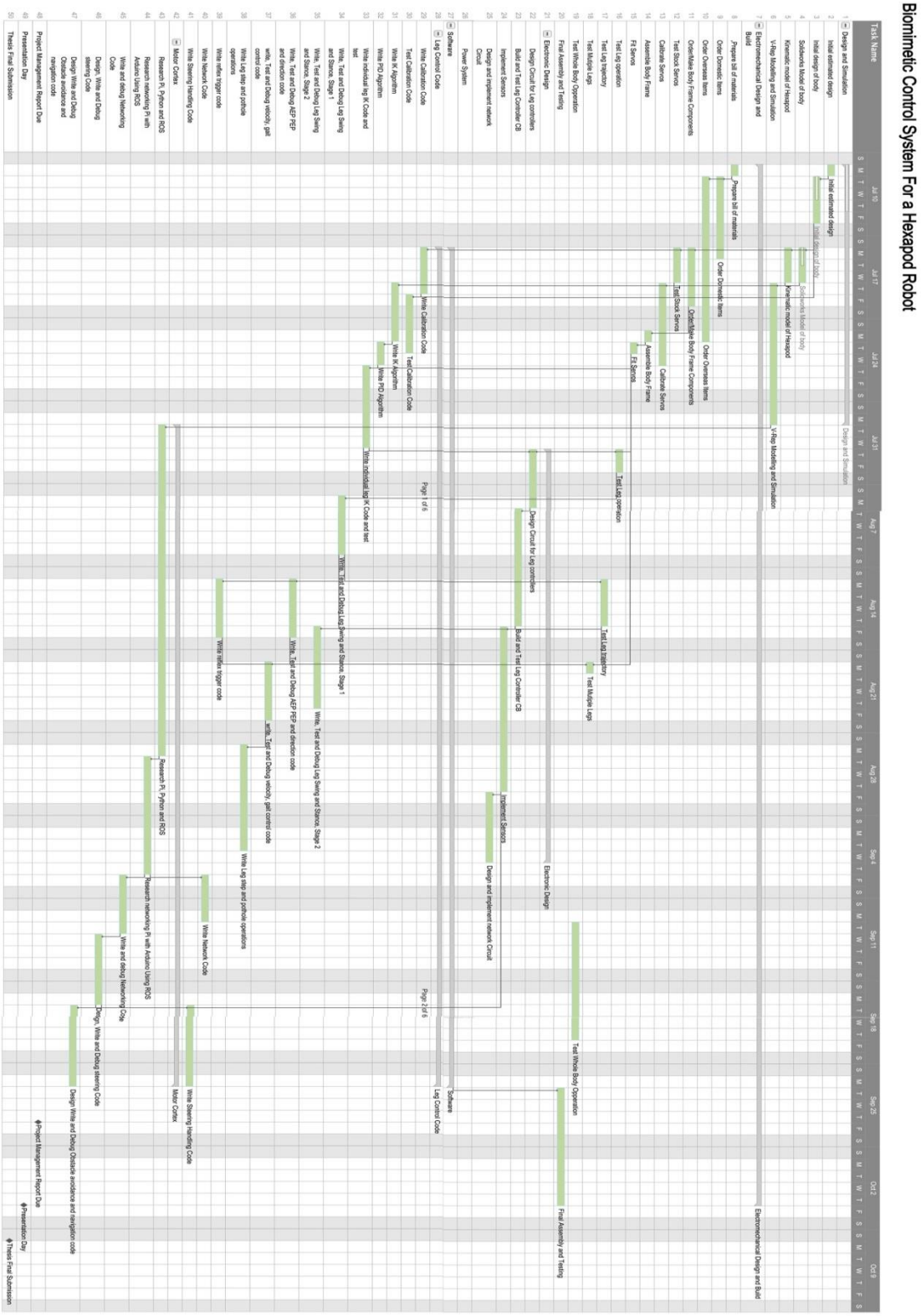


Figure 63: Schedule Gantt Chart

Appendix II: Control Diagrams

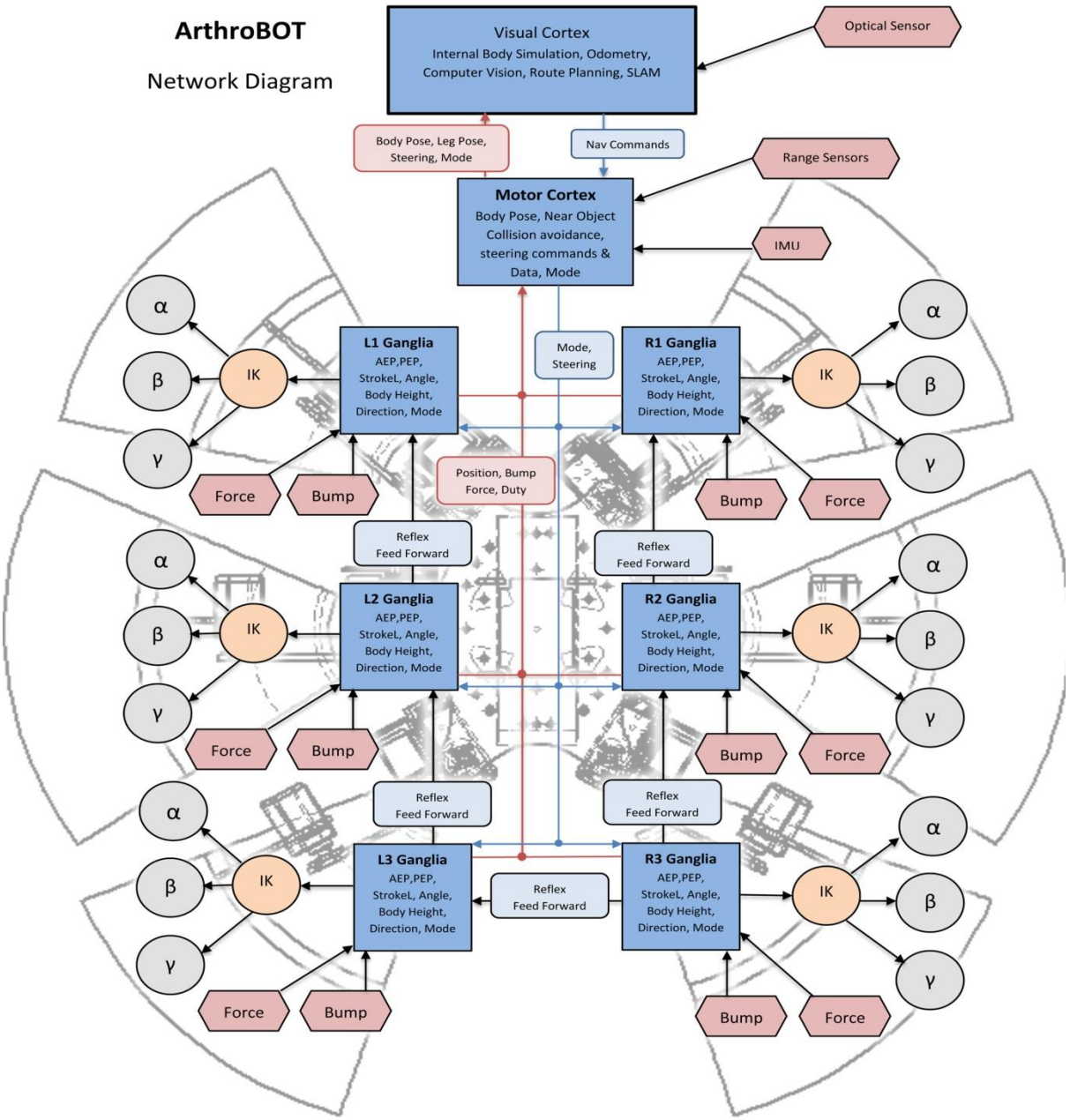


Figure 64: ArthroBOT Network Diagram

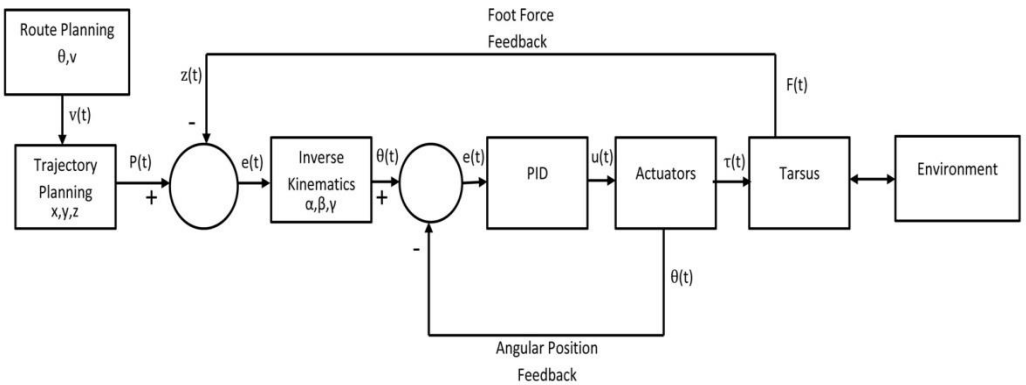


Figure 65: Leg Controller Block Diagram

Appendix III: Analysis Diagrams

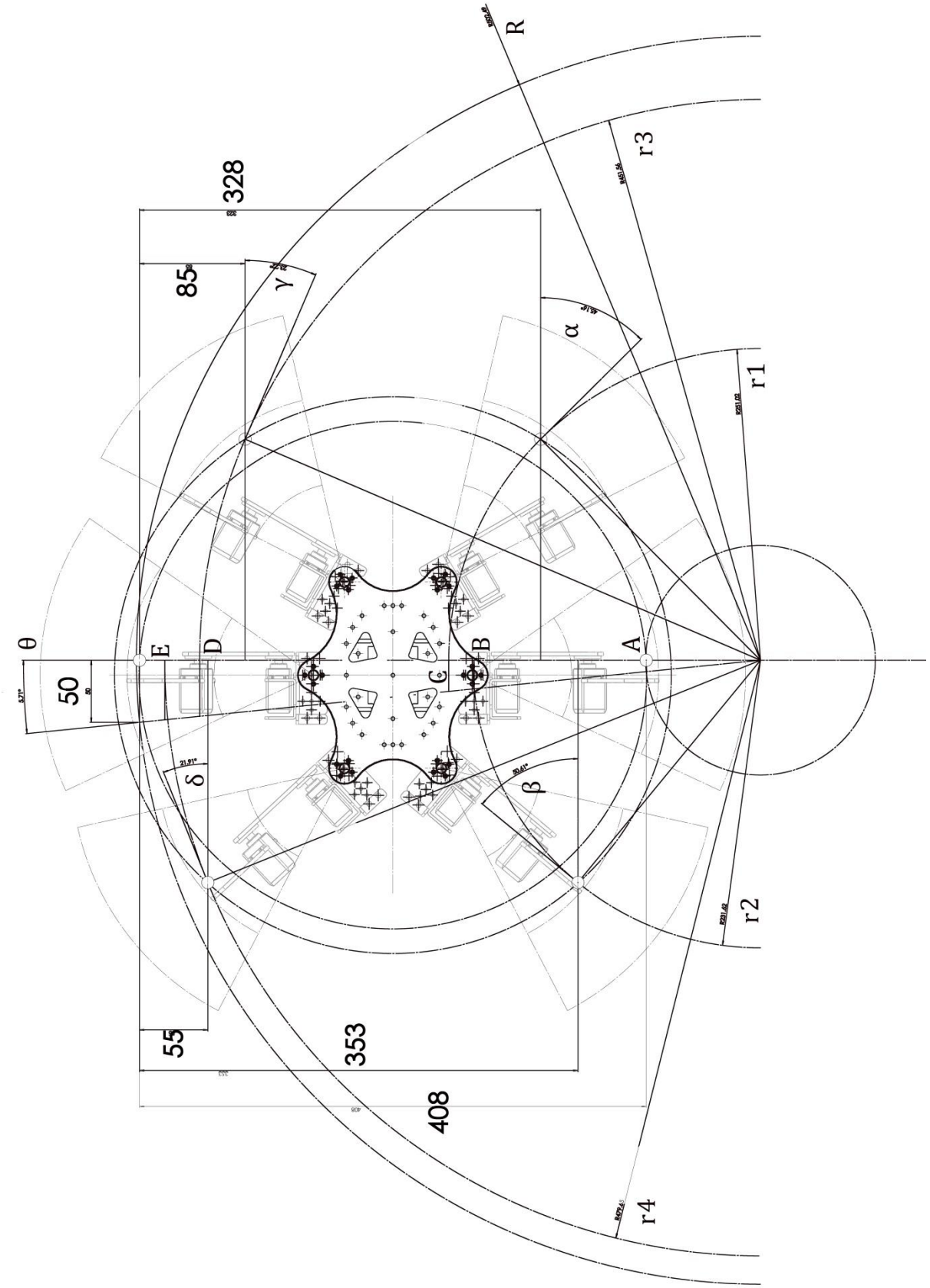


Figure 66: Ackerman Steering Diagram

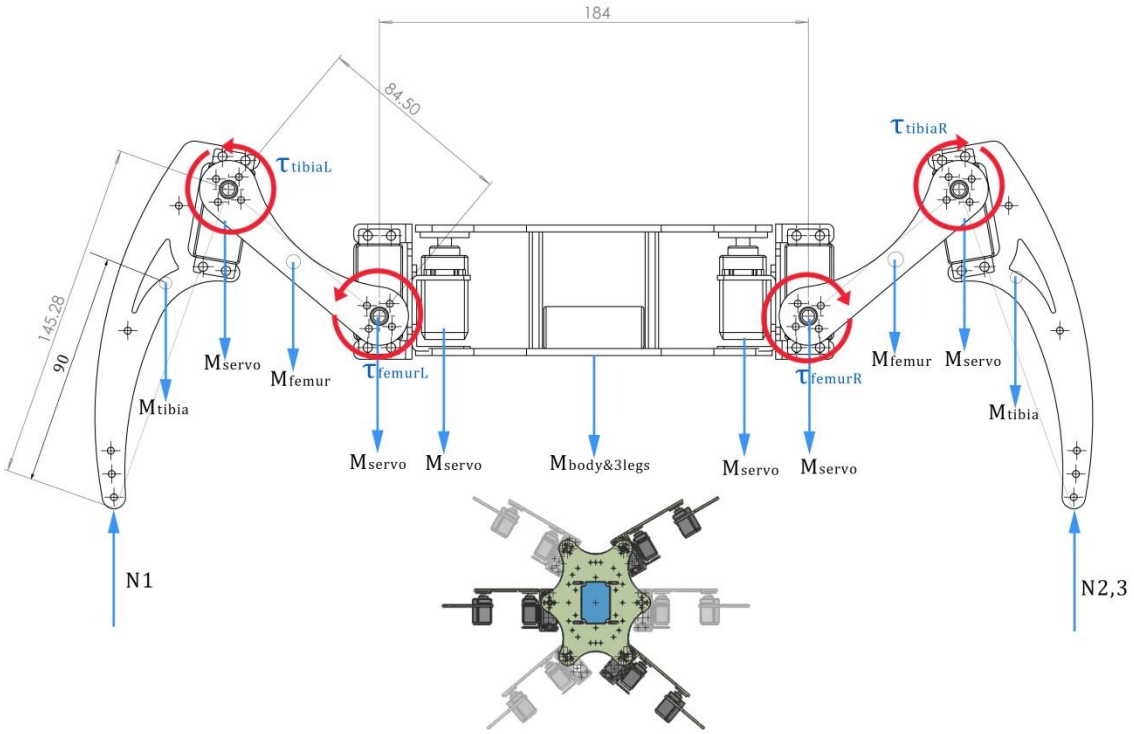


Figure 67: Hexapod FDB

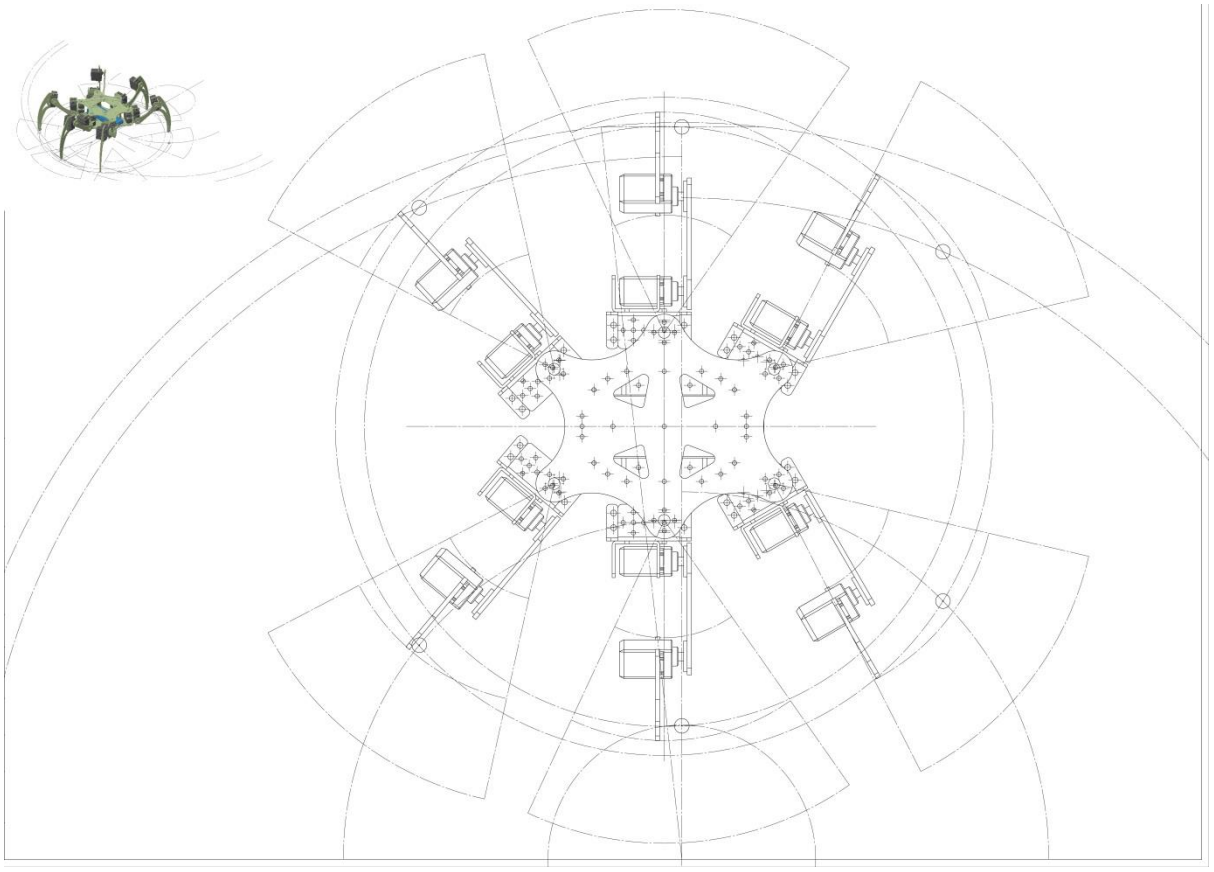


Figure 68: Physical Calibration Map

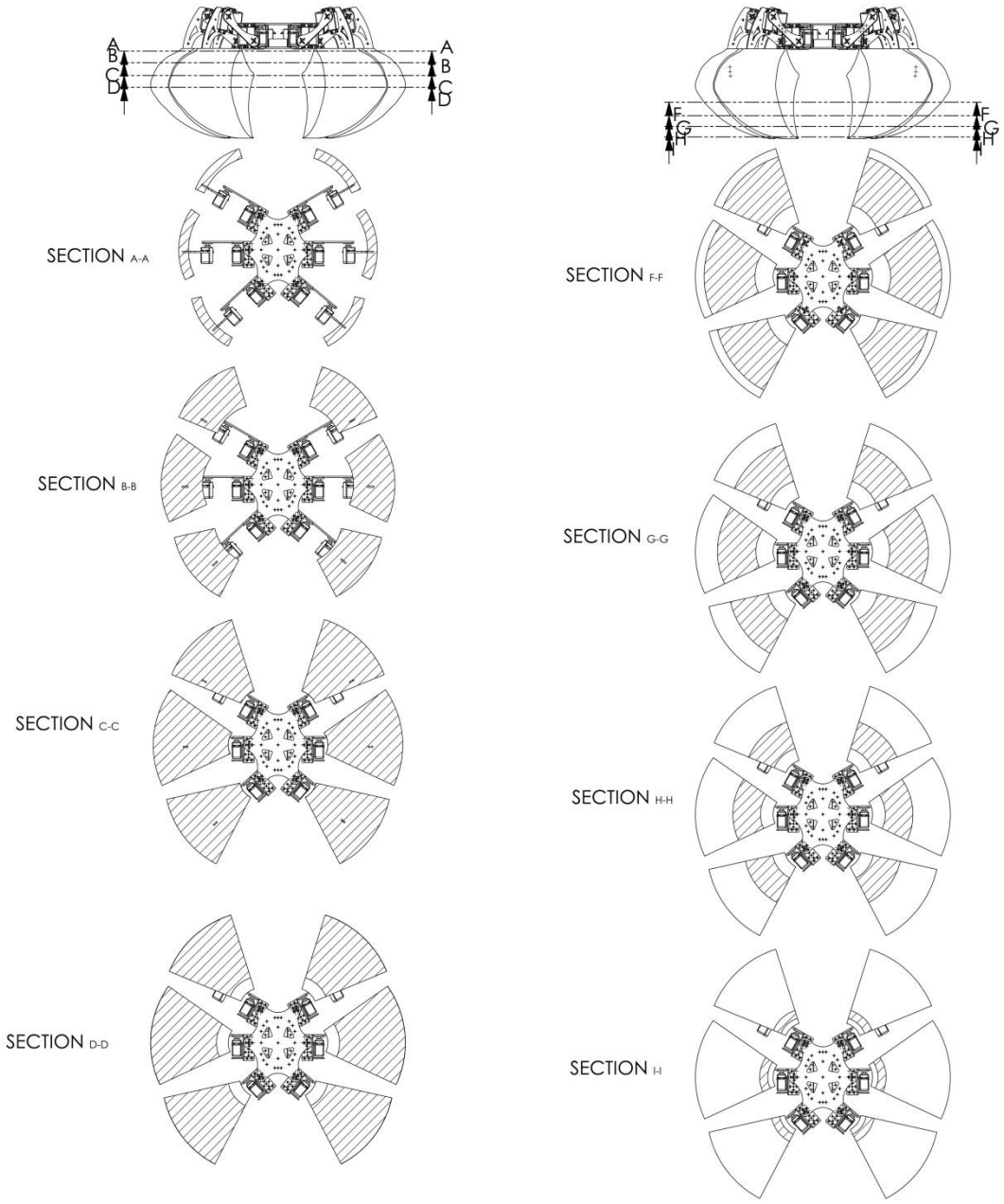


Figure 69: Workspace Analysis

Appendix IV: CAD Drawing

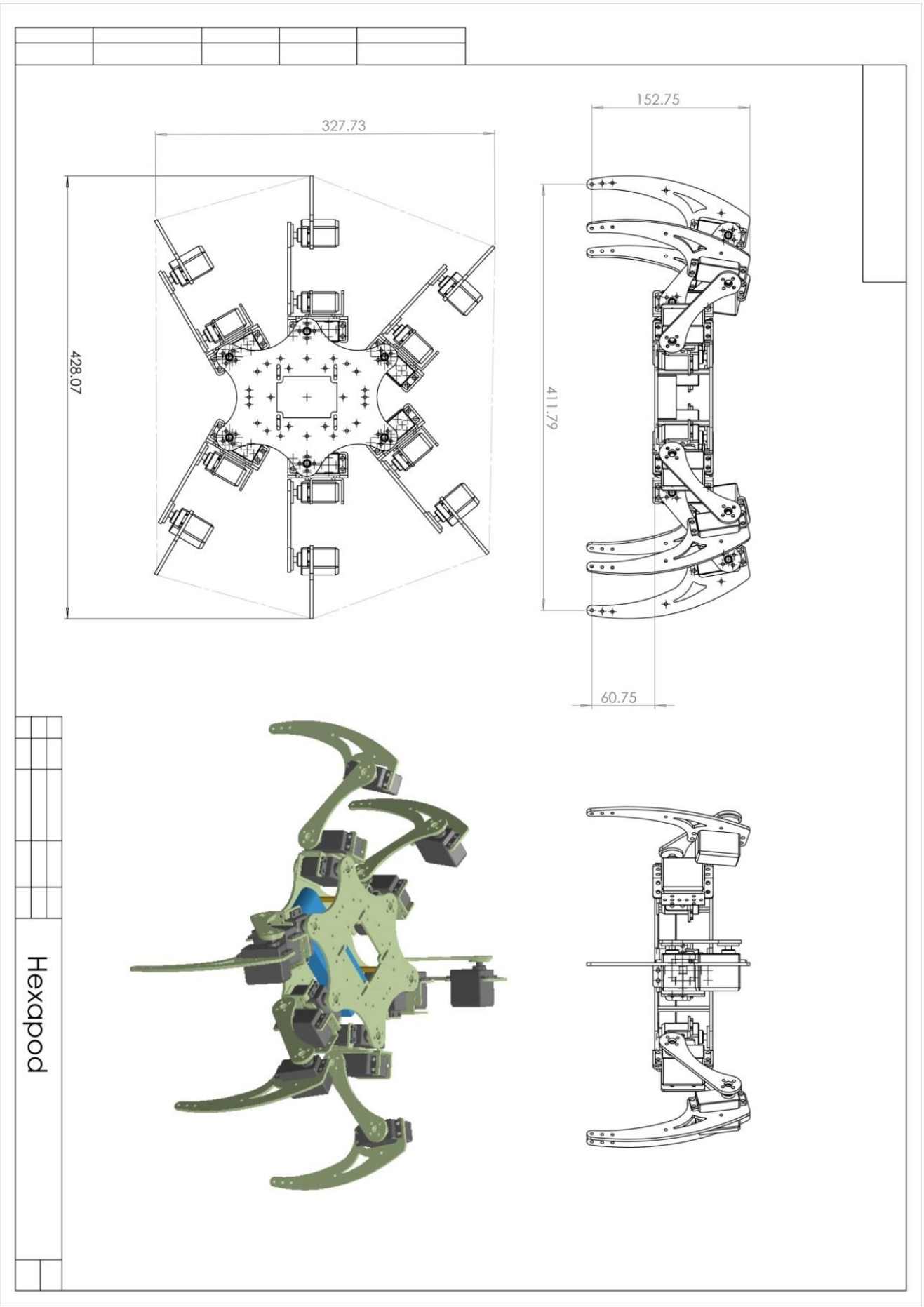


Figure 70: Hexapod CAD Drawing

Appendix V: Calibration Data

Joint	MinPulse	CenPulse	MaxPulse	180Pulse	90Pulse
L1 Coxia	550	1470	2390	2162	1356
L1 Femur	540	1465	2390	2252	1396
L1 Tibia	550	1430	2310	2166	1358
L2 Coxia	670	1477.5	2285	2016	1343
L2 Femur	540	1475	2410	2169	1355
L2 Tibia	550	1475	2400	2262	1406
L3 Coxia	630	1395	2160	2817	1724
L3 Femur	550	1470	2390	2229	1390
L3 Tibia	550	1470	2390	2229	1390
R1 Coxia	550	1470	2390	2184	1367
R1 Femur	560	1480	2400	2323	1441
R1 Tibia	550	1445	2340	2160	1355
R2 Coxia	550	1465	2380	2121	1335
R2 Femur	540	1440	2340	2117	1328
R2 Tibia	550	1480	2410	2148	1349
R3 Coxia	550	1405	2260	2175	1363
R3 Femur	550	1405	2260	2152	1351
R3 Tibia	540	1410	2280	2183	1361

	AngleRange	CenAngle(90)	ActAngleRange	MinAngle	MaxAngle	Low/Rear Extreme	High/Fore Extreme
L1 Coxia	199	113	50	100	150	103	53
L1 Femur	191	43	130	-40	90	7	137
L1 Tibia	192	71	110	25	135	44	154
L2 Coxia	204	92	50	65	115	117	67
L2 Femur	200	45	130	-40	90	5	135
L2 Tibia	191	66	110	25	135	49	159
L3 Coxia	138	118	50	100	150	108	58
L3 Femur	193	50	130	-40	90	0	130
L3 Tibia	193	68	110	25	135	47	157
R1 Coxia	197	124	50	100	150	66	116
R1 Femur	186	49	130	-40	90	179	49
R1 Tibia	195	68	110	25	135	133	23
R2 Coxia	202	95	50	65	115	60	110
R2 Femur	199	48	130	-40	90	178	48
R2 Tibia	202	71	110	25	135	136	26
R3 Coxia	187	124	50	100	150	66	116
R3 Femur	189	46	130	-40	90	176	46
R3 Tibia	188	67	110	25	135	132	22

Table 2: Electronic Servo Calibration

Leg	neutralX	neutralY	neutralZ	L Reach(mm)	Leg Angle
Front	-87	123.69	72	151.00	125.00
Centre	0	151	72	151.00	90.00
Back	87	123.69	72	151.00	55.00

Table 3: Leg Data

Limb	Length(mm)	Mass(g)
Coxia	28	180
Femur	86	90
Tibia	147	130

Table 4: Leg Joint Data

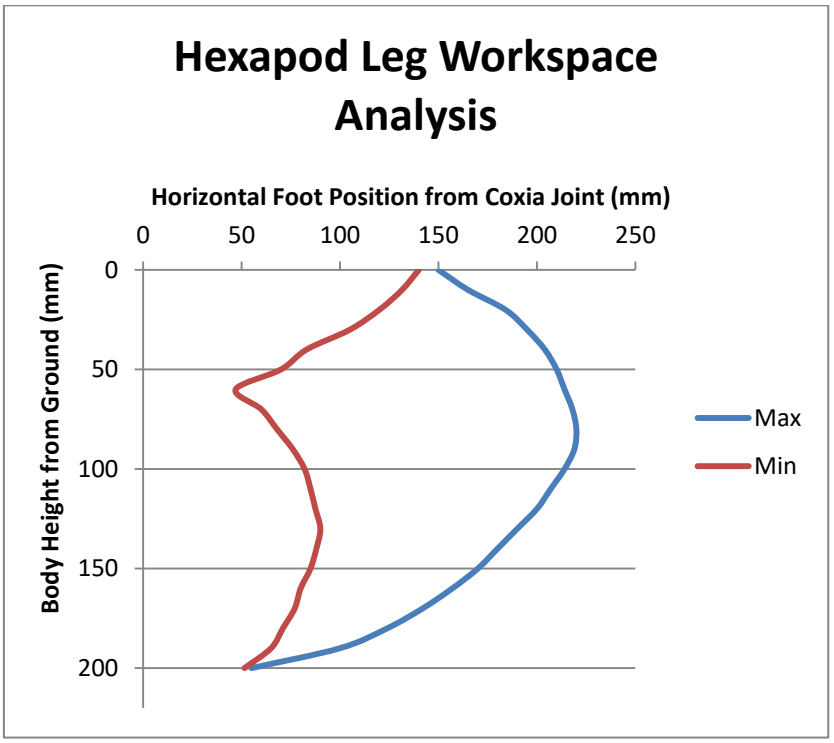
Leg	Coxia	C Pulse	Femur	F Pulse	Tibia	T Pulse
R3	Angle	Pulse	Angle	Pulse	Angle	Pulse
1	100	1575	104	1616	62	1183
2	58	1142	105	1626	64	1203
3	102	1595	110	1678	93	1507
4	58	1142	111	1688	93	1502
5	103	1606	137	1956	145	2039
6	58	1142	138	1966	145	2039
7	80	1368	111	1688	93	1502
8	81	1379	136	1946	145	2039
9(np)	83	1399	109	1667	88	1451
stow	51	1609	54	1100	16	708
low	11	657	13	678	13	678
high	173	2327			179	2389
R2						
1	100	1575	96	1533	67	1234
2	44	997	95	1523	66	1224
3	101	1585	98	1554	98	1554
4	46	1018	99	1564	98	1554
5	101	1585	123	1812	147	2059
6	46	1018	123	1812	146	2049
7	73	1296	100	1575	98	1554
8	73	1296	124	1822	147	2059
9	72	1286	98	1554	92	1492
stow	73	1296	49	1049	28	832
low			11	657	28	832
high					179	2389
R1						
1	86	1430	110	1678	62	1183
2	40	956	108	1657	62	1183
3	39	946	113	1709	88	1451
4	84	1410	116	1740	88	1451
5	84	1410	142	2008	141	1997
6	37	925	141	1997	141	1997
7	59	1152	115	1729	88	1451
8	61	1172	142	2008	141	1997
9	58	1142	112	1698	84	1410
stow	84	1513	55	1111	19	739
low						
high						
L3						
1	67	1234	62	1183	97	1544
2	134	1925	67	1234	95	1523
3	68	1245	60	1162	66	1224
4	132	1905	62	1183	67	1234
5	67	1234	37	925	20	750
6	130	1884	39	946	20	750
7	98	1554	63	1193	66	1224
8	97	1544	36	915	19	739
9						
stow	151	2100	120	1781	141	1997
low						
high						
L2						
1	62	1183	62	1183	99	1564
2	111	1688	63	1193	99	1564
3	63	1193	58	1142	67	1234
4	111	1688	54	1100	66	1224
5	64	1203	35	904	21	760
6	111	1688	34	894	22	770
7	87	1441	56	1121	67	1234
8	89	1461	34	894	22	770
9						
stow	83	1399	108	1657	120	1781
L1						
1	84	1410	113	1709	98	1554
2	129	1874	111	1688	98	1554
3	84	1410	106	1636	69	1255
4	130	1884	106	1636	69	1255
5	84	1410	79	1358	21	760
6	130	1884	80	1368	21	760
7	106	1636	106	1636	69	1255

	8	107	1647	79	1358	21	760
	9						
stow		51	1069	156	2152	140	1987

Table 5: Servo Physical Callibration Data

Height	Min	Max	Range
0	140	150	10
10	131.5	165	33.5
20	120	184	64
30	105	195	90
40	83	204	121
50	70	210	140
60	47	214	167
70	60	218	158
80	68	220	152
90	76	219	143
100	82	214	132
110	85	207	122
120	87.5	200	112.5
130	90	190	100
140	88	180	92
150	85	170	85
160	80	157	77
170	77	142	65
180	71	124	53
190	65	100	35
200	51.5	55	3.5

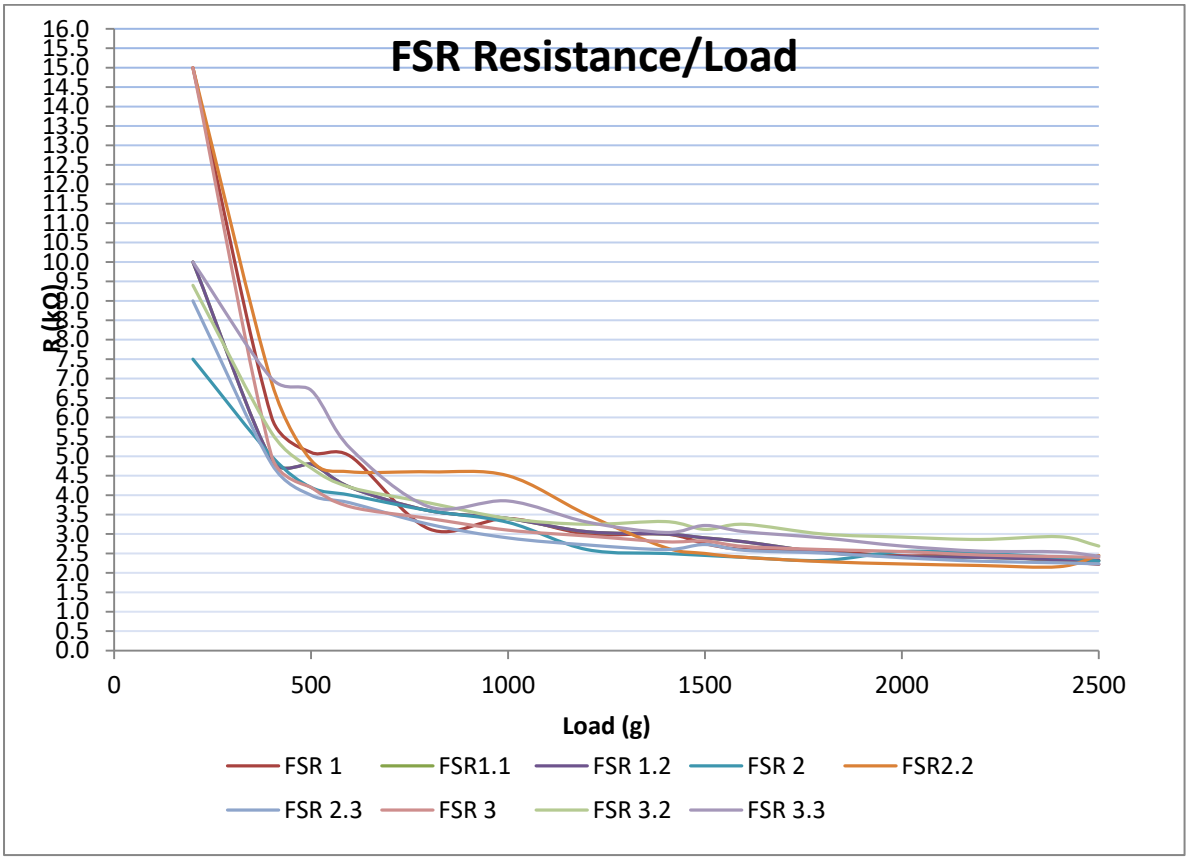
Table 6: Leg Workspace Calibration Data



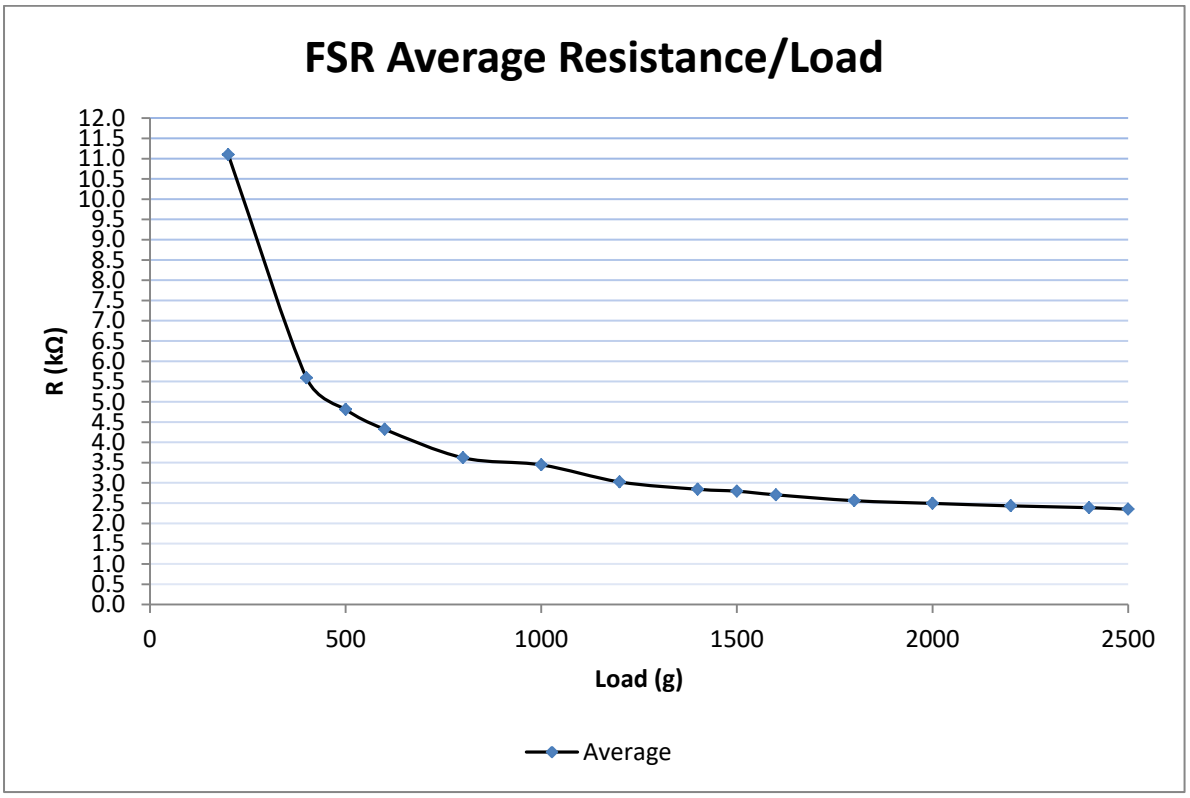
Graph 10: Workspace Calibration

Load (g)	FSR1 R(Ω)	FSR1 R(Ω)2	FSR1 R(Ω)3	FSR2 R(Ω)	FSR2 R(Ω)2	FSR2 R(Ω)3	FSR3 R(Ω)	FSR3 R(Ω)2	FSR3 R(Ω)3	Mid	Mean
200	15.00	10.00	7.50	15.00	9.00	9.00	15.00	9.40	10.00	10.00	11.10
400	6.00	4.90	5.00	6.90	5.10	4.80	5.00	5.60	7.00	5.10	5.59
500	5.10	4.80	4.20	4.90	4.70	4.00	4.20	4.70	6.70	4.70	4.81
600	5.00	4.20	4.00	4.60	4.20	3.80	3.70	4.20	5.20	4.20	4.32
800	3.13	3.60	3.60	4.60	3.50	3.25	3.40	3.80	3.70	3.60	3.62
1000	3.40	3.40	3.30	4.50	3.15	2.90	3.10	3.40	3.85	3.40	3.44
1200	3.00	3.06	2.60	3.50	2.83	2.72	2.95	3.25	3.31	3.00	3.02
1400	3.00	3.00	2.50	2.65	2.68	2.60	2.80	3.32	3.04	2.80	2.84
1500	2.75	2.90	2.45	2.50	2.67	2.73	2.82	3.12	3.22	2.75	2.80
1600	2.60	2.80	2.40	2.40	2.58	2.58	2.68	3.25	3.06	2.60	2.71
1800	2.55	2.53	2.32	2.29	2.35	2.51	2.60	3.00	2.90	2.53	2.56
2000	2.44	2.42	2.55	2.23	2.26	2.39	2.55	2.92	2.69	2.44	2.49
2200	2.45	2.40	2.50	2.19	2.18	2.30	2.46	2.86	2.56	2.45	2.43
2400	2.30	2.34	2.40	2.16	2.15	2.26	2.42	2.93	2.54	2.34	2.39
2500	2.22	2.32	2.30	2.45	2.12	2.23	2.40	2.69	2.43	2.32	2.35

Table 7: FSR Calibration Data



Graph 11: FSR Resistance/Load



Graph 12: FSR Resistance/Load Mean

Appendix VI: Leg Control Program

```
/*
arthroBOT individual leg controller code
Developed for Deakin Robotics Engineering Thesis Project
ArthroBOT
(Or, A Biomimetic Approach to a Distributed Leg Control
Architecture For Gait Generation in Polypedal Mobile Robots)
Supervisor A Prof Matthew Joordens
Dillon MacEwan 211285035
13/10/16
*/



#include <Servo.h> //import servo library
#include <Wire.h>
#include <PinChangeInt.h>

//#include <ros.h>     //ros libraries
//#include <std_msgs/String.h>

#define BUTTON 2

#define COR_IN 7  //Signal pins for Motor Cortex
#define COR_OUT 8

#define LAT_IN 9   //Signal Pins between legs
#define LAT_OUT 10
#define FOR_IN 11
#define FOR_OUT 12

#define BUMP 13

#define COXIA 0 //servos array
#define FEMUR 1
#define TIBIA 2

#define ALPHA 1 //angle array
#define BETA 2
#define GAMA 0

#define X 0 //aep/pep axis array
#define Y 1
#define Z 2

#define LEFT 1
#define RIGHT -1

#define FORWARD 1
#define REVERSE -1

#define CL 29.0 //joint lengths
#define FL 84.0
#define TL 152.0


unsigned long time;

//IK variables
float L, L1;
float alpha, alpha1, alpha2, beta, gama;
const float dg = 180 / PI; //Radians >> Degrees
const float rd = PI / 180; //Degrees >> Radians


float a2; // float for direction angle trig
float dx;
float dy;

const int numServos = 3; //how many servos do you have?
const int sPin[numServos] = {3, 5, 6}; //what pins do they correlate to?
Servo servo[numServos]; //declare the servo array
String joint[numServos] = {"Coxia", "Femur", "Tibia"}; //name of each servo

int bodyZ = 92; //ideal body height
int lastZ = bodyZ;

//arrays for servo data - enter data from table in description
int side;

int highPulse[numServos]; //pulse set for 180dg
```

```
int lowPulse[numServos]; //0 deg pulse

int np[3]; //xyz coordinates for neutral leg position

int realMaxA[numServos]; //angle on servo
int realMinA[numServos];


int maxA[numServos]; //angle for local coordinates
int minA[numServos];

#define ADDR 2


int leg = ADDR;



int aep[3]; //xyz coordinates for aep
int pep[3]; //xyz coordinates for pep

int tp[3]; //target position xyz

int ta[numServos]; //target angle

int travelAngle = 0; // variables for omnidirectional motion
int strokeL = 100; //length of leg stroke
int setStride = 100; //stores an ideal forward motion strude length

int steps = 30; // variable for how many IK targets are used in a given path
int lift = 30; //height foot steps in mm
int halfS = steps / 2;
int mode = 1;

int pathR = 3; //increment between IK points in leg path in mm
/*duty cycle as a value between 0 and 40 - 0 is trpod, 40 is tetra gait
*/
int duty = 40;
int IKdelay = 10;
int acker = 0;
int stOut = 0;

int dir = FORWARD; //FORWARD = 1 REVERSE = -1

bool walking = false; //states for walk()
bool striding = false;
bool ackman = false;
bool run = false;


int pb = 0; //used to hold push button reading
int cycles = 0;




void setup()
{

  getParams();
  Serial.begin(19200); // initialize serial output
  Serial.println("Press button to center joints...");

  //Wire.begin(ADDR); //initiate Wire, with leg address
  //Wire.onReceive(receiveEvent); // register event
  //Wire.onRequest(requestEvent);

  pinMode (BUTTON, INPUT);
  pinMode (LAT_IN, INPUT);
  pinMode (FOR_IN, INPUT);
  pinMode (COR_IN, INPUT);
  pinMode (BUMP, INPUT);
  pinMode (LAT_OUT, OUTPUT);
  pinMode (FOR_OUT, OUTPUT);
  pinMode (COR_OUT, OUTPUT);

  if (leg != 6)
  {
    PCintPort::attachInterrupt(FOR_IN, &reflex, FALLING);
    PCintPort::attachInterrupt(LAT_IN, &reflex, FALLING);
  }
  PCintPort::attachInterrupt(BUTTON, &halt, FALLING);
```

```
  Serial.println();
  while(!pb)
  {
    pb = digitalRead(BUTTON);
  }
  pb = 0;
  delay(100); //debounce


  for (int i = 0; i < numServos; i++)
  {
    servo[i].attach(sPin[i]);
  }


  //this routine attaches the servos with set limits, and maps position to degrees

  for(int i = 0; i < numServos; i++)
  {



    servo[i].attach(sPin[i], lowPulse[i], highPulse[i]); //attach this servo


    Serial.print(i);
    Serial.print(joint[i]);
    Serial.print(" attached");
    Serial.println();




  }

  //set foot to neutral
  getAEP PEP(strokeL, travelAngle);
//  Serial.println("go to np");
  posFoot(np[X], np[Y], np[Z]);

  delay(500);





}

void loop()
{

//  Serial.println("Press button for single leg walk routine...");




  Serial.println();
  while(!pb)
  {
    pb = digitalRead(BUTTON);
  }

  pb = 0;
  delay(200); //debounce
//  Serial.println("Walk forward, 18 steps");
//  Serial.println();

  travelAngle = 0;
  getAEP PEP(strokeL, travelAngle);
  cycles = -2;

  walk();

//    Serial.println("Press button for single leg walk routine...");
```

```
//  Serial.println();
  while(!pb)
  {
    pb = digitalRead(BUTTON);
  }

  pb = 0;
  delay(200);

//  Serial.println("turn left 20");
  mode = 2;
  walk();
//    Serial.println("Press button for single leg walk routine...");





//  Serial.println();
  while(!pb)
  {
    pb = digitalRead(BUTTON);
  }

  pb = 0;
  delay(200);

//  Serial.println("turn right 10 steps");
  mode = 3;
  walk();
  mode = 1;
  travelAngle = 0;

  while(!pb)
  {
    pb = digitalRead(BUTTON);
  }

  pb = 0;
  delay(200);

  turn(LEFT, 8);


  while(!pb)
  {
    pb = digitalRead(BUTTON);
  }

  pb = 0;
  delay(200);

  turn(RIGHT, 8);





}

//FUNCTIONS

void walk()
{
  duty = 40;
  int dutyInc = 2;
  int angleInc = 3;
  getAEP_PEP(strokeL, travelAngle);
  dx = (aep[X] - pep[X]) / float(steps);
  dy = (aep[Y] - pep[Y]) / float(steps);
  run = false;
  walking = true;
  swingTo(np[X], np[Y]);

  if (mode == 4)
  {
    duty = 20;
  }


  switch(leg)
  {
```

```
    case 1: //L1
    {
      swingTo(aep[X], aep[Y]);
      delay(50);
      stride();
    }
    break;
    case 2: //L2
    {
      swingTo(aep[X], aep[Y]);
      delay(100);
      stride();
    }
    break;
    case 3: //L3
    {
      delay(100);
      stride();
    }
    break;
    case 4: //R1
    {
      delay(100);
      stride();
    }
    break;
    case 5: //R2
    {
      delay(100);
      stride();
    }
    break;
    case 6: //R3
    {

//      swingTo(aep[X], aep[Y]);
//      delay(100);
//      swingTo(np[X], np[Y]);

      swingTo(pep[X], pep[Y]);
      delay(200);


      swingTo(aep[X], aep[Y]);
      delay(100);
      run = true;
      stride();

    }
    break;
  }

  while (walking)
  {

    if ((mode < 3) && (acker != 0))
    {
      ackman = true;
      ackerman();
    }
    if ((ackman == true) && (acker == 0))
    {
      ackman = false;
      travelAngle = 0;
      strokeL = setStride;
    }

    getAEP_PEP(strokeL, travelAngle);
    run = true;
    swingTo(aep[X], aep[Y]);


    if (walking) //in case a halt is called in swing
    {
      stride();
    }
    if (mode == 2)
    {
      duty -= dutyInc;
      if (duty <= 3)
      {
        dutyInc = -2;
```

```
      }
      else if (duty >= 37)
      {
        dutyInc = 2;
      }
    }
    if (mode == 3)
    {
      duty = 20;
      travelAngle -= angleInc;
      if (travelAngle <= -45)
      {
        angleInc = -2;
      }
      else if (travelAngle >= 45)
      {
        angleInc = 2;
      }
    }
    cycles--;
    if (cycles < -2)
    {
      cycles = -2;
    }

    else if (cycles == 0)
    {
      walking = false;
    }
  }
  run = false;
  delay(leg * 100);
  swingTo(np[X], np[Y]);
  delay(100);

}


void turn(int dr, int cyc) //direction, duty(speed), steps
{
//  if (du > 40)
//  {
//    du = 40;
//  }
//  duty = du;
  mode = 4;
  cycles = cyc;

  switch(leg)
  {
    case 1: //L1
    {

      travelAngle = -48;
      dir = dr * (-1);
    }
    break;
    case 2: //L2
    {
      travelAngle = 0;
      dir = dr * (-1);
    }
    break;
    case 3: //L3
    {

      travelAngle = 48;
      dir = dr * (-1);
    }
    break;
    case 4: //R1
    {

      travelAngle = 48;
      dir = dr;
    }
    break;
    case 5: //R2
    {
      travelAngle = 0;
      dir = dr;
    }
```

```
    break;
    case 6: //R3
    {

      travelAngle = -48;
      dir = dr;
      Serial.println(travelAngle);

    }
    break;
  }
  walk();
}

//posFoot moved the foot position to the desired x,y,z coordinates
// by retreiving the servo angles using IKtrig() then maping to the
//true servo angle values

void posFoot(int x, int y, int z)
{
  tp[X] = x;
  tp[Y] = y;
  tp[Z] = z;
  IKtrig(x, y, z); //retrieve servo positions
  for(int i = 0; i < numServos; i++)
  {
    mapWriteServo(i); //map and write true servo positions
  }

}



//swing along an eliptical path
void swing()
{

  float dx = (aep[X] - pep[X]) / float(steps);
  float dy = (aep[Y] - pep[Y]) / float(steps);
  float EllipseLift = 0;
  float cx = pep[X];
  float cy = pep[Y];
  float a;
//  Serial.println(steps);
//  Serial.println(halfS);


  for(int j = 0; j < 3; j++)
  {
    tp[j] = pep[j];


  }



  cx = tp[X];
  cy = tp[Y];

  for(int i = 0; i < steps; i++)
  {
    posFoot(tp[X], tp[Y], tp[Z]);

    a = (halfS - float(i + 1)) / halfS;
    a = a * a;
//    Serial.println(i);
//    Serial.println(a);
    if (a > 1)
    {
      a = 1;
    }

    EllipseLift =  sqrt(1 - a) * lift;

    cx += dx;
    cy += dy;
    tp[X] = cx ;
    tp[Y] = cy ;


    tp[Z] = np[Z] - EllipseLift ;
```

```
//    Serial.println(tp[Z]);
//    Serial.println(EllipseLift);



    delay(IKdelay);

  }
  posFoot(tp[X], tp[Y], tp[Z]);
  delay(IKdelay);
}

// swingToAEP lifts and places foot at AEP from any position

void swingToAEP()
{
  long delX = aep[X] - tp[X];
  long delY = aep[Y] - tp[Y];
  float strk = hypot(delX, delY);
  int stp = strk / pathR;
  float dx = delX / float(stp);
  float dy = delY / float(stp);
  int hlfS = stp / 2;
  float EllipseLift = 0;
  float cx = tp[X];
  float cy = tp[Y];
  float a;

  for(int i = 0; i < stp; i++)
  {
    posFoot(tp[X], tp[Y], tp[Z]);
    a = (hlfS - float(i + 1)) / hlfS;
    a = a * a;
    if (a > 1)
    {
      a = 1;
    }

    EllipseLift =  sqrt(1 - a) * lift;

    cx += dx ;
    cy += dy ;
    tp[X] = cx;
    tp[Y] = cy;
    tp[Z] = np[Z] - EllipseLift;
    delay(IKdelay);

  }


  posFoot(aep[X], aep[Y], aep[Z]);
  delay(IKdelay);

}

//This functions lifts and places the foot to a specified x y location at neutral foot height

void swingTo(int x, int y)
{
  long delX = x - tp[X];
  long delY = y - tp[Y];
  if (delX == 0)
  {
    delX = 1;
  }
  if (delY == 0)
  {
    delY = 1;
  }
  float strk = hypot(delX, delY);
  int stp = strk / pathR;
  dx = delX / float(stp);
  dy = delY / float(stp);
  int hlfS = stp / 2;
  float EllipseLift;
  float cx = tp[X];
  float cy = tp[Y];
  float a;
  int bump = 0;
  lift = 30;

  digitalWrite(FOR_OUT, HIGH);
```

```
  if (leg == 6)
  {
    digitalWrite(LAT OUT, HIGH);
  }

  for(int i = 0; i < stp; i++)
  {
    posFoot(tp[X], tp[Y], tp[Z]);

    a = (hlfS - float(i + 1)) / hlfS;
    a = a * a;
    if (a > 1)
    {
      a = 1;
    }

    EllipseLift =  sqrt(1 - a) * lift;

    cx += dx ;
    cy += dy ;
    tp[X] = cx;
    tp[Y] = cy;
    tp[Z] = np[Z] - EllipseLift;
    delay(IKdelay);
    bump = digitalRead(BUMP);
    if (bump)
    {
      Serial.println("Bump!");
      stepUp();
    }

  }


  posFoot(x, y, np[Z]);
  delay(IKdelay);

}

//stride algorithm:  from aep to pep, straight line trajectory through np

void stride()
{

  dx = (aep[X] - pep[X]) / float(steps);
  dy = (aep[Y] - pep[Y]) / float(steps);

  float cx = aep[X]; //reduce rounding errors
  float cy = aep[Y];

  int del = IKdelay + duty; //duty cycle delay for striding

  int st = 0; //step count
  int ls = steps * duty / 100; // tmie for lateral signal from R3

  int lat = 0; //inputs for lateral and foraqwaed feedforward signal
  int fwd = 0;

  tp[X] = aep[X];
  tp[Y] = aep[Y];
  tp[Z] = lastZ;
  striding = true;


  while (striding)
  {
    st++; //count steps used

    posFoot(tp[X], tp[Y], tp[Z]);

    cx -= dx;
    cy -= dy;
    tp[X] = cx ;
    tp[Y] = cy ;
    delay(del);

    if (leg == 3)
    {
      fwd = digitalRead(LAT_IN);
    }
    else
    {
```

```
      fwd = digitalRead(FOR_IN);
    }



//    if (fwd == 1)
//    {
//      Serial.println("Forward Signal In!");
//    }


    if (run == true)
    {
      if (st >= 2)
      {

        digitalWrite(FOR_OUT, LOW);
      }


    }

    if (leg == 6) //driving leg R3
    {
      if (st >= ls)
      {
        Serial.println("Lateral Signal!");
        digitalWrite(LAT_OUT, LOW);
      }

      if (st >= steps)
      {
        striding = false;
      }
    }
    else if(st >= 26) //replace with ramping function
    {
      if ((st == 26) && (del < 29))
      {
        del = 29;
      }
      del++;
    }
    else if(st > 38)
    {
      dx = 0;
      dy = 0;

    }

    if ((st > steps) && (fwd == 0))
    {
      striding = false;
    }

//    if (lat || fwd)
//    {
//      striding = false;
//    }
  }
}

void ackerman()
{
  int radius; //radius of travel of outer centre foot
  int r;    //radius of travel of this foot
  int d;    //perpendicular offset of this foot
  int dr = 1; //+ or - direction of foot travel
  float b;  //holder for offset for maths
  float a;  //holder for trig angle
  int L = setStride; //outside centre leg stroke length
  radius = 480 + (abs(acker) * 20); // find desired turning raduis

  switch(leg)
  {
    case 1: //L1
    {
      if (acker > 0)
      {
        d = 328;
        dr = 1;
      }
```

```
      else
      {
        d = 85;
        dr = -1;
      }
    }
    break;
    case 2: //L2
    {
      if (acker > 0)
      {
        strokeL = L * (1 - (408 / radius));
        travelAngle = 0;
      }
      else
      {
        strokeL = setStride;
        travelAngle = 0;
      }
    }
    break;
    case 3: //L3
    {
      if (acker > 0)
      {
        d = 353;
        dr = -1;
      }
      else
      {
        d = 55;
        dr = 1;
      }
    }
    break;
    case 4: //R1
    {
      if (acker > 0)
      {
        d = 85;
        dr = 1;
      }
      else
      {
        d = 328;
        dr = -1;
      }
    }
    break;
    case 5: //R2
    {
      if (acker > 0)
      {
        strokeL = setStride;
        travelAngle = 0;
      }
      else
      {
        strokeL = L * (1 - (408 / radius));
        travelAngle = 0;
      }
    }
    break;
    case 6: //R3
    {
      if (acker > 0)
      {
        d = 55;
        dr = -1;
      }
      else
      {
        d = 353;
        dr = 1;
      }
    }
    break;

    if ((leg != 2) && (leg != 5)) // if not centre legs
    {
      b = radius - d;
      r = hypot(178, b);
```

```
      strokeL = L * r / radius;
      a = atan2 (178.0, b);
      travelAngle = a * dg * dr;
    }
  }
}

void stepUp()
{
  int exs;
  int wiy;
  int zed;

  zed = tp[Z] - 15;
  exs = tp[X];
  wiy = tp[Y];
  exs -= 6 * dx;
  wiy -= 6 * dy;

  posFoot(exs, wiy, zed);
  delay(IKdelay);
  delay(IKdelay);

  zed = zed - 5;
  exs += 7 * dx;
  wiy += 7 * dy;

  posFoot(exs, wiy, zed);

  delay(IKdelay);
  delay(IKdelay);

  tp[X] += 4 * dx;
  tp[Y] += 4 * dy;
  tp[Z] = zed;
  lift += 20;
}

void stepOut()
{
  int exs;
  int wiy;
  int zed;

  zed = tp[Z] - 15;
  exs = tp[X];
  wiy = tp[Y];
  exs -= 6 * dx;
  wiy -= 6 * dy;

  posFoot(exs, wiy, zed);
  delay(IKdelay);
  delay(IKdelay);

  zed = zed - 5;
  exs += 7 * dx;
  wiy += 7 * dy;

  posFoot(exs, wiy, zed);

  delay(IKdelay);
  delay(IKdelay);

  tp[X] += 4 * dx;
  tp[Y] += 4 * dy;
  tp[Z] = zed;
  lift += 20;
}


// function converts IK angles to real angles and writes them to servo

int mapWriteServo (int s)//s = servo
{
  int x = map(ta[s], minA[s], maxA[s], realMinA[s], realMaxA[s]);
  servo[s].write(x);
}


//IK maths

void IKtrig(int x, int y, int z)
```

```
{


  float x1 = x;
  float y1 = y;
  float z1 = z;
  L1 = hypot(x1, y1);

  gama = atan2(x1, y1) * dg;

  L = hypot((L1 - CL), z1);

  beta = acos((sq(TL) + sq(FL) - sq(L)) / ( 2 * TL * FL)) * dg;

  alpha1 = acos(z1 / L) * dg;
  alpha2 = acos((sq(FL) + sq(L) - sq(TL)) / (2 * FL * L)) * dg;
  alpha = alpha1 + alpha2;

  ta[ALPHA] = alpha;
  ta[BETA] = beta;
  ta[GAMA] = gama;

  for (int i = 0; i < numServos; i++)
  {
    if (ta[i] < minA[i])
    {
      ta[i] = minA[i];
    }
    if (ta[i] > maxA[i])
    {
      ta[i] = maxA[i];
    }
  }



}


//function retrieves AEP and PEP x and y values fom direction angle

void getAEP PEP(int l, int a) //l - length of stroke - a - angle of direction in degrees
{
  a2 = a * rd; // convert a to Radians for trig
  aep[X] = np[X] + dir * (0.5 * l * cos(a2));
  aep[Y] = np[Y] + dir * (side * 0.5 * l * sin(a2));
  aep[Z] = np[Z];
  pep[X] = np[X] - dir * (0.5 * l * cos(a2));
  pep[Y] = np[Y] - dir * (side * 0.5 * l * sin(a2));
  pep[Z] = np[Z];
}

//Interrupt Pin Functions

void reflex()
{
  Serial.println("Reflex!");
  striding = false;
}

void halt()
{
  if (walking == true)
  {
    setAll();
  }
}

void setFlags() // a quick function just to reset all walk flags
{
  bool walking = false; //states for walk()
  bool striding = false;
  bool ackman = false;
  bool run = false;
}

void setAll()
{
  setFlags();

  mode = 1;
  strokeL = setStride;
```

```
  travelAngle = 0;
  duty = 40;
  stOut = 0;
}

//Communication Functions

void receiveEvent(int HowMany)
{


}

void requestEvent()
{


}

//store parameters for each leg controller chip


void getParams()
{

  switch (leg)

  {
    case 1://L1 Front Left
    {
      side = LEFT;
      int Params [7][3] =
      {
        {2303, 2460, 2296},
        {611, 480, 568},
        {115, 80, bodyZ},
        {135, 170, 19},
        {85, 50, 139},
        {77, 180, 135},
        {27, 60, 15}
      };

      for (int i = 0; i < numServos; i++)
      {
        highPulse[i] = Params[0][i]; //pulse set for 180dg
        lowPulse[i] = Params[1][i]; //0 deg pulse

        np[i] = Params[2][i]; //xyz coordinates for neutral leg position

        realMaxA[i] = Params[3][i]; //angle on servo
        realMinA[i] = Params[4][i];


        maxA[i] = Params[5][i]; //angle for local coordinates
        minA[i] = Params[6][i];


      }

    }
    break;



    case 2://L2
    {
      side = LEFT;
      int Params [7][3] =
      {
        {2150, 2534, 2387},
        {650, 410, 605},
        {12, 140, bodyZ},
        {125, 160, 45},
        {65, 40, 165},
        {35, 180, 135},
        { -25, 60, 15}
      };

      for (int i = 0; i < numServos; i++)
      {
        highPulse[i] = Params[0][i]; //pulse set for 180dg
        lowPulse[i] = Params[1][i]; //0 deg pulse
```

```
      np[i] = Params[2][i]; //xyz coordinates for neutral leg position

      realMaxA[i] = Params[3][i]; //angle on servo
      realMinA[i] = Params[4][i];


      maxA[i] = Params[5][i]; //angle for local coordinates
      minA[i] = Params[6][i];


    }
  }
  break;



  case 3://L3
  {
    side = LEFT;

    int Params [7][3] =
    {
      {2600, 2331, 2290},
      {260, 571, 580},
      { -95, 110, bodyZ},
      {125, 125, 19},
      {75, 5, 139},
      { -12, 180, 135},
      { -62, 60, 15}
    };

    for (int i = 0; i < numServos; i++)
    {
      highPulse[i] = Params[0][i]; //pulse set for 180dg
      lowPulse[i] = Params[1][i]; //0 deg pulse

      np[i] = Params[2][i]; //xyz coordinates for neutral leg position

      realMaxA[i] = Params[3][i]; //angle on servo
      realMinA[i] = Params[4][i];


      maxA[i] = Params[5][i]; //angle for local coordinates
      minA[i] = Params[6][i];


    }

  }
  break;



  case 4://R1
  {
    side = RIGHT;

    int Params [7][3] =
    {
      {2313, 2423, 2361},
      {621, 515, 597},
      {115, 80, bodyZ},
      {35, 50, 144},
      {85, 170, 24},
      {77, 180, 135},
      {27, 60, 15}
    };

    for (int i = 0; i < numServos; i++)
    {
      highPulse[i] = Params[0][i]; //pulse set for 180dg
      lowPulse[i] = Params[1][i]; //0 deg pulse

      np[i] = Params[2][i]; //xyz coordinates for neutral leg position

      realMaxA[i] = Params[3][i]; //angle on servo
      realMinA[i] = Params[4][i];


      maxA[i] = Params[5][i]; //angle for local coordinates
      minA[i] = Params[6][i];
```

```
    }

  }
  break;



  case 5://R2
  {
    side = RIGHT;

    int Params [7][3] =
    {
      {2198, 2312, 2368},
      {488, 515, 586},
      {12, 140, bodyZ},
      {55, 39, 145},
      {115, 159, 25},
      {35, 180, 135},
      { -25, 60, 15}
    };

    for (int i = 0; i < numServos; i++)
    {
      highPulse[i] = Params[0][i]; //pulse set for 180dg
      lowPulse[i] = Params[1][i]; //0 deg pulse

      np[i] = Params[2][i]; //xyz coordinates for neutral leg position

      realMaxA[i] = Params[3][i]; //angle on servo
      realMinA[i] = Params[4][i];


      maxA[i] = Params[5][i]; //angle for local coordinates
      minA[i] = Params[6][i];


    }

  }
  break;


  case 6://R3
  {
    side = RIGHT;

    int Params [7][3] =
    {
      {2077, 2430, 2414},
      {547, 486, 578},
      { -94, 110, bodyZ},
      {15, 46, 143},
      {65, 166, 23},
      { -12, 180, 135},
      { -62, 60, 15}
    };

    for (int i = 0; i < numServos; i++)
    {
      highPulse[i] = Params[0][i]; //pulse set for 180dg
      lowPulse[i] = Params[1][i]; //0 deg pulse

      np[i] = Params[2][i]; //xyz coordinates for neutral leg position

      realMaxA[i] = Params[3][i]; //angle on servo
      realMinA[i] = Params[4][i];


      maxA[i] = Params[5][i]; //angle for local coordinates
      minA[i] = Params[6][i];


    }


  }
  break;
```

```
  }

}

/*

this is the values for each leg assembly
//swap both coxia and tibia real max/min values



//this following sequence is the original test for foot positions

  getAEP PEP(strokeL, travelAngle);

  Serial.println();
  Serial.print("NP Coordinates: ");
  for(int i = 0; i < 3 ; i++)
  {
    Serial.print(np[i]);
    Serial.print(", ");
  }

  IKtrig(np[X], np[Y], np[Z]);
  Serial.println();
  Serial.print("NP Joint Angles: ");
  for(int i = 0; i < 3 ; i++)
  {
    Serial.print(joint[i]);
    Serial.print(": ");
    Serial.print(ta[i]);
    Serial.print(", ");
    mapWriteServo(i);
  }
  delay(500);

  Serial.println();
  Serial.print("AEP Coordinates: ");
  for(int i = 0; i < 3 ; i++)
  {
    Serial.print(aep[i]);
    Serial.print(", ");
  }

  IKtrig(aep[X], aep[Y], aep[Z]);
  Serial.println();
  Serial.print("AEP Joint Angles: ");
  for(int i = 0; i < 3 ; i++)
  {
    Serial.print(joint[i]);
    Serial.print(": ");
    Serial.print(ta[i]);
    Serial.print(", ");
    mapWriteServo(i);
  }
  delay(500);




  Serial.println();
  Serial.print("PEP Coordinates: ");
  for(int i = 0; i < 3 ; i++)
  {
    Serial.print(pep[i]);
    Serial.print(", ");
  }
  IKtrig(pep[X], pep[Y], pep[Z]);
  Serial.println();
  Serial.print("PEP Joint Angles: ");
  for(int i = 0; i < 3 ; i++)
  {
    Serial.print(joint[i]);
    Serial.print(": ");
    Serial.print(ta[i]);
    Serial.print(", ");
    mapWriteServo(i);
  }
  delay(2500);

//end of foot position test
```

```
*/


/*
Swing moves the leg from AEP to PEP arcing through a sinosoidal path
to a height determined by the variable lift

*/

//void swing()
//{
//  //int dx = (cos(travelAngle * rd) * strokeL) / steps;
//  //int dy = (sin(travelAngle * rd) * strokeL) / steps;
//  float dx = (aep[X] - pep[X])/float(steps);
//  float dy = (aep[Y] - pep[Y])/float(steps);
//  float sinLift;
//  float cx;
//  float cy;
//
//  for(int j = 0; j < 3; j++)
//  {
//    tp[j] = pep[j];
//
//
//  }
//  posFoot(tp[X], tp[Y], tp[Z]);
//  //delay(10);
//
//  cx = tp[X];
//  cy = tp[Y];
//
//  for(int i = 0; i < steps; i++)
//  {
//    posFoot(tp[X], tp[Y], tp[Z]);
//
//    sinLift =  sin(PI * (float(i + 1) / steps)) * lift;
//
//    cx += dx;
//    cy += dy;
//    tp[X] = cx ;
//    tp[Y] = cy ;
//
//
//    tp[Z] = np[Z] - sinLift ;
//
//    //delay(3);
//
//  }
//
//
//}
```

Appendix VII: Calibration Programs

Physical Calibration code

```
/*
physical callibration code
default servo library pulse range is 544 to 2400 for 0 - 180 deg
*/


#include <Servo.h>

#define BUTTON 2
#define COXIA 0
#define FEMUR 1
#define TIBIA 2


//create servo objects for each lamp joint

const int numServos = 3; //how many servos do you have?
const int sPin[numServos] = {3, 5, 6}; //what pins do they correlate to?, 5, 6
Servo legServo[numServos]; //servo array
String joint[numServos] = {"Coxia", "Femur", "Tibia"}; //name of each servo

int pb = 0;

// define Pots ADC Pins

int coxiaPot = A0;
int femurPot = A1;
int tibiaPot = A2;


int controlPot[numServos]; //control pot array
int potVal[numServos];    // variable to read the value from the analog pin
int sPos[numServos];
int pulse[numServos];

void setup()
{

  Serial.begin(19200); // initialize serial output
  Serial.println("it's on!");
  Serial.println("Press button to output current angle and pulse");
  pinMode (BUTTON, INPUT);
  for (int i = 0; i < 3; i++)
  {
    legServo[i].attach(sPin[i]);
  }


}

void loop()
{

  while(!pb)
  {

    for(int i = 0; i < numServos ; i++)
    {
      potVal[i] = analogRead(i);            // reads the value of the potentiometer (value between 0 and 1023)
      sPos[i] = map(potVal[i], 0, 1023, 0, 180);    // scale it to use it with the servo (value between 0 and 180)
      legServo[i].write(sPos[i]);                  // sets the servo position according to the scaled value
      delay(5);        // waits for the servo to get there
    }
    pb = digitalRead(BUTTON);
  }
  for (int x = 0; x < numServos; x++)

  {
    pulse[x] = map(sPos[x], 0, 180, 544, 2400);
    Serial.println(joint[x]);
    Serial.print("Pulse; ");
    Serial.print(pulse[x]);
    Serial.print("   Position: ");
    Serial.println(sPos[x]);


  }
```

```
  Serial.println();
  delay(100);
  pb = 0;


}
```

Electronic Calibration Code

```
Program outputs a range of position readings for a range of positions on
3 servos for a hexapod leg to a terminal
CoolTerm was used to save that data to a file for analysis
For 50 pulse positions, it outputs 50 raw and 50 smothed readings
The purpose is to analyse the noise and accuracy of the pos feedback
This code can be addapted for other servo configurations using pos feedback
by changing numServo, and the joint name array
Dillon MacEwan 13/8/16
*/

#include <Servo.h> //import servo library

#define BUTTON 2
#define COXIA 0
#define FEMUR 1
#define TIBIA 2

const int numServos = 1; //how many servos do you have?
const int sPin[numServos] = {3}; //what pins do they correlate to?, 5, 6
Servo servo[numServos]; //declare the servo array
String joint[numServos] = {"Coxia"}; //name of each servo , "Femur", "Tibia"
//arrays for servo data
int highPulse[numServos];
int lowPulse[numServos];
int cenPulse[numServos];
int highPos[numServos];
int lowPos[numServos];
int cenPos[numServos];

int here[numServos]; //angle moving from
int there[numServos]; //angle moving to

float x[numServos]; //angle converted to radians to derive cosine wave
int angle; //angle derived from cosine function. sent to servo in loop
boolean doneMove[numServos];


int btwReadings = 20; //delay time between
int whlReading = 3; //delay time between analog readings of internal pot
int polT = whlReading * 20;

float vRef = 2.37;


int pb = 0; //used to hold push button reading
int pos;
int t;
int pulse = 1500; //first uS pulse used in range test

int feedBack;

void setup()
{

        Serial.begin(19200); // initialize serial output
        Serial.println("it's on!");
        analogReference(EXTERNAL);
        pinMode (BUTTON, INPUT);
        for (int i = 0; i < numServos; i++)
        {
                servo[i].attach(sPin[i]);
        }

        calibrate();


        //this routine attaches the servos with set limits, and maps position to degrees

        for(int i = 0; i < numServos; i++)
        {

                servo[i].writeMicroseconds(cenPulse[i]);
```

```
                feedBack = cenPos[i]; //take current reading from pot
                there[i] = map(feedBack, lowPos[i], highPos[i], 0, 180); //adjust feedback to degree output
                servo[i].attach(sPin[i], lowPulse[i], highPulse[i]); //attach this servo
                servo[i].write(there[i]); //send out pulse for where we are

                Serial.print(i);
                Serial.print(joint[i]);
                Serial.print(" attached");
                Serial.println();
                doneMove[i] = true;



        }


}

void loop()
{
        Serial.println("testing");
        moveTest();
        checkAngleRange();

}

void calibrate()
{
        Serial.print("vRef: ");
        Serial.print(vRef);
        Serial.println("V");
        Serial.print("polling time: ");
        Serial.print(polT);
        Serial.println("ms");
        for (int x = 0; x < numServos; x++)
        {
                Serial.print("Press button to calibrate Servo: ");
                servo[x].writeMicroseconds(pulse); //send servo to start of range
                Serial.print(joint[x]);
                Serial.println();
                while(!pb)
                {
                        pb = digitalRead(BUTTON);
                }
                pb = 0;
                Serial.print("Calibrating in ..3");
                for (int i = 2; i >= 0; i--) //count down three seconds
                {
                        delay(1000);
                        Serial.print("..");
                        Serial.print(i);
                }

                // up range
                pulse = 1500;
                delay(20);
                servo[x].writeMicroseconds(pulse); //send servo to start of range
                delay(2000); //wait for it to get there
                Serial.println();
                pos = getFeedback(x);
                pos -= 20;
                t = pos;

                Serial.print(joint[x]);
                pulse = 1500;
                delay(20);
                servo[x].writeMicroseconds(pulse); //send servo to start of range
                delay(2000); //wait for it to get there


                pulse += 10;   //overcome the servos initial inertia before calibration
                servo[x].writeMicroseconds(pulse);



                Serial.println();
                Serial.print("High Pulse Range");
                Serial.println();
                do
                {
```

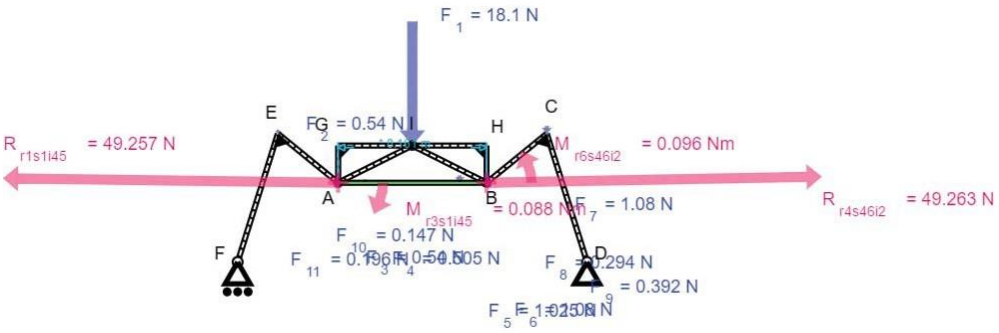
```
        t = pos;

        Serial.print(t);
        Serial.print(",");
        pulse += 15;
        servo[x].writeMicroseconds(pulse);
        delay(250);
        pos = getFeedback(x);
        Serial.print(pos);
        Serial.print(",");
        Serial.print(pulse);
        Serial.println();



}
while(pos > t);

highPulse[x] = pulse - 20;
highPos[x] = pos - 10;

//down range

pulse = 1500;
delay(20);
servo[x].writeMicroseconds(pulse); //send servo to start of range
delay(2000); //wait for it to get there
Serial.println();
pos = getFeedback(x);
pos += 20;
t = pos;

Serial.print(joint[x]);
pulse = 1500;
delay(20);
servo[x].writeMicroseconds(pulse); //send servo to start of range
delay(2000); //wait for it to get there

pulse -= 10;   //overcome the servos initial inertia before calibration
servo[x].writeMicroseconds(pulse);


Serial.println();
Serial.print("Low Pulse Range");
Serial.println();
do
{

        t = pos;

        Serial.print(t);
        Serial.print(",");
        pulse -= 15;
        servo[x].writeMicroseconds(pulse);
        delay(200);
        pos = getFeedback(x);
        Serial.print(pos);
        Serial.print(",");
        Serial.print(pulse);
        Serial.println();



}
while(pos < t);

lowPulse[x] = pulse + 20;
lowPos[x] = pos + 10;
cenPulse[x] = ((highPulse[x] - lowPulse[x]) / 2) + lowPulse[x];

cenPos[x] = ((highPos[x] - lowPos[x]) / 2) + lowPos[x];
Serial.print(lowPulse[x]);
Serial.print(",");
Serial.print(cenPulse[x]);
Serial.print(",");
Serial.print(highPulse[x]);
Serial.println();
Serial.print(lowPos[x]);
Serial.print(",");
Serial.print(cenPos[x]);
Serial.print(",");
Serial.print(highPos[x]);
```

```
                Serial.println();

        }
        //output results
        Serial.print("vRef: ");
        Serial.print(vRef);
        Serial.println("V");
        Serial.print("polling time: ");
        Serial.print(polT);
        Serial.println("ms");
        Serial.println();
        for (int x = 0; x < numServos; x++)
        {
                Serial.println(joint[x]);
                Serial.print("Pulse Range(us); ");
                Serial.print(lowPulse[x]);
                Serial.print(" <> ");
                Serial.println(highPulse[x]);
                Serial.print("Position Range(bits): ");
                Serial.print(lowPos[x]);
                Serial.print(" <> ");
                Serial.println(highPos[x]);
                Serial.println();

        }
}




void checkAngleRange()
{
        for (int x = 0; x < numServos; x++)
        {
                Serial.print("Press button to check angle range of Servo: ");
                servo[x].writeMicroseconds(pulse); //send servo to start of range
                Serial.print(joint[x]);
                Serial.println();
                while(!pb)
                {
                        pb = digitalRead(BUTTON);
                }
                pb = 0;
                Serial.println("Low position");
                delay(1000);
                servo[x].writeMicroseconds(lowPulse[x]);
                delay(3000);
                Serial.println();
                Serial.println("Press button to set high position: ");
                while(!pb)
                {
                        pb = digitalRead(BUTTON);
                }
                pb = 0;
                Serial.println("High position");
                delay(1000);
                servo[x].writeMicroseconds(highPulse[x]);
                delay(3000);
                Serial.println();
                Serial.println("Press button to set centre position: ");
                while(!pb)
                {
                        pb = digitalRead(BUTTON);
                }
                pb = 0;
                Serial.println("Centre position");
                delay(1000);
                servo[x].writeMicroseconds(cenPulse[x]);
                delay(3000);

        }

}

void moveTest()
{
        Serial.print("Press button to start move test ");

        Serial.println();
        while(!pb)
        {
                pb = digitalRead(BUTTON);
```

```
        }
        pb = 0;
        delay(100); //debounce
        Serial.println("Press button to stop move test ");
        Serial.println();
        while(!pb)
        {

                for (int i = 0; i < numServos; i++)
                {
                        if(doneMove[i] == true)
                        {
                                doneMove[i] = false;
                                here[i] = there[i];
                                there[i] = random(180.1) + 0.5;
                                if(there[i] == here[i])
                                {
                                        there[i] = random(180.1) + 0.5;
                                }
                                if(here[i] < there[i])
                                {
                                        x[i] = 0;
                                }
                                else
                                {
                                        x[i] = 180;
                                }
                                Serial.print("Move servo ");
                                Serial.print(i);
                                Serial.print(" from ");
                                Serial.print(here[i]);
                                Serial.print(" to ");
                                Serial.println(there[i]);
                        }
                }
                //calcCos(current position, desired position, step, servo array position)
                for (int i = 0; i < numServos; i++)
                {
                        angle = calcCos(here[i], there[i], 1.5, i);
                        if (doneMove[i] == false)
                        {
                                servo[i].write(angle);
                                delay(5);
                        }
                }
                pb = digitalRead(BUTTON);
        }


}
/*
THIS FUNCTION READS THE INTERNAL SERVO POTENTIOMETER and smooths the data
*/
int getFeedback(int a)
{
        int j;
        int mean;
        int result;
        int test;
        int reading[20];
        boolean done;

        for (j = 0; j < 20; j++)
        {
                reading[j] = analogRead(a); //get raw data from servo potentiometer
                delay(whlReading);
        } // sort the readings low to high in array
        done = false; // clear sorting flag
        while(done != true)  // simple swap sort, sorts numbers from lowest to highest
        {
                done = true;
                for (j = 0; j < 19; j++)
                {
                        if (reading[j] > reading[j + 1])  // sorting numbers here
                        {
                                test = reading[j + 1];
                                reading [j + 1] = reading[j] ;
                                reading[j] = test;
                                done = false;
                        }
                }
        }
```

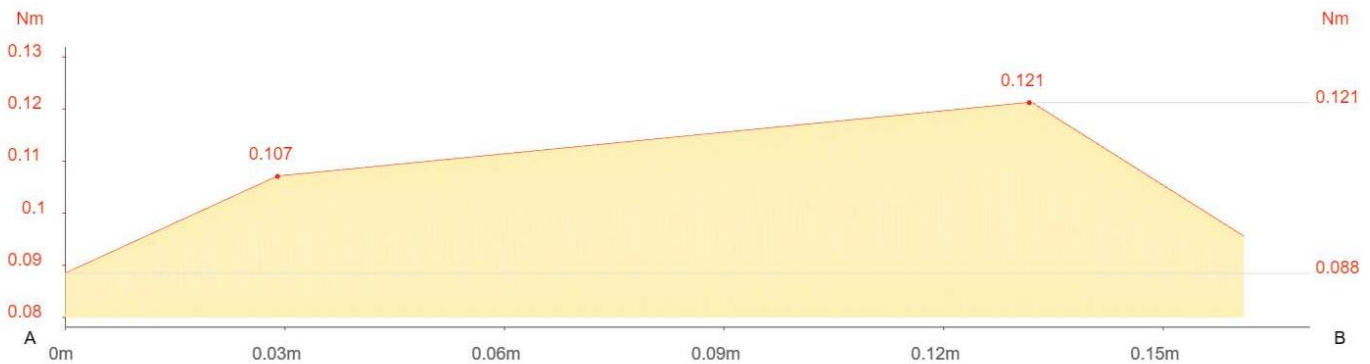
```
        mean = 0;
        for (int k = 6; k < 14; k++) //discard the 6 highest and 6 lowest readings
        {
                mean += reading[k];
        }
        result = mean / 8; //average useful readings
        return(result);
}    // END GET FEEDBACK

/*
THIS FUNCTION CREATES SMOOTH (COSINE) MOVEMENT FROM HERE TO THERE
*/
//calcCos(current position, desired position, step, servo array position)
int calcCos(int h, int th, float s, int n)
{
        int r;
        int a;
        if(h < th)
        {
                x[n] += s;
                if (x[n] >= 181)
                {
                        doneMove[n] = true;
                }
                r = (cos(radians(x[n])) * 100);
                a = map(r, 100, -100, h, t);
        }
        if(h > th)
        {
                x[n] -= s;
                if (x[n] <= -1)
                {
                        doneMove[n] = true;
                }
                r = (cos(radians(x[n])) * 100);
                a = map(r, -100, 100, h, t);
        }
        return a;
}     //END CALC COS
```

Appendix VIII: Force Effect Report

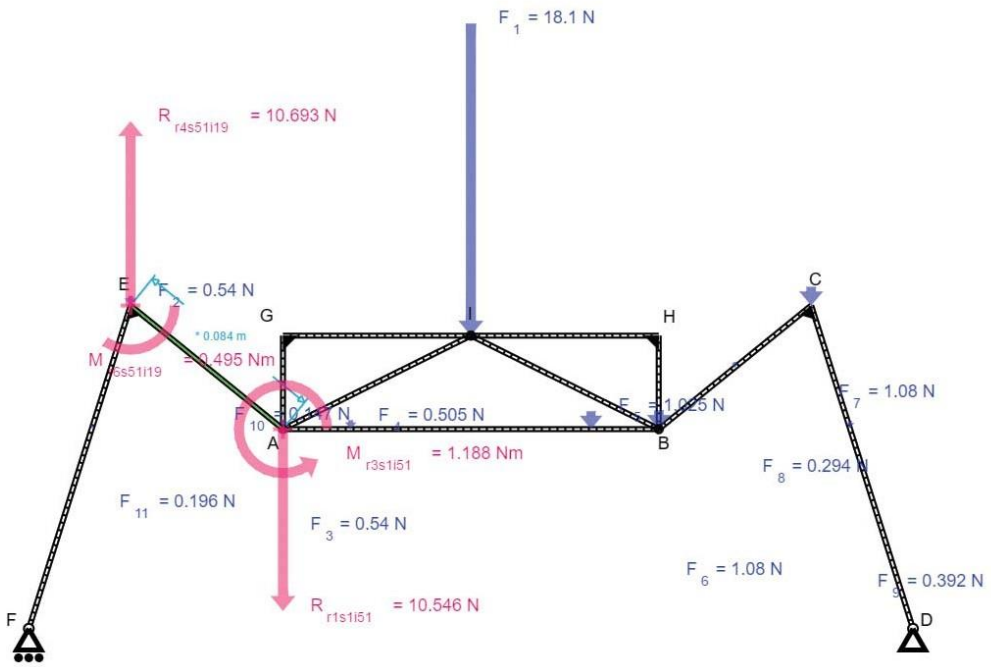


Results *(No equations available)*		
$M_{r3s1i28}$ = −0.718 Nm	$R_{r1s1i28}$ = 21.539 N	$R_{r2s1i35}$ = 6.664 N
$M_{r3s1i35}$ = −0.382 Nm	$R_{r1s1i35}$ = 27.713 N	$R_{r2s1i45}$ = 0.643 N
$M_{r3s1i45}$ = −0.088 Nm	$R_{r1s1i45}$ = −49.253 N	$R_{r2s1i51}$ = −10.546 N
$M_{r3s1i51}$ = 1.188 Nm	$R_{r1s1i51}$ = −0.000 N	$R_{r2s2i31}$ = 2.870 N
$M_{r3s2i31}$ = 0.752 Nm	$R_{r1s2i31}$ = −22.576 N	$R_{r2s2i35}$ = 5.867 N
$M_{r3s2i35}$ = 0.399 Nm	$R_{r1s2i35}$ = −26.676 N	$R_{r2s2i49}$ = −10.704 N
$M_{r3s2i49}$ = −1.246 Nm	$R_{r1s2i49}$ = −0.002 N	$R_{r4s46i2}$ = 49.255 N
$M_{r6s46i2}$ = 0.096 Nm	$R_{r2s1i28}$ = 2.699 N	$R_{r5s46i2}$ = 0.887 N

Shear Force and Moment Diagram

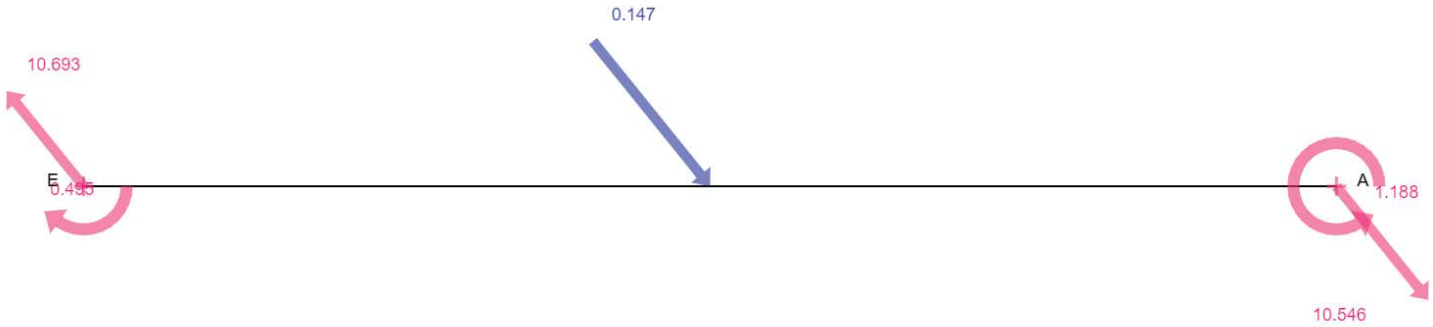


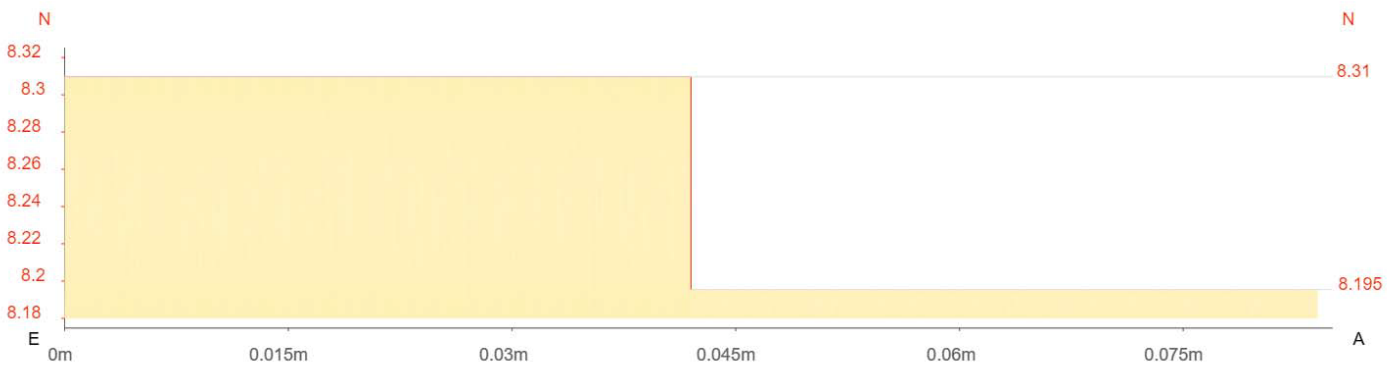
Element A-E

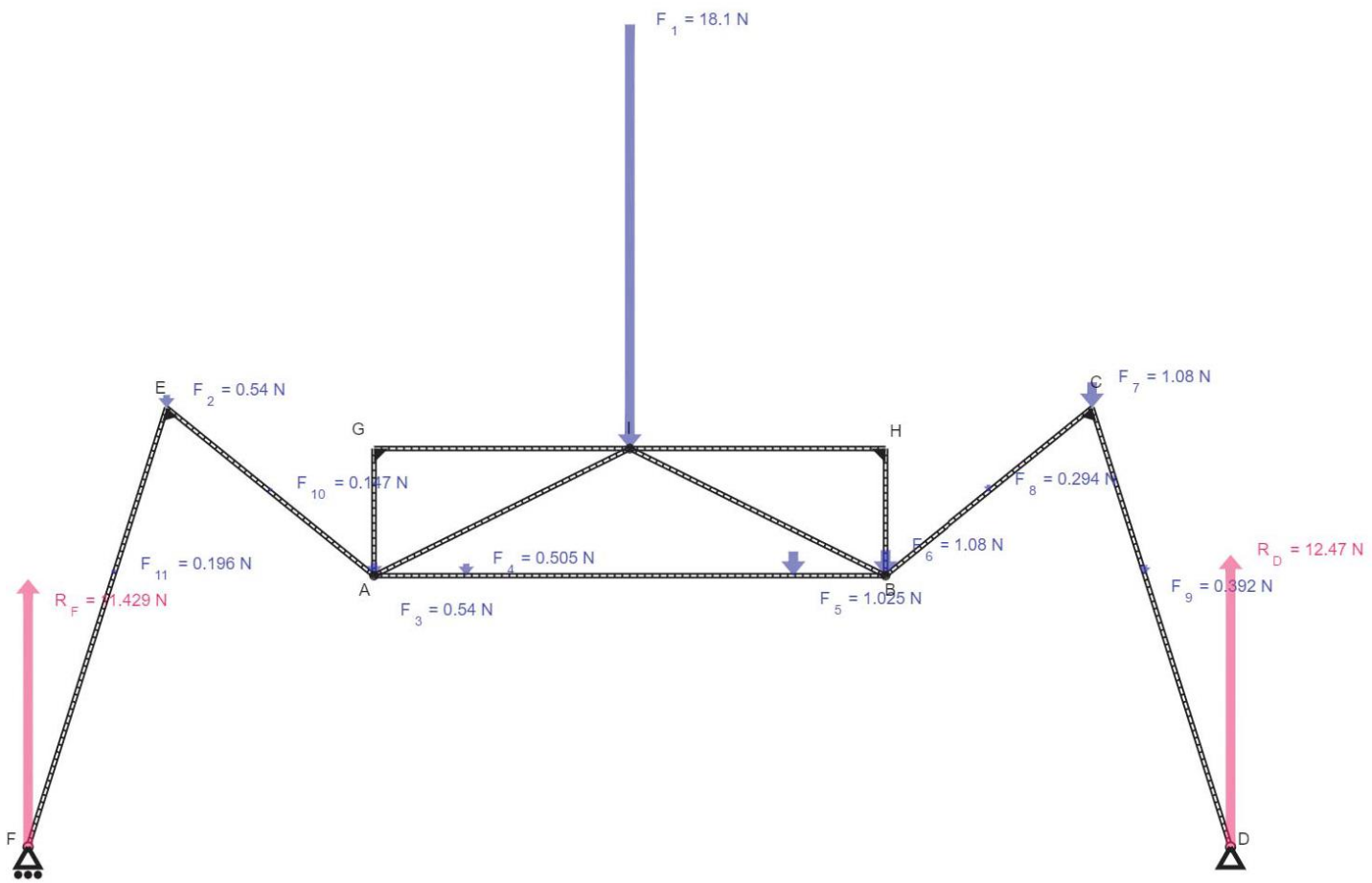


Results *(No equations available)*		
$M_{r3s19i52}$ = 0.495 Nm	$R_{r1s19i52}$ = −0.000 N	$R_{r2s1i28}$ = 2.699 N
$M_{r3s1i28}$ = −0.718 Nm	$R_{r1s1i28}$ = 21.539 N	$R_{r2s1i35}$ = 6.664 N
$M_{r3s1i35}$ = −0.382 Nm	$R_{r1s1i35}$ = 27.713 N	$R_{r2s1i45}$ = 0.643 N
$M_{r3s1i45}$ = −0.088 Nm	$R_{r1s1i45}$ = −49.253 N	$R_{r2s1i51}$ = −10.546 N
$M_{r3s1i51}$ = 1.188 Nm	$R_{r1s1i51}$ = −0.000 N	$R_{r4s51i19}$ = 0.000 N
$M_{r6s51i19}$ = −0.495 Nm	$R_{r2s19i52}$ = −11.233 N	$R_{r5s51i19}$ = 10.693 N

Shear Force and Moment Diagram







Inputs:

Elements

Element	Length	Weight
A-B	0.161 m	
B-C	0.084 m	
C-D	0.145 m	
A-E	0.084 m	
E-F	0.145 m	
A-G	0.040 m	
B-H	0.040 m	
G-H	0.161 m	
A-I	0.090 m	
B-I	0.090 m	

Forces

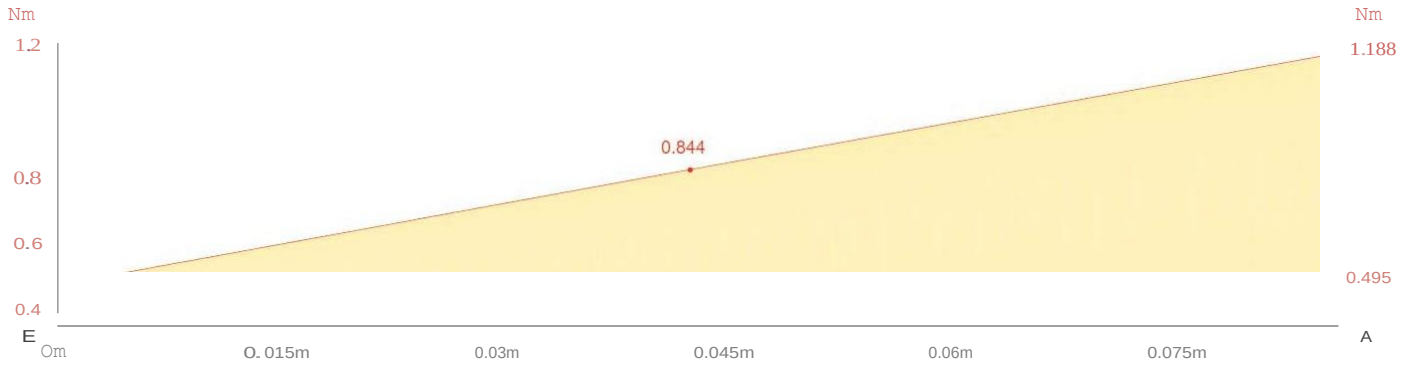
Force	Direction	Size	Angle
F_1	↓	18.100 N	270.0°
F_2	↓	0.540 N	270.0°
F_3	↓	0.540 N	270.0°
F_4	↓	0.505 N	269.3°
F_5	↓	1.025 N	270.2°
F_6	↓	1.080 N	270.0°
F_7	↓	1.080 N	270.0°
F_8	↓	0.294 N	270.0°
F_9	↓	0.392 N	270.0°
F_{10}	↓	0.147 N	270.0°
F_{11}	↓	0.196 N	270.0°

Results:

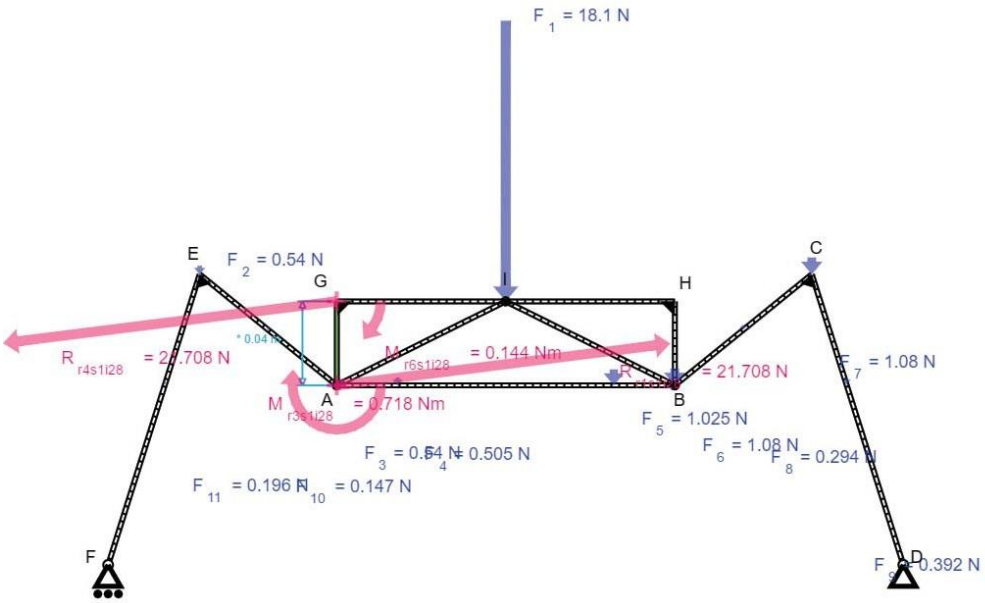
Overconstrained System: No equations available.

Reaction Forces

Force	Direction	Size	Angle
R_D	↑	12.470 N	90.0°
R_F	↑	11.429 N	90.0°


Nm
1.2
0.8
0.6
0.4
E
Nm
1.188
0.844
0.495
A
Om
O. 015m
0.03m
0.045m
0.06m
0.075m

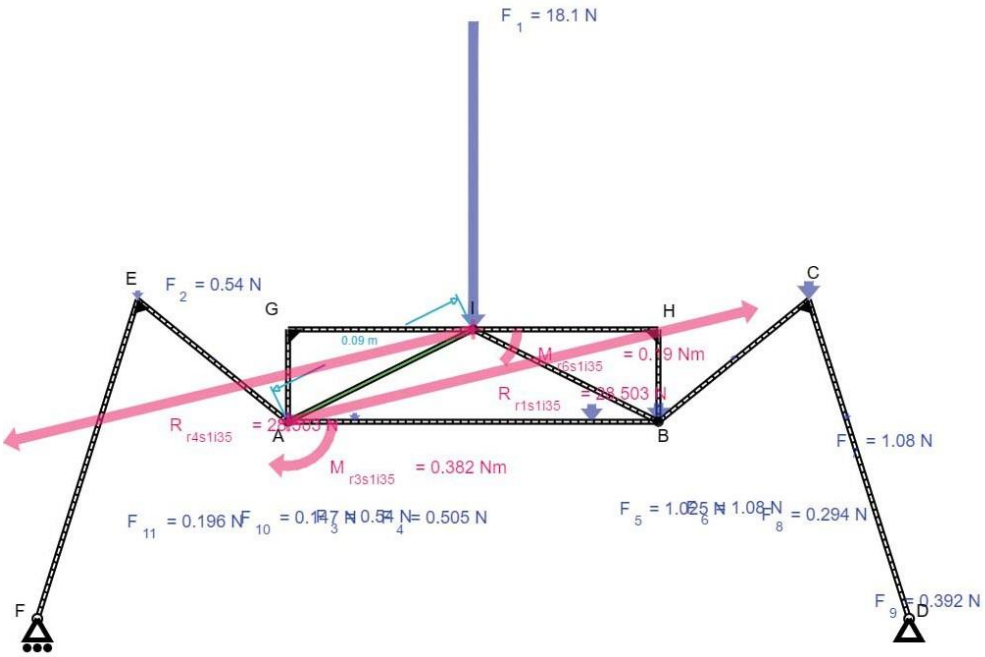
Element A-G



Results *(No equations available)*		
$M_{r3s1i28}$ = −0.718 Nm	$R_{r1s1i28}$ = 21.539 N	$R_{r2s1i35}$ = 6.664 N
$M_{r3s1i35}$ = −0.382 Nm	$R_{r1s1i35}$ = 27.713 N	$R_{r2s1i45}$ = 0.643 N
$M_{r3s1i45}$ = −0.088 Nm	$R_{r1s1i45}$ = −49.253 N	$R_{r2s1i51}$ = −10.546 N
$M_{r3s1i51}$ = 1.188 Nm	$R_{r1s1i51}$ = −0.000 N	$R_{r2s28i35}$ = 2.699 N
$M_{r3s28i35}$ = 0.144 Nm	$R_{r1s28i35}$ = 21.539 N	$R_{r4s1i28}$ = −21.539 N
$M_{r6s1i28}$ = −0.144 Nm	$R_{r2s1i28}$ = 2.699 N	$R_{r5s1i28}$ = −2.699 N

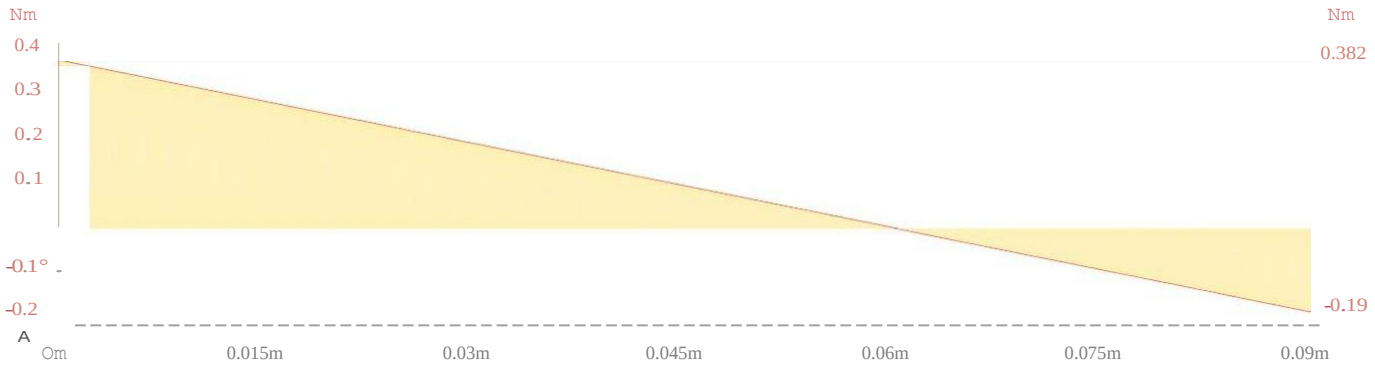
Shear Force and Moment Diagram

Element A-I

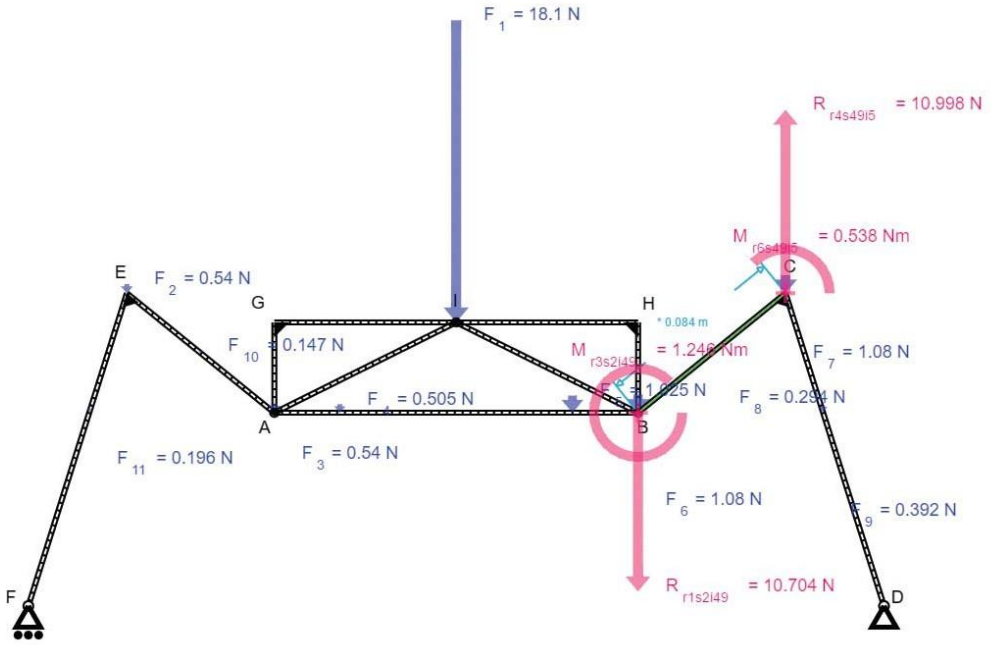


Results *(No equations available)*		
$M_{r3s1i28}$ = −0.718 Nm	$R_{r1s1i28}$ = 21.539 N	$R_{r2s1i45}$ = 0.643 N
$M_{r3s1i35}$ = −0.382 Nm	$R_{r1s1i35}$ = 27.713 N	$R_{r2s1i51}$ = −10.546 N
$M_{r3s1i45}$ = −0.088 Nm	$R_{r1s1i45}$ = −49.253 N	$R_{r4s1i35}$ = −27.713 N
$M_{r3s1i51}$ = 1.188 Nm	$R_{r1s1i51}$ = −0.000 N	$R_{r4s2i35}$ = 26.676 N
$M_{r6s1i35}$ = −0.190 Nm	$R_{r2s1i28}$ = 2.699 N	$R_{r5s1i35}$ = −6.664 N
$M_{r6s2i35}$ = 0.196 Nm	$R_{r2s1i35}$ = 6.664 N	$R_{r5s2i35}$ = −5.867 N

Shear Force and Moment Diagram

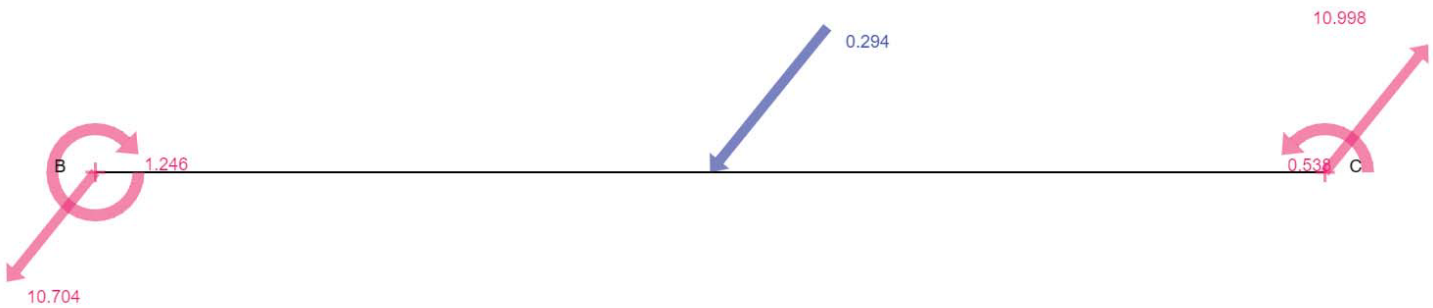

Nm
Nm
0.4
0.3
0.2
0.1
0.382
-0.1°
-0.2
-0.19
A
Om
0.015m
0.03m
0.045m
0.06m
0.075m
0.09m

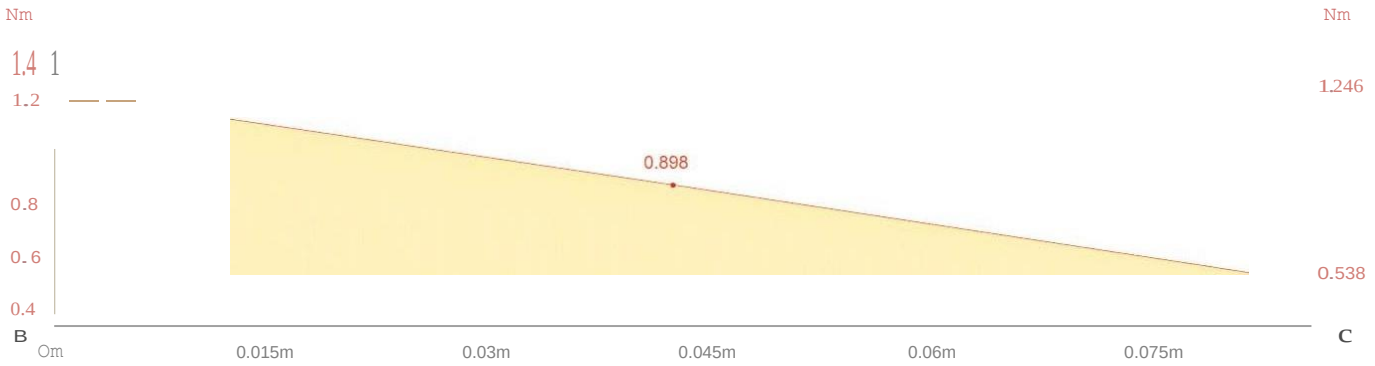
Element B-C



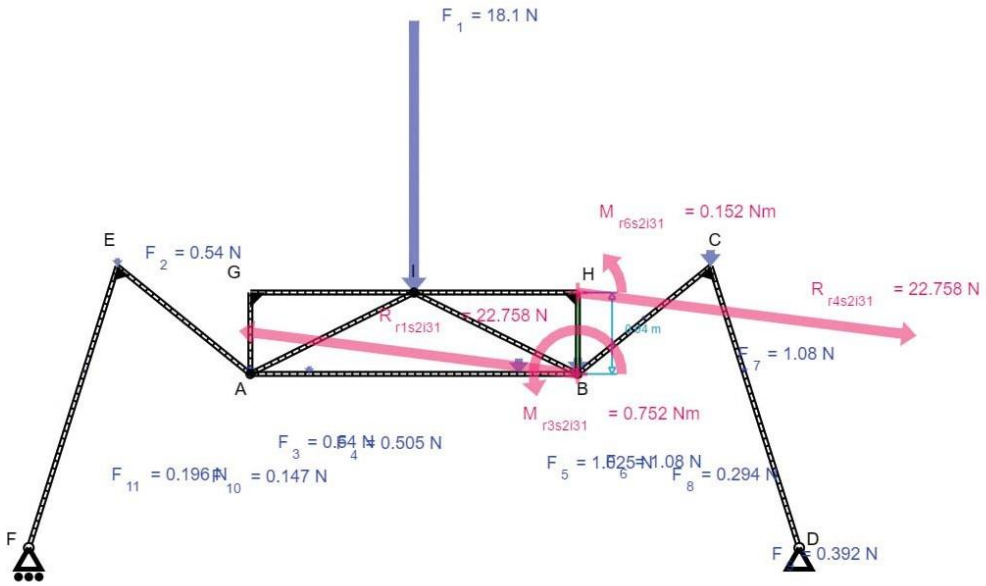
Results *(No equations available)*		
$M_{r3s2i31}$ = 0.752 Nm	$R_{r1s2i31}$ = −22.576 N	$R_{r2s2i49}$ = −10.704 N
$M_{r3s2i35}$ = 0.399 Nm	$R_{r1s2i35}$ = −26.676 N	$R_{r2s5i50}$ = −12.078 N
$M_{r3s2i49}$ = −1.246 Nm	$R_{r1s2i49}$ = −0.002 N	$R_{r4s46i2}$ = 49.255 N
$M_{r3s5i50}$ = −0.538 Nm	$R_{r1s5i50}$ = −0.002 N	$R_{r4s49i5}$ = 0.002 N
$M_{r6s46i2}$ = 0.096 Nm	$R_{r2s2i31}$ = 2.870 N	$R_{r5s46i2}$ = 0.887 N
$M_{r6s49i5}$ = 0.538 Nm	$R_{r2s2i35}$ = 5.867 N	$R_{r5s49i5}$ = 10.998 N

Shear Force and Moment Diagram




Nm
Nm
1.4
1.2
0.8
0.6
0.4
1.246
0.898
0.538
B
C
0m
0.015m
0.03m
0.045m
0.06m
0.075m

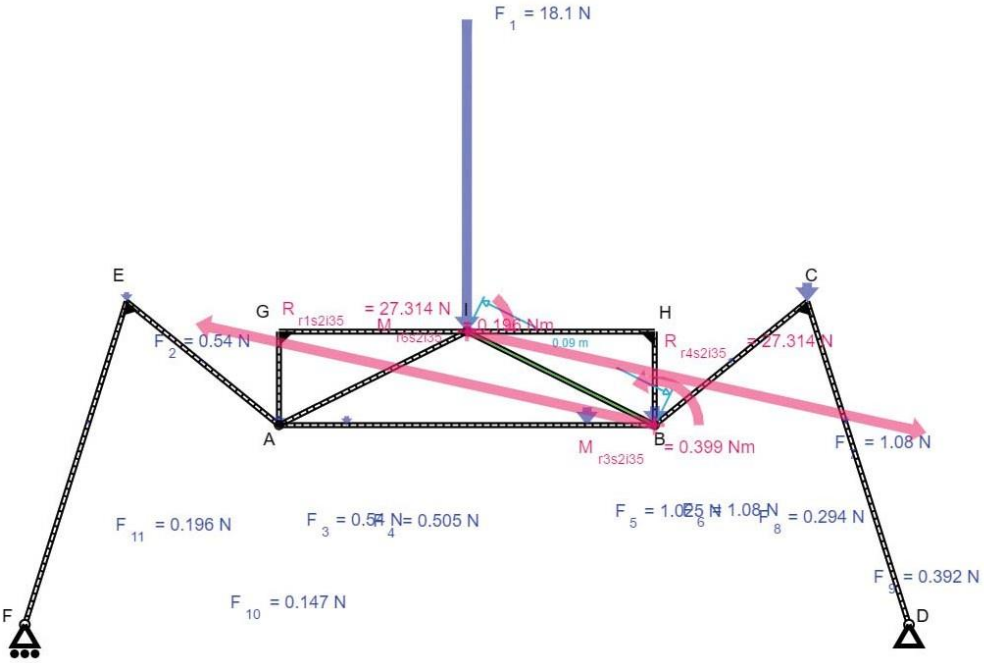
Element B-H



Results *(No equations available)*		
$M_{r3s2i31}$ = 0.752 Nm	$R_{r1s2i31}$ = −22.576 N	$R_{r4s2i31}$ = 22.576 N
$M_{r3s2i35}$ = 0.399 Nm	$R_{r1s2i35}$ = −26.676 N	$R_{r4s35i31}$ = −22.576 N
$M_{r3s2i49}$ = −1.246 Nm	$R_{r1s2i49}$ = −0.002 N	$R_{r4s46i2}$ = 49.255 N
$M_{r6s2i31}$ = 0.152 Nm	$R_{r2s2i31}$ = 2.870 N	$R_{r5s2i31}$ = −2.870 N
$M_{r6s35i31}$ = −0.152 Nm	$R_{r2s2i35}$ = 5.867 N	$R_{r5s35i31}$ = 2.870 N
$M_{r6s46i2}$ = 0.096 Nm	$R_{r2s2i49}$ = −10.704 N	$R_{r5s46i2}$ = 0.887 N

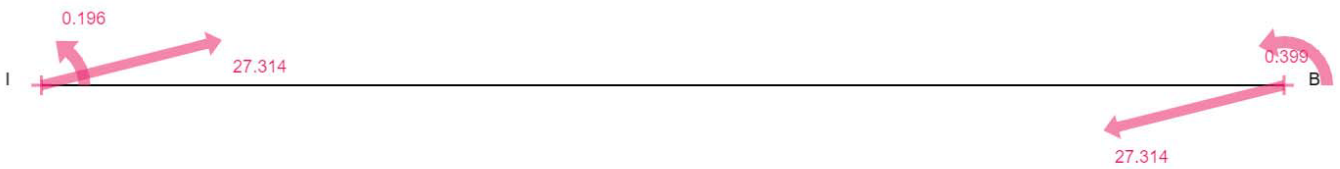
Shear Force and Moment Diagram

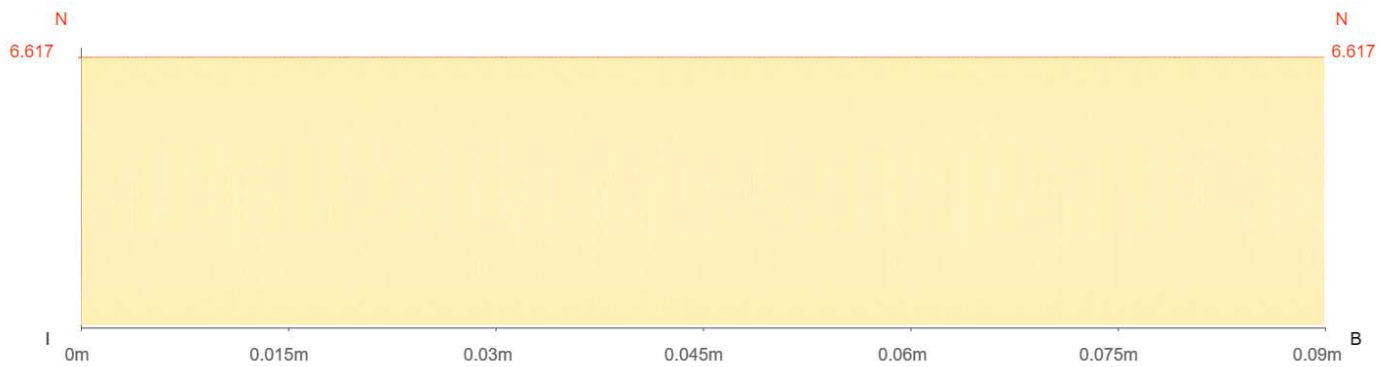
Element B-I

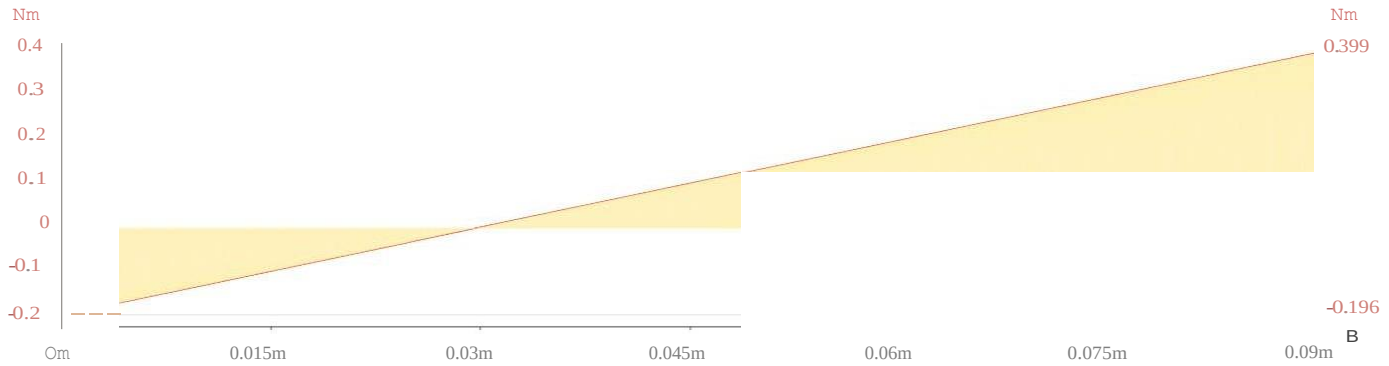


Results *(No equations available)*		
$M_{r3s2i31}$ = 0.752 Nm	$R_{r1s2i31}$ = −22.576 N	$R_{r4s1i35}$ = −27.713 N
$M_{r3s2i35}$ = 0.399 Nm	$R_{r1s2i35}$ = −26.676 N	$R_{r4s2i35}$ = 26.676 N
$M_{r3s2i49}$ = −1.246 Nm	$R_{r1s2i49}$ = −0.002 N	$R_{r4s46i2}$ = 49.255 N
$M_{r6s1i35}$ = −0.190 Nm	$R_{r2s2i31}$ = 2.870 N	$R_{r5s1i35}$ = −6.664 N
$M_{r6s2i35}$ = 0.196 Nm	$R_{r2s2i35}$ = 5.867 N	$R_{r5s2i35}$ = −5.867 N
$M_{r6s46i2}$ = 0.096 Nm	$R_{r2s2i49}$ = −10.704 N	$R_{r5s46i2}$ = 0.887 N

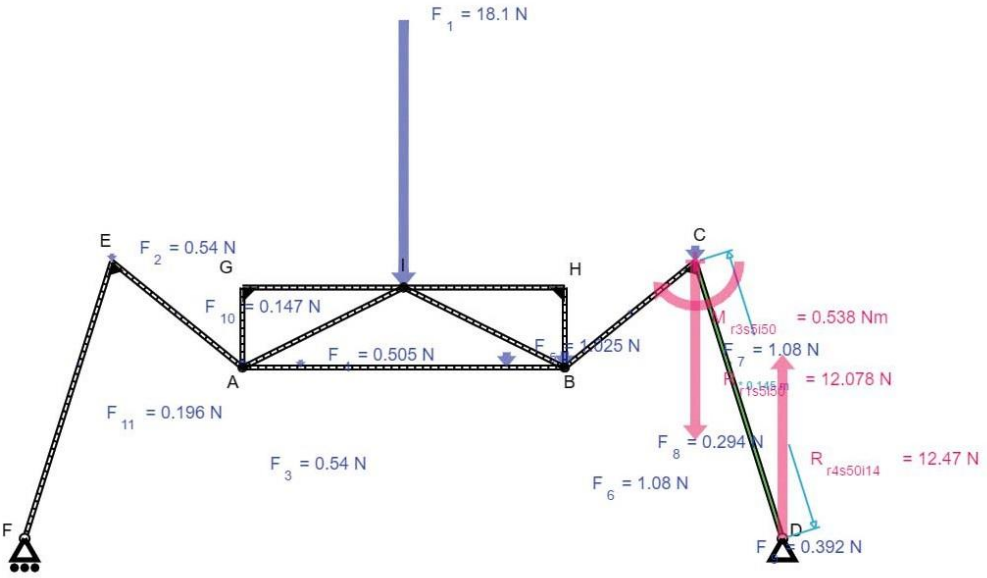
Shear Force and Moment Diagram





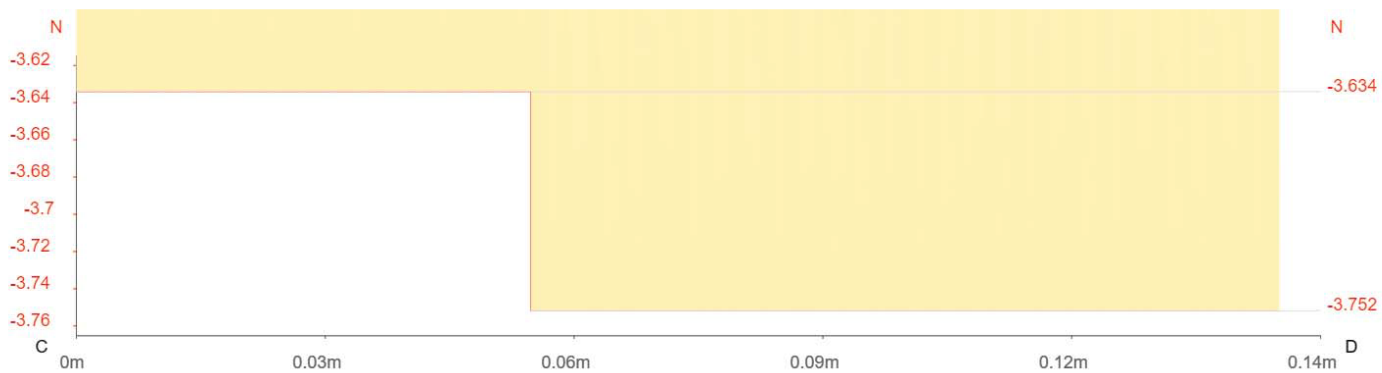

Nm
0.4
0.3
0.2
0.1
0
-0.1
-0.2
Nm
0.399
-0.196
B
Om
0.015m
0.03m
0.045m
0.06m
0.075m
0.09m

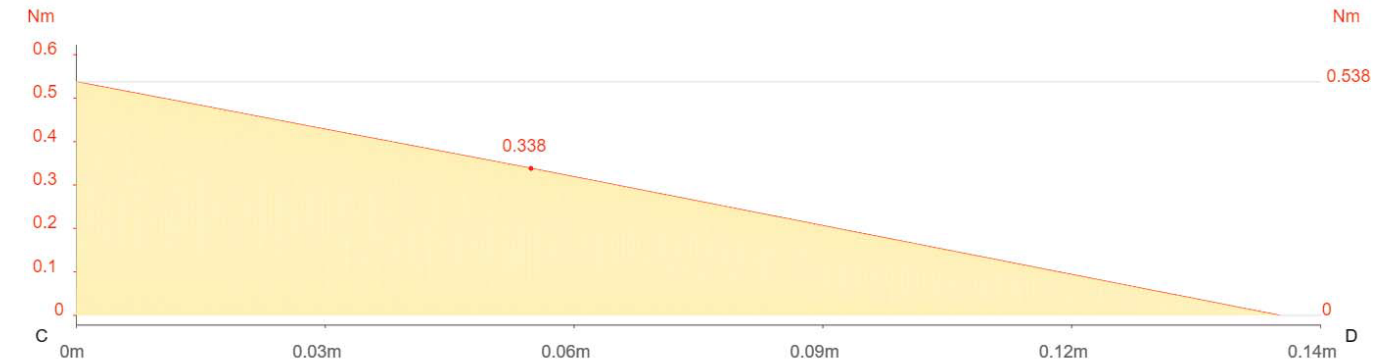
Element C-D



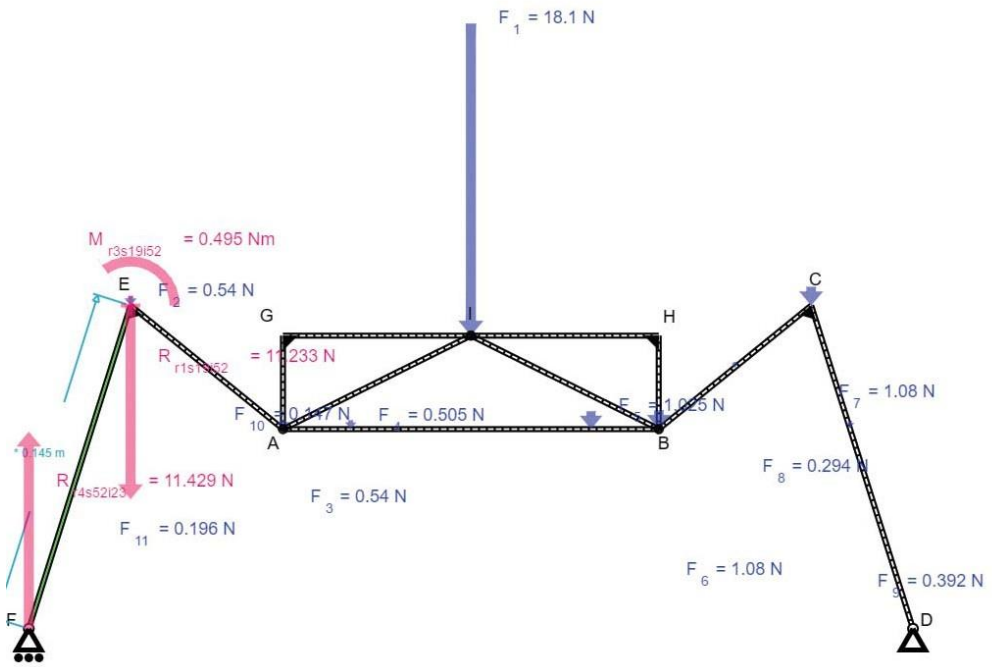
Results *(No equations available)*		
$M_{r3s5i50}$ = −0.538 Nm	$R_{r1s5i50}$ = −0.002 N	$R_{r4s50i14}$ = 0.002 N
$M_{r6s49i5}$ = 0.538 Nm	$R_{r2s5i50}$ = −12.078 N	$R_{r5s49i5}$ = 10.998 N
$M_{r6s50i14}$ = −0.000 Nm	$R_{r4s49i5}$ = 0.002 N	$R_{r5s50i14}$ = 12.470 N

Shear Force and Moment Diagram



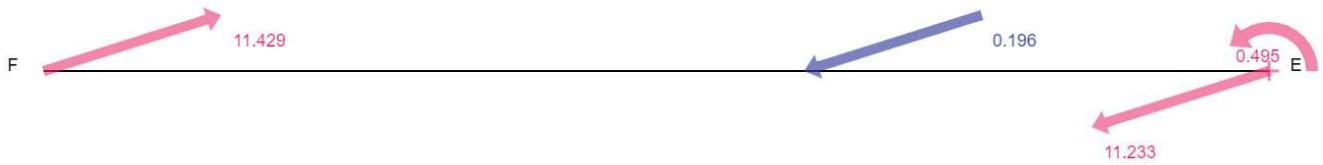


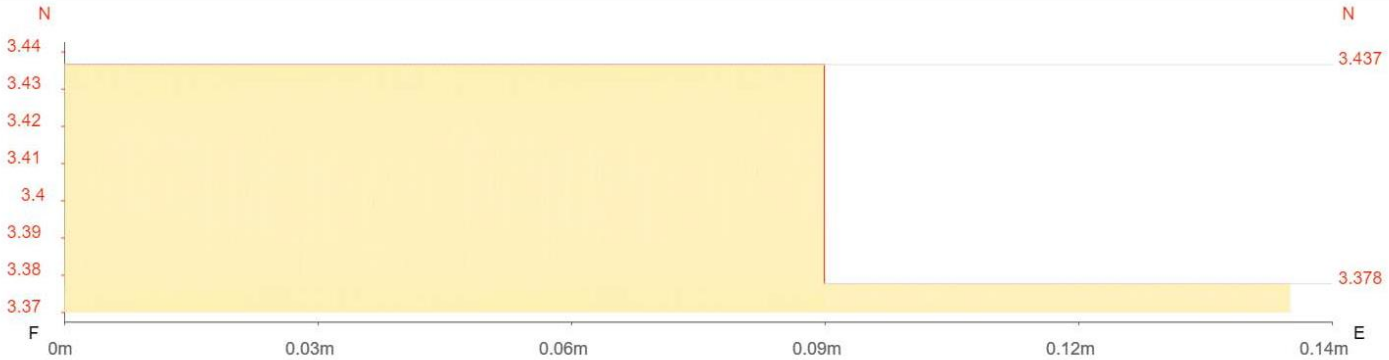
Element E-F

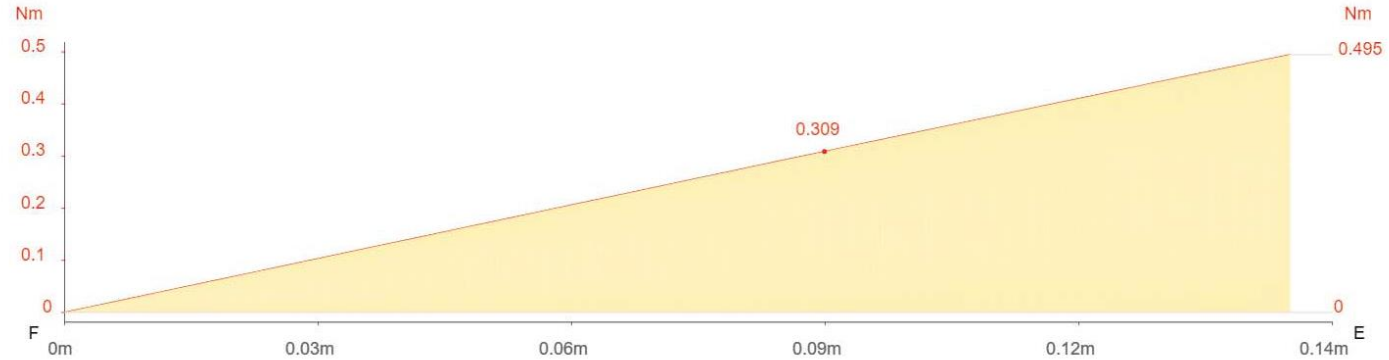


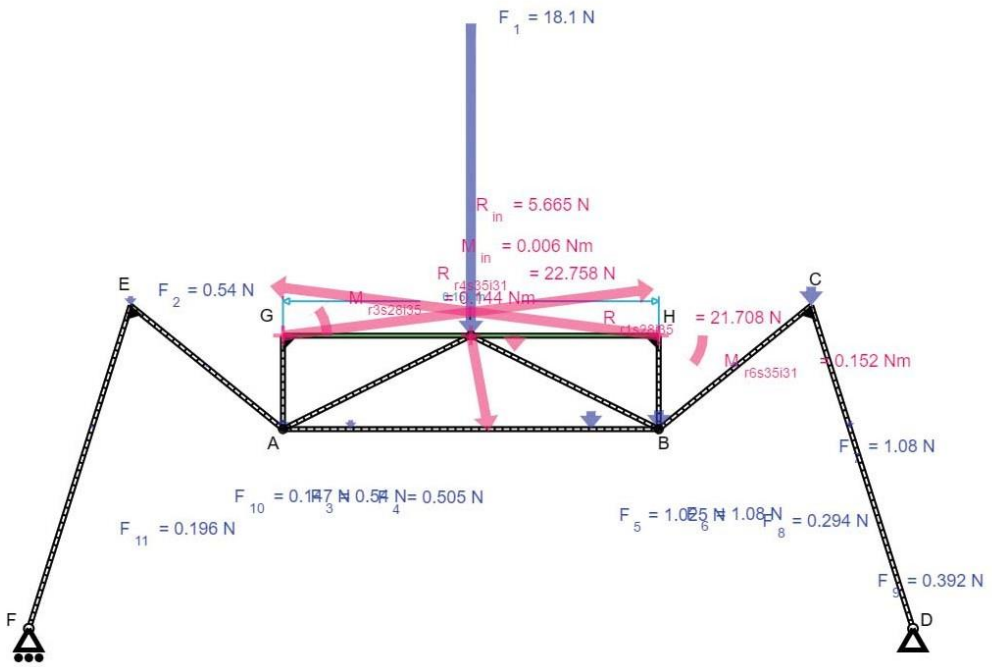
Results *(No equations available)*		
$M_{r3s19i52}$ = 0.495 Nm	$R_{r1s19i52}$ = −0.000 N	$R_{r4s52i23}$ = 0.000 N
$M_{r6s51i19}$ = −0.495 Nm	$R_{r2s19i52}$ = −11.233 N	$R_{r5s51i19}$ = 10.693 N
$M_{r6s52i23}$ = 0.000 Nm	$R_{r4s51i19}$ = 0.000 N	$R_{r5s52i23}$ = 11.429 N

Shear Force and Moment Diagram



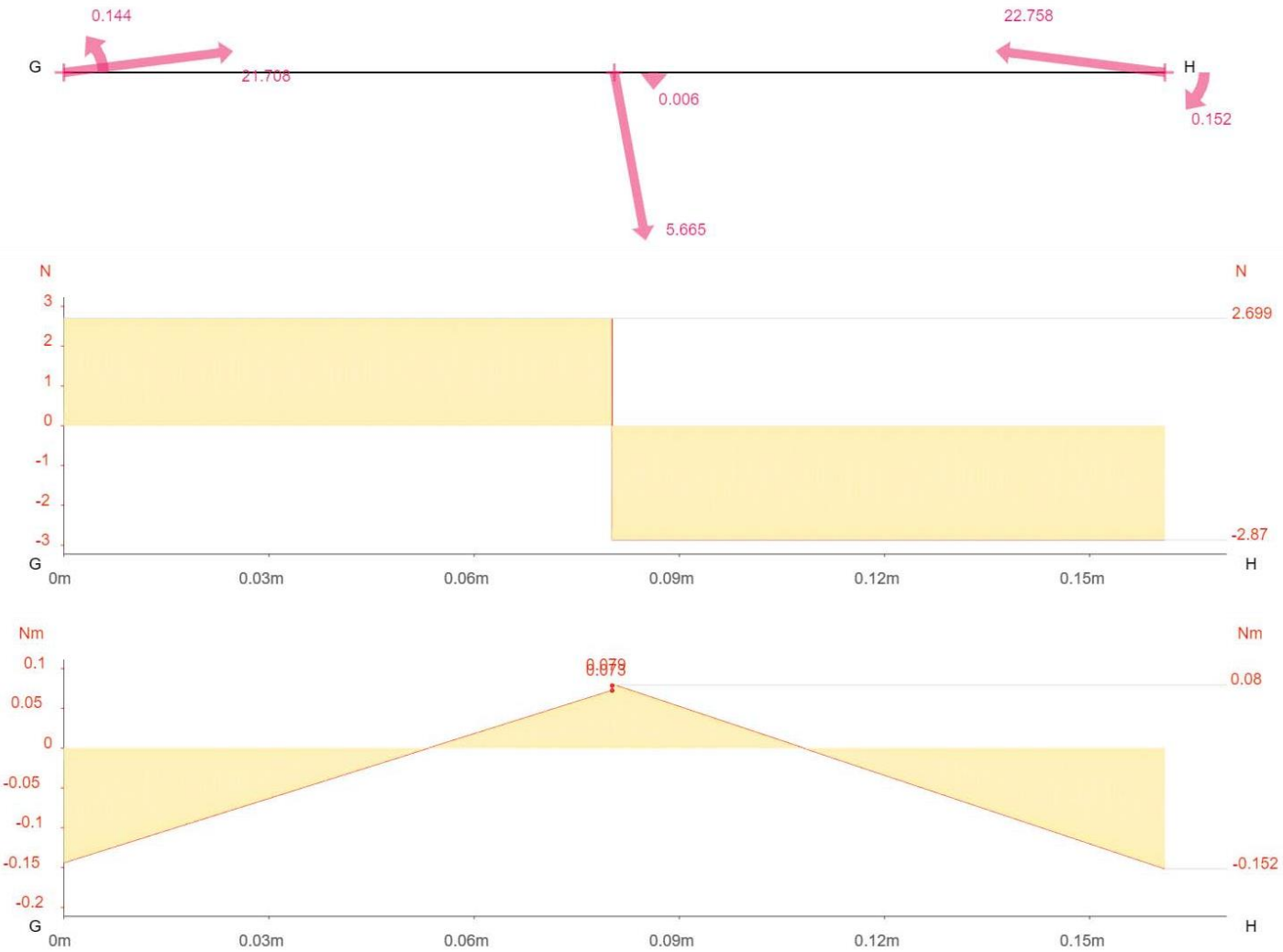






Results *(No equations available)*		
$M_{r3s28i35}$ = 0.144 Nm	$R_{r1s28i35}$ = 21.539 N	$R_{r4s35i31}$ = −22.576 N
$M_{r6s1i28}$ = −0.144 Nm	$R_{r2s28i35}$ = 2.699 N	$R_{r5s1i28}$ = −2.699 N
$M_{r6s2i31}$ = 0.152 Nm	$R_{r4s1i28}$ = −21.539 N	$R_{r5s2i31}$ = −2.870 N
$M_{r6s35i31}$ = −0.152 Nm	$R_{r4s2i31}$ = 22.576 N	$R_{r5s35i31}$ = 2.870 N

Shear Force and Moment Diagram



Appendix IX: Dynamixel Smart Servos

	The Robotis Dynamixel AX-18A Robot Servo is a very innovative robot servo. This specific actor is known for being the fastest in terms of positioning. The Robotis Dynamixel AX-18A is the successor of the Dynamixel AX-12A. The Dynamixel AX-18A has nearly twice the speed of the AX-12A and about 15% more torque. The Dynamixel AX-18A fits into all brackets of his predecessor. Furthermore there is another quite interesting feature. The control algorithm can be adjusted individually with 50 commands for each servo, so you can control the speed and strength of the motor's response easily. The Robotis Dynamixel AX-18A Robot Servo is able to track its speed, temperature, shaft position, voltage and load. The Robotis Dynamixel AX-18A handles the sensor management and position controller. This distributed approach leaves your main controller free to perform other functions. We also sell the Robotis Dynamixel MX-28T and other products of the Robotis Dynamixel series. Please note that the assembly and bracket hardware is not included to the kit of the Robotis Dynamixel AX-18A. Technical Data: 18 kgcm holding torque (12V) 96rpm 300 degrees rotation and endless RS232 single wire (see manual) bus, daisy chainable reports position, velocity, current, temperature 9-12V weight 56gr max current 2.2A coreless motor
	Features High Speed and High Torque Full Metal gears Dual ball bearing Coreless Motor RS-485 communication Rear Pivot mount for superior support Four 2.5mm mouting holes allowing servo to be mounted positively Specifications Torque : 26 kg.cm @ 12V, 2.4A Speed : 126rpm Length : 50.6mm Breadth : 35.6mm Height : 35.5mm Weight : 67 grams Operating Angle : 360 Deg Operating Voltage : 9V-12V Gear Ration : 1 : 193

	Features Good Speed and High Torque Full Metal gears Dual ball bearing Maxon Motor RS-485 communication Rear Pivot mount for superior support Four 2.5mm mouting holes allowing servo to be mounted positively Includes 1pc of RX-28 Dynamixel 1pc of Wire 1pc of Metal Horn type N101 (NOT compatible with old OF- frames) Misc Hardware Instruction manual Specifications (General Tolerance of 1mm and 5g applies) Torque : 37kg.cm @ 18.5V, 1.9A Speed : 67rpm Length : 50.6mm Breadth : 35.6mm Height : 35.5mm Weight : 72 grams Operating Angle : 360 Deg Operating Voltage : 12V-18.5V Gear Ration : 1 : 193

Table 8: Dynamixel Specifications