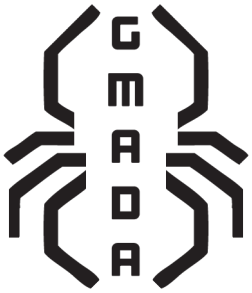
Report on Refinement of the Hexapod Robot: Enabling Full Control



To:

The University of Texas at Arlington

By:
General Mechatronics and Automation Design Associates dba. GMADA

August 9, 2018

Report on Refinement of Hexapod Robot:
Enabling Full Control

The objective of this report is to detail the means and methods by which GMADA has further enabled control of the University of Texas at Arlington's hexapod robot system. The areas of work include joint repair, component selection including: batteries, sensors, and controller hardware. The team also developed the software architecture needed to control all legs via a PlayStation 4 controller. Additional work was done on the collision avoidance aspect of gait analysis. By completing each of these tasks, GMADA has furthered the hexapod's development.

GMADA TEAM MEMBERS

Paul Bullard – Team Lead
Bryan.bullard@mavs.uta.edu – 713-679-8364

Chris Campbell – Mechanical Design
christopherscampbell@mavs.uta.edu – 817-806-7197

David Nguyen – Software Design
Dung.nguyen29@mavs.uta.edu – 626-560-9351

Peter Irvin – Gait and Inverse Kinematics
Peter.irvin@mavs.uta.edu – 972-672-0633

Kendall Prewitt – Mechanical Analysis
Kendall.prewitt@mavs.uta.edu – 214-799-2700

Faculty Advisors

Dr. Bryan Huff
bhuff@uta.edu

Dr. Robert Woods
woods@uta.edu

TABLE OF CONTENTS

1. INTRODUCTION

1.1 BACKGROUND

There are countless robotic systems that exist today. From space probes, to industrial manipulators, and self-driving cars, robot ubiquity is increasing at a staggering rate. What sets hexapod robots apart is an obvious design feature: they walk on six legs. It is the stability and versatility provided by this insectoid design that makes hexapod robots a common topic of research in both academia and industry. While applications are limited for this technology, it provides a rich learning platform for developing successful, robust systems.

1.2 MOTIVATION

The University of Texas at Arlington (UTA) owns a hexapod robot. This device was built by two previous senior design groups. The robot should be able to stand and walk under independent power supply and control. The core function of UTA's hexapod robot is generating design ability in students and displaying that ability to prospective students, faculty, and investors.

However, the project, which started the Spring of 2011, is in a state of serious decay. This is due to unforeseen challenges experienced by the previous groups. The list of physical components in need of replacing or repair included the joints, batteries, wiring, and digital control hardware. The only functioning sensors on the system were optical encoders.

The purpose of this project is the revitalization of the defunct hexapod system. This work should enable future development teams and succinctly provide the foundational work completed by prior teams. GMADA's efforts will enable the system to once again provide experience to students and prestige to the university.

1.3 DESIGN OBJECTIVES

GMADA has been tasked with refurbishing the hexapod system so that it can finally walk. This is not a trivial undertaking. The preliminary work in Spring 2018 enabled the full control over one leg with remote commands. The primary design objective is the utilization of spring design decision to enable full control of the system. The following general tasks are required: installation of all control hardware, design and manufacture of bracketing, cable management, power grid wiring, communication grid wiring, calibration of sensors, programming of sub controllers, programming of main controller, design gait algorithm, design general leg motion, integrate PS4 controller with Raspberry Pi. These are split into formally defined tasks in the signed proposal between GMADA and the client. In the proposal, all definitions regarding task success and completion are defined and available.

Refer to the Future Development Section for the project results. This section of the report will aid future teams in planning their semester's tasks and goals. As the primary goal is to enable future students, this document must provide complete comprehension for future teams.

2. TECHNICAL DETAILS

2.1 MECHANICAL DETAILS

This system was conceived and built from 2011 to 2012 by engineering capstone teams. These teams completed the design and fabrication of all carbon fiber components, the selection of motors, gears, belts, and motor drivers. The design lacks finesse and many factors were overlooked: ease of disassembly, motor power, aesthetic appeal, etc.

Figure 1. Model of Hexapod System

GMADA built upon the original system while discarded some elements. The carbon frame, motors, leg brackets, foot design, hip-box gear assemblies, and the onboard belt-drive assemblies were salvaged from the original system. These components were integrated with new control hardware and power system hardware. However, the performance of the joints and gears proved unsatisfactory and considerable effort was expended analyzing the design. These efforts are detailed in Section 2.1.1.

2.1.1 JOINT REFINEMENT

The original system suffered from loose movement in all joints. This was determined to be caused by an unfortunate axle design in the femur (J2), poor meshing of worm gear pair, and loosely mounted housings. The system consists of 18 joints with as many motors. It was a key requirement of the client to retain the maximum amount of the original hardware to reduce cost and development time.

A shaft redesign (details in the spring report and solid models available in files) was manufactured by the UTA machine shop in Spring 2018 which featured: D-profile, spot faces for set screws, larger set screw size (from #4 to #8) and adjusted tolerances. This

solution was completed and propagated to all J2 joints. This reduced slop movement from roughly 30° per joint to 5° per joint.

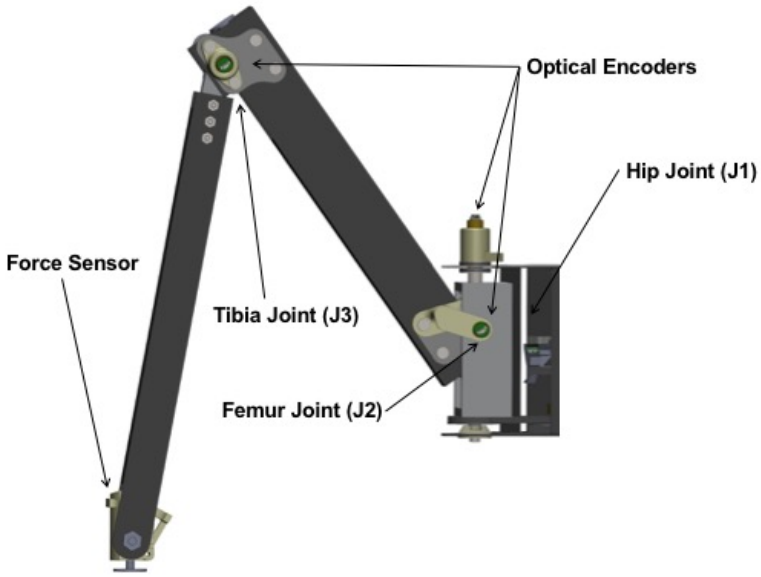


Figure 2. Model of a single leg

Further efforts were expended to resize fasteners that hold the motors as they are undersized. It was thought that this would reduce potential wobble in the motor and thus increase joint stiffness. Precisely sized bolts were obtained and installed in one joint, but no improved was observed. These efforts were abandoned and updated fasteners were not propagated to other joints.

2.1.2 GEARS AND BELTS

The worm gear, worm-drive gear, motors, and a few other components were salvaged from industrial robots by the first teams. Thus there are no documents or specifications available for the gears. Further efforts to reduce slop below 5° by adjusting the meshing between the gears failed to produce meaningful results. The correct center distance for the salvage gears is not known. The current slop was determined as acceptable for basic movement.

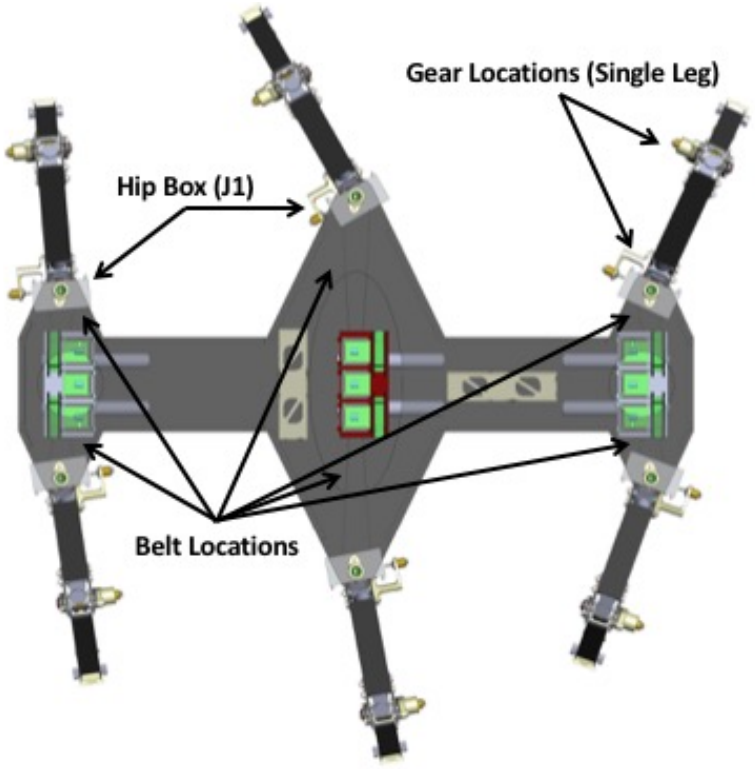


Figure 3. Top-View of System

The reinforced belts in J1 joints are commercially available and replacement belts were ordered in Summer 2018 (details in Appendix and BoM) to replace two failed belts. Slop movement in these joints was eliminated by tightening the housings within the hexapod frame.

2.1.3 BRACKETS AND PLATES

Rapid prototyping methods were utilized to produce joint plate brackets and hardware mounting brackets. These methods include laser cutting (Acrylic) and 3D printing (ABS). These brackets hold the motor drivers, sub controllers, time of flight sensors, and sensor conditioning circuit. The previous teams designed brackets for the encoders, force sensors, and batteries. The solid models are available in the files.

The sub controller bracket holds 2 sub controllers and 3 motor drivers. This part is 3D printed from ABS and is designed with snap-fit connections to hold the components. Additionally, it features snap-fit feet to attach itself to the onboard belt-drive assemblies. This allows it to float freely, while the belt-drive assembly can be calibrated to ensure proper belt tension and joint performance. The models are provided in the files and 3D printer settings. The snap-fits require a certain print orientation to ensure strength.

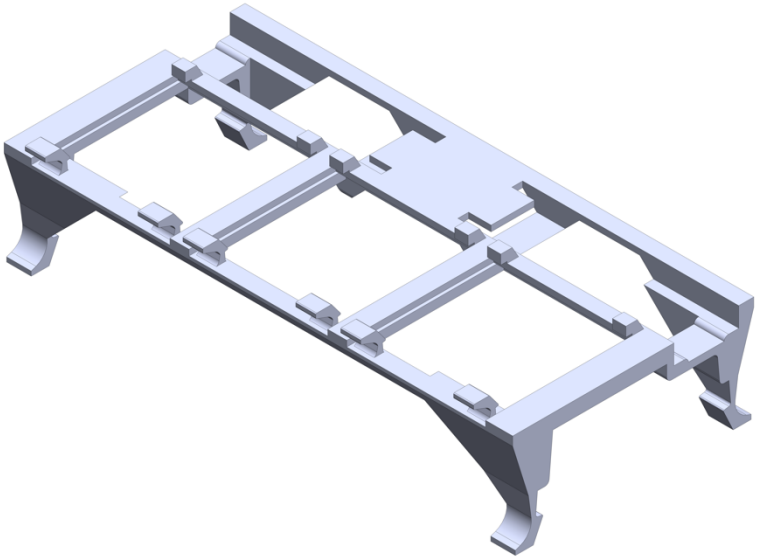


Figure 4. Sub controller Bracket

The time of flight bracket is mounted to the leg using an aggressive double sided tape. Each bracket holds one time of flight sensor. This part is 3D printed from ABS. The model is available in the 'Bracket Models' File.

Figure 5. Femur and Tibia Bracket

Efforts to correct gear meshing in Spring 2018 by changing the center distance of the gears required manufacturing new plates for J2 joints. These joints have a curvy, black bracket that sets the shaft distances. As no information is available on the gears, the distance was iterated in multiple sets of laser cut brackets. The laser lacked precision and CNC machining was attempted by the UTA Machine Shop. These efforts ultimately failed to achieve any better performance. Currently, there is ~5° of slop in most joints (if all set screws are adequately tightened).

2.2 CONTROL HARDWARE DETAILS

2.2.1 CONTROLLERS

Per the client, all the controllers that were used by the previous senior design group were manufactured by a company that is no longer in business. Therefore, new controllers for the robot had to be selected. The previous senior design group used nine sub-controllers so that each sub-controller sent PWM signals to a single motor driver. However, GMADA has opted to use a six sub-controller configuration so that each sub-controller will control an entire leg of the robot. Next, the main controller that coordinates the robot's legs also needed to be replaced. However, it was determined that different types of controllers could be selected for the main and sub-controllers because they each performed very different roles. This caused some initial difficulty in deciding which controller setups to choose from.

For the sub-controllers, a wide range of controllers from multiple different controller families were examined. Ultimately, the Teensy 3.5, which is part of the Arduino family of controllers, was chosen to be the robot's sub-controller. The Teensy was selected because of its small size, relatively fast processor, familiar programming language, and extensive documentation. The Teensy was not the fastest controller that was considered, but it had better documentation and a more user-friendly programming system than any of the faster options. Because there are no experts in programming on the team, the Teensy was selected to save time on the programming and troubleshooting portions of the project.

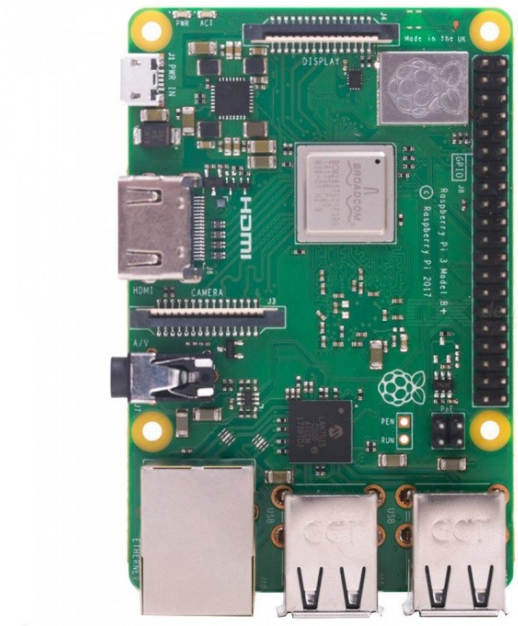


Figure 6. Main Controller: Raspberry Pi 3b+

Figure 7. Teensy 3.5

Evaluating the options for the main controller forced the team to reexamine the potential sub-controllers. Small form factor computers were likely candidates for the main controller. However, the small form factor computers were eliminated as options because they were much more expensive and had much higher power requirements than the other options. Ultimately, the Raspberry Pi 3B+ was selected as the hexapod's main controller. The Raspberry Pi has built in wireless Ethernet, a built-in Ethernet port, built in Bluetooth,

is well documented, and has the best processor and memory specifications of the controllers that GMADA considered.

By selecting controllers that were from commonly used controller families with good documentation, the amount of time needed to program and troubleshoot the robot was, is, and will be minimized. This is accomplished due to the large user base and sizeable repositories of published projects for each of the controller families. Additionally, some team members were already familiar with the controller families that the sub-controller and main controller were from. This was also factored into the selection process because it decreased the amount of time needed to program the controllers. It is for the above reasons that GMADA has selected and implemented these controllers within the robot.

After obtaining this hardware, a bracket was designed and printed that could hold it within the body of the robot. The bracket would hold three motor drives and two Teensy sub controllers and would be placed right above the brackets and belts used to drive the hip motions. These components within the control hardware bracket would be used to drive two legs, and would be place in the front middle and back adding up to full control of all six legs.

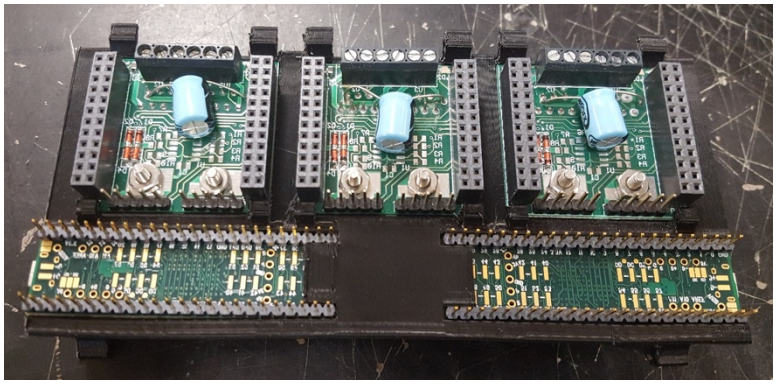


Figure 8. Teensy and Motor Driver Bracket

2.2.2 SENSORS

At the request of the client, the team selected distance and force sensors that were installed on the robot. The distance sensors were required to mount to a leg and predict when that leg will collide with some nearby object during normal operation. Time of flight sensors were chosen as they were the best fit for the hexapod robot (for justification refer to Spring 2018 report).

It also is necessary for the robot to have force sensors in its feet. This is so that the robot can detect when each foot is in contact with the ground. The client advised GMADA that the robot will only be able to flail its legs without them. Combining both sensors gives the robot expanded functionality and the ability to handle terrain more complicated than a flat and level plane.

A time of flight sensor is a range imaging camera system that resolves the distance of an object by emitting light, then measuring the phase shift difference in the returning reflection. This occurs on a near continuous basis. Another advantage of the phase shift difference phenomenon is that it is more difficult to blind the sensor with outside light in comparison to the infrared sensors examined previous.

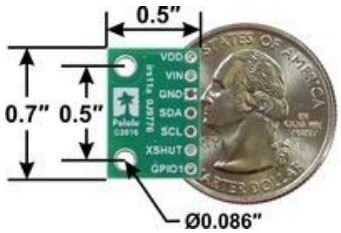


Figure 9. Size of VL6180X TOF Sensor

An example of a time of flight sensor is the VL6180X. Its physical dimensions are 12.7mm x 17.8mm x 2.159 mm with a weight of 0.5g (Pololu - VL6180X Time-of-Flight Distance Sensor). Fig. 10, Fig. 11, and Fig. 12 show these actual dimensions to scale, further indicating that the TOF sensor will fit comfortably on the hexapod robot's limbs. The sensor communicates using I2C. If the robot does not use I2C to communicate with sensors and motor drivers, then this sensor will require additional overhead to integrate into the controller architecture.

The VL6180X has a resolution of 1mm and a maximum range of 600 mm. This time of flight sensor operates under an absolute viewing range, which means, theoretically, that its minimum sensing range is 0 mm. Even so, the experimental minimum sensing range that can still maintain clear resolution is roughly 1cm (Pololu - VL6180X Time-of-Flight Distance Sensor). The device has an operating voltage of 2.7 V to 5.5 V, with a supply current of 5mA. These electrical requirements are like the other distance sensors examined. This means that no out of the ordinary considerations need to be made when connecting the time of flight sensor to the robot's electrical system.

During the implementation, a design for a mountable bracket that would house the time of flight sensors was needed. Using a software called space claim, GMADA designed 2 versions of brackets. The first was extremely bulky and required the sensor to be screwed in, it also had a cove for the sensor that was the source of a lot of the original designs bulk. The second iteration was redesigned to be held in place by pins built into the mount, and slimmed down the cover to reduce bulk and for the added function of further holding the sensor in place. These sensors were mounted to the side of the leg so that it would always have a clear view of the ground. The mounts themselves are adhered to the leg wit 3M double sided sticky tape, this is to ensure it stays in place during application, but isn't irremovable.

The addition of a force sensor to the hexapod will greatly expand the robot's controllability and future potential. The purpose of installing this sensor on the robot is to detect exactly when the foot has contacted the ground. This will give the robot a distinct input to help it transition between moving through the air and pushing against the ground. The team is planning to implement the Flexi-Force A201. The reason behind this choice is that the existing robot feet are designed around this sensor. The sensor is thin, and the sensing area is relatively small. This allows one to place the device within the feet of the robot. Fig. 13 depicts the size and shape of the sensor in question.

The Flexi-Force sensor is a two-wire device whose resistance varies as a function of applied force. This relationship between force and resistance can be converted into analog voltage by setting up a voltage divider. The only remaining step is for the computers installed in the robot to interpret the voltage in a useful and meaningful way. Furthermore, this sensor has a maximum force reading of up to 25 lbs (Tekscan).

Afterwards GMADA began to work on assembling the feet of the robot in a manner to accommodate the force sensor that would be held inside. First springs were placed between the moving part of the foot and its housing. The foot itself was allowed linear movement so that it could actuate the force sensor held within the housing, placing the spring between the two was to ensure that the sensor would be unloaded when the foot lifted of the ground again. Next, washers and printed parts were added and removed in a way to allow clearance for the force sensor to be adhered to the inside of the foot housing using double sided sticky tape. The washers were arranged in a way to give the sensor enough space so that it wouldn't be touched until a considerable load, the weight of the robot, was applied, so that there wouldn't be any false touchdown signals. After this the feet were reattached to the robot and data lines connected to a special circuit to help interpret the data. Next, GMADA began to design a mount that would hold the force sensor circuit board, the design was a flat plate with four hollow cylinders at each corner these cylinders would be were the circuit is screwed onto the mount in a way that allowed the electronics room to breathe. Once finished the boards were attached to the mounts and adhered to the inside of the robot with the 3M double sided sticky tape, to ensure safe operations while still being removable if needed. The circuits were then mounted inside the robot in three places close to each of the main hardware brackets.

The MA3that was used in the project is a miniature rotary absolute shaft encoder that reports the shaft position over 360° with no stops or gaps. As stated on the official page "The MA3 is available with an analog or a pulse width modulated (PWM) digital output. Analog output provides an analog voltage that is proportional to the absolute shaft position. Analog output is only available in 10-bit resolution. PWM output provides a pulse duty cycle that is proportional to the absolute shaft position. PWM output is available in 10-bit and 12-bit resolutions. While the accuracy is the same for both encoders, the 12-bit version provides higher resolution.

Three shaft torque versions are available: high torque (-D option), low torque (-N) and ball-bearing (-B). The high and low torque versions have a stainless-steel shaft and brass bushing lubricated with grease to provide ideal torque for panel mount, human-interface applications. The ball-bearing version has a brass shaft and miniature precision ball bearings suitable for high speed and ultra-low torque applications. The ball-bearing version is only available with a 1/8" shaft diameter.

Connecting to the MA3 is simple. The 3-pin high retention snap-in 1.25mm pitch polarized connector provides for +5V, output, and ground." These encoders were placed at each joint of the robot to measure its current position and to provide feedback and accuracy to the robot's motions while walking. These encoders needed to be calibrated so that GMADA may establish a reference frame that could be used while programing. Using this reference frame, it would be possible for the robot to know where it is located and were it needs to go depending on the coding.

While checking the calibration for the angle encoders, GMADA used a piece of string and a hanging level to check the straightness of the femur making sure that the axles of the two joints were straight in line with each other. This is because the straightness reference GMADA wanted this part of the leg was between the axles, which did not align well with the leg itself, which was slightly angled in this positon. Next for the tibia, GMADA used a more conventional level and placed it against the leg to check if it was directly vertical to the ground. Lastly, GMADA checked that the hip joint was coming directly out from the

body, this was done using a long stick and reference points on the body of the robot. The process was then repeated for each of the legs.

Figure 10. Optical Encoder

2.2.3 COMMUNICATION PROTOCOL

The robot's control is designed such that the main controller must send information to the sub-controllers that then command the motors that drive the motors to move the robot. This process must occur over some type of communication protocol. When selecting new controller hardware, a variety of communication protocols were also examined to determine which would best fit the project's needs. The client initially brought up noise as a potential concern, causing communication protocols that could tolerate signal noise to be initially preferred.

During the communication protocol selection process, the SPI, I^2C, UART, Ethernet using UDP packets, and CAN protocols were examined. As part of the selection criteria, the published communication speed, the addressing overhead, whether the protocol could send information in both directions simultaneously, and the estimated number of packets to send a full set of instructions were considered. For the protocol performance, a target command transmission rate of at least 100 commands each second while also receiving current position data was chosen.

Initial preferences toward protocols that had innate noise handling mechanisms dropped soon after communicating with other student groups. The built-in noise handling requirement was removed because of signal filtering techniques learned from the UTA Racecar team and wire routing techniques learned from the ROVER team. Because the ROVER team and Racecar teams run higher powered motors, the wire routing and signal filtering techniques were deemed sufficient for the signal noise the robot should encounter.

The fastest communication protocol was Ethernet, but it had significant addressing overhead and required additional hardware. However, the communication protocol that was selected was I^2C. I^2C is fast enough to reach the target command rate, uses a small number of communication wires, and sends addressed data packets. Sending addressed data packets simplified the process of transferring data between controllers by allowing GMADA to send complete sets of data in a single packet.

These advantages and easily handled limitations show that I^2C is the best fit for the hexapod robot.

2.2.4 WIRING DETAILS

The sub controllers are connected to each other directly via the I^2C protocol bus schema. These wires are red and brown. The brown wire is SCL (the clock for I^2C) and the red wire is SDA (the data for I^2C). The I^2C network uses the pullup resistors built into the

Raspberry Pi. The I^2C lines are crimped with JST SM pin connections to interface with the controllers.

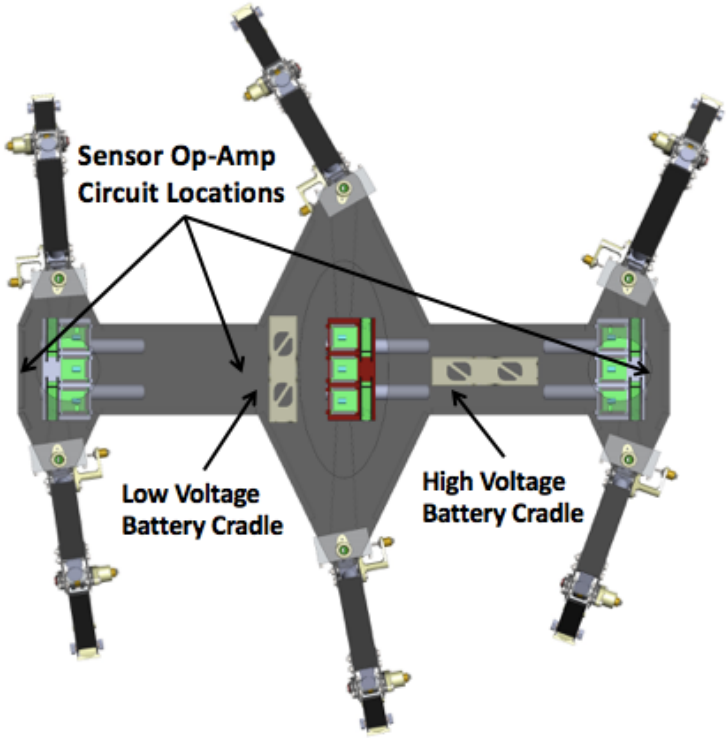


Figure 11. Layout of Batteries and Op-Amp Circuits

The force sensor and time of flight sensor both connect to the circuit board that linearizes the force sensor output. The force sensor circuit boards also have pins that are used to provide 3.3V and ground connections for time of flight sensor and the angle encoders. The force sensor circuits also provide the ground pins used to link the Teensy and motor driver grounds. There are three circuits each servicing two legs.

The force sensor circuit is connected to the Teensy using a three-wire ribbon cable. Two of the wires are power and analog ground while the third wire is the force sensor analog output. The force sensor is connected to the circuit board using a pair of wires. The force sensor wires can be connected to the circuit board in either orientation because the force sensor acts like a resistor.

The time of flight sensors use a 4-pin ribbon cable that goes between the sensor and the force sensor circuit board. The circuit board provides power and pullup resistance to the time of flight sensors. Two of the time of flight sensor wires connect to the power and ground rails. The SDA and SCL lines of the time of flight sensor are connected to the force sensor circuit board. Next the SDA and SCL lines are connected to the Teensy sub-controller with an additional pair of wires.

For the angle encoders, each one has two wires connected to the power rails on the force sensor circuit board. Additionally, a single wire that caries the sensor PWM output is connected to the Teensy. Each of the angle encoders must be connected to the correct pin on the Teensy.

The Teensy send instructions to the motor drivers using PWM signals. Wires that carry the PWM signals are connected between the Teensy and the motor drivers. Each Teensy is

connected to one motor driver with four wires to control the tibia and femur. Each teensy is also connected to a second motor driver with two wires to control the hip rotation. This second motor driver is connected to two Teensy controllers with two wires each.

The pinout diagrams for the sub controllers, motor drivers, main controller, and sensor circuit are available in their respective appendix sections.

2.3 POWER SYSTEM DETAILS

The communication and controller power cabling was completed in Summer 2018. The motor drivers are powered by one battery and the control hardware is powered by another. This is required as these components have different voltage and amperage requirements.

2.3.1 BATTERIES AND REGULATORS

The power system consists of two batteries, a voltage regulator, a kill switch, a fuse and fuse holder, wiring, a ground screw in terminal, and a positive voltage screw in terminal. The motor power grid runs on 14 gauge stranded wire. The comms power grid runs on 18 gauge and 22 gauge stranded wire.

The team chose the 11.1V Zippy Flightmax 5800mAh 3 cell lithium polymer battery (refer to Spring 2018 for discussion of alternatives). The selected battery exceeds the minimum required capacity and is capable of a 30C discharge rate which is significantly higher than the minimum required value. The full specifications of the battery can be found in the Spring 2018 report.

The kill switch is a Dorman 35A toggle switch. This switch is rated for the approximate maximum voltage that all the motors can draw if they are all stalled simultaneously. The switch is a three-position switch that is connected between the positive terminal of the battery and the positive high voltage rail. The fuse holder that was added to the high voltage power system was a Cooper Bussman fuse holder that accepts standard ATM blade fuses. The fuse holder currently holds a 30 Amp ATM blade fuse. Additional details can be found in Appendix 6.3.4

The voltage regulator is a 5V 3A voltage regulator that was selected last semester. Additional details can be found in the product page for the voltage regulator.

2.3.2 POWER WIRING DETAILS

The motor power grid runs on 14 gauge stranded wire at 12V while the coms grid runs on 18 and 22 gauge stranded wire at 5V. The motor power grid connects the nine motor drivers to the XT60 connector in parallel. The positive battery terminal is currently connected to a physical switch and a 30A fuse before being attached to a screw in terminal that is used to distribute power. The ground for the high voltage power system also uses a second screw in terminal for its connections. The ends of the positive voltage and ground wires are stripped and connect directly into the motor drivers. Everything on the power grid is on a shared ground. Additionally, each of the motor drivers connects to the ground rail of the force sensor circuits using the ground wire pin (not the main ground screw in terminal) on the motor driver.

The sub-controllers receive 5V-3A power from one battery via the voltage regulator. The sub controllers are connected in parallel via 18 gauge red and black wire. The power lines for the different Teensy are connected using crimped butt connectors. For the butt connectors, multiple wires were inserted into a single side after they were soldered together

to allow multiple wires to branch off of the main power wires. These lines are crimped with 7 pin-header connection pairs (for the controllers) and one XT60 connector end (for the battery). Note: the voltage regulator is mounted to the power lines before the XT60 so either battery can be placed in any battery cradle. The wiring diagram can be found in the Appendix Section 6.3.6.

2.4 SOFTWARE DETAILS

The control system consists of a single Raspberry Pi communicating via I^2C with six teensy sub controllers and all the subsequent sensors and components. Thus, there are two pieces of software that must be discussed: the sub controller program and the main controller program. This section seeks to provide familiarity with these computer programs and their responsibilities. In addition to these, there is also an introduction to the remote-control program that runs on one of the Raspberry Pi cores. These pieces of software were created by GMADA for this system to enable remotely controlled movement of the hexapod.

2.4.1 SUB CONTROLLER PROGRAM

The Teensy sub-controllers are responsible for a variety of different tasks. These tasks include receiving the desired leg position from the Raspberry Pi, moving the legs to the desired position, collecting sensor data, and returning the sensor data to the Raspberry Pi. To accomplish these tasks, a generalized program was developed that could be used on all six sub-controllers with minimal modifications. To accomplish this, the angle encoder calibration data was tied to a sub-controller number that could be modified at the top of the program. Additionally, the I^2C address of each sub-controller is configurable at the top of the program. This allows multiple sub-controllers to be programmed while only requiring that two values are changed.

The sub-controller is programmed to send and receive I^2C messages from the main controller. The I^2C bus is configured to run at 100 kHz because it became unstable when GMADA increased the speed to 200 kHz. The sub-controller receives the x, y, and z coordinates from the main controller as a series of 12 bytes. These bytes are then converted to floats using structures and unions. The sub-controllers are configured to send the current leg angles, proximity sensor data, and force sensor data to the main controller as a series of 20 Bytes. The data is converted from its original float format into Bytes by using structures and unions. The sub-controllers send and receive data as bytes to ensure that the data is being transmitted quickly and efficiently. The unions in the program allow the memory locations that store either the outgoing or incoming data as either floats or the Bytes that make up the floats. Because the Structures and Unions are used in the I^2C interrupts, all variables defined in the Structure are defined as volatile. Additionally, the variables defined in the ReceiveFields Union are also set to volatile. This was done to improve the stability of the Teensy program on the I^2C bus. Before these variables were defined as volatile, the Teensy would stop communicating on the I^2C bus after a few minutes.

The desired x, y, z coordinates that the sub-controllers receive from the main controller are measured in meters. These x, y, z coordinates are converted into desired angles using a slightly modified form of the inverse kinematics equations that were recorded by a previous group. The coordinates that the current sub-controller program use vary from the

definitions that previous groups used. For the current sub-controller programs, the x coordinate is perpendicular to the robot's body and the y coordinate is parallel to the robot's body. Additionally, the z coordinate of the foot is typically negative according to the kinematics formulas that were derived for the robot. However, the sub-controller program uses a positive z coordinate to indicate the distance the foot is below the hip joint. This was done to accommodate the inverse kinematics equations that were obtained by other groups in previous semesters.

To move the leg to the desired position, the sub-controllers run P controllers to control the voltage applied to each motor. The P controllers compare the current leg angles to the desired angles. The error between the current and desired angles is multiplied by a gain to obtain an output voltage. As a note, the output voltage to the femur is increased by a static amount whenever the voltage is positive. This was done to compensate for gravity. Finally, the output voltage is converted into a PWM signal and sent to the motor drivers.

The sub-controllers are responsible for checking the proximity sensor data which communicates using I^2C. To minimize the amount of data transmitted and to avoid extending the main I^2C bus to the feet of the robot, the proximity sensors use a second I^2C bus. To use a second I^2C bus, the i2c_t3.h library was imported. This library is available through the Arduino IDE library manager. However, the library for communicating with the proximity sensor that Pololu provided uses the original Wire library. To use the second I^2C bus, the VL6180X library by Pololu was modified to use the i2c_t3.h library's I^2C communication functions. The original library can be imported using the Arduino IDE Library Manager. The modified library can be found on the Hexapod Team google drive or in the files turned in at the end of the semester in the Summer 2018 → Controller programs →Modified Arduino Libraries folder.

There is one issue with the Time-of-Flight proximity sensor in the sub-controller program. The proximity sensor was configured with a shorter time-out period to increase the speed of the sub-controller program. However, if the proximity sensor does not see the ground (within approximately 20cm) for an extended period of time, the sub-controller program will lock up and will continuously send the same voltage to the leg motors. To avoid this, some later versions of the program have commented out the line of code that reads the proximity sensor because it was not used in the main controller program. However, this behavior should be investigated in the future to determine the cause of the issue and correct it. Other methods of reading the sensor data with the existing or alternative libraries may better suit the needs of future groups.

For the angle encoders, they are supposed to send a PWM value that the Teensy translates to a number between 0 and 4095 (or 4096). However, slightly higher values have been observed. The error caused by this issue was roughly estimated to be a few percent of the reading. This error in the angle measurements that this caused was deemed small enough to be ignored until more precise control is needed. Future groups may want to investigate the behavior to minimize any error that the unexpectedly high encoder values may cause.

Finally, it is possible for the Teensy to stop communicating on the current main I^2C bus. This can occur after the Teensy has been running for a while. However, if all the Teensy sub-controllers are disconnected from power and reconnected, the Teensy have run for more than 10 or 15 minutes before dropping off of the I^2C bus. Nevertheless, future groups may want to investigate and correct this behavior.

For additional resources on the making of the sub-controller program, please see the official Aduino reference materials and the links below:

http://www.benripley.com/diy/arduino/three-ways-to-read-a-pwm-signal-with-arduino/
Structures and Unions: http://forum.arduino.cc/index.php?topic=158911.0
https://www.arduino.cc/reference/en/language/variables/variable-scope--qualifiers/volatile/
https://www.avrfreaks.net/forum/volatile-struct-how-it-works

2.4.2 MAIN CONTROLLER PROGRAM

In the project, the Raspberry Pi 3B+ is chosen to become main controller of the Hexapod Robot. The first task of main controller is it must communicate with 2 different type of devices. They are PlayStation 4 (PS4) controller, and Teensy for sub-controllers. The Bluetooth connection is used to connect the Pi and PS4 controller. The second communication type that is used in this project is I^2C communication for connecting the Pi and 6 sub-controllers. Besides that, the second task of main controller is it needs to execute the gait algorithm of the hexapod and send these data to sub-controllers. In this project, there are several libraries are used. They are smbus, evdev, numpy, math, numpy.linalg, struct, time, and threading library.

To execute these two main tasks efficiently, the parallel programming is being used to ensure that there will be no delay between these tasks. The parallel program will run on three separated threads since establishing these threads will help the program can run multiple functions at the same time. In the program, the first thread will run 'RC_input' function and this function will only take input from PS4 controller. The second thread will run the 'main' function. The 'main' function will run multiple different sub-functions that are used to predict the next move of the robot's legs. Lastly, the third thread will only execute I^2C communication, and the function on this thread is 'sending_and_receive_info' function. In addition, there are two functions that are used in the main program are 'bounding_box' and 'determine_Y_f_and_Y_b' function. These two functions will only run one time when the program is executed since they only calculate the limited range that leg allow to move and y-position that the leg need to have to move forward or backward.

In the first thread, the 'RC_input' function will try to get input PS4 controller and store it in the array, which is called btn_array_cur. This array always updates and contains the most recent input from PS4 controller. In the current program, the triangle button is for moving forward, and x button is for moving backward. Moreover, before executing function, the user need to know what event the PS4 controller connect to. This event information can obtain by simply using this command line.

```
$ python /usr/local/lib/python2.7/dist-packages/evdev/evtest.py
```

In the second thread, the 'main' function will run 'loop_time_offset', 'desired_position', 'angle_to_coordinate', and record data from the sub-controller. Firstly, 'loop_time_offset' function allow the Pi knows what time the button is pressed and released, which are t_0 and phase_init variables in the program. These variables will help the Pi know where the legs are currently at. Besides that, this function will also reset the phase_init back to zero after 18 seconds. Secondly, 'desired_position' function is about

calculating where the legs need to move over the time. The 'desired_position' function has two argument, one is the index of legs and it is called as phase. The second argument is the current time of the leg. The function will these two argument to calculate the leg's movement, which will have cycloid path in Y-Z plane, and the X- direction will be assume to be constant. Lastly, 'angle_to_coordinate' function is used to convert feedback angle from sub-controllers into Cartesian coordinate system and this will be used record as 'recording_measured_data.txt' file.

The third thread is 'sending_and_receive_info' function, which mainly run the I^2C communication function. Besides that, there are also some small tasks are executed in this thread. They are recording theorical data and changing sign of y component of left side of the hexapod before sending into teensy. The changing sign is needed because the Teensy was programmed and tested on the right side originally. The 'sending_and_receive_info' thread uses 'I2C_com' function to send and receive data from sub-controllers. In the 'I2C_com' function has two different functions, which are 'writeNumber' and 'get_data' function. The 'writeNumber' take variables 'address' and 'byte_to_send' as arguments. The variable 'address' is the leg that the Pi want to communicate with. The 'byte_to_send' variable will convert x,y,z values into 12 bytes and store it before passing into 'writeNumber' function. In addition, in 'writeNumber' function, there is a bug need to be fixed in the future. The print(address) line is not allowed to comment out, it has to be executed so that the communication can activate without any errors or issues. Besides that, there is also another bug that need to investigate in the future is sometimes when the forward button changes to backward button, the robot suddenly reset to initial position, which should not happen. Besides that, the I^2C communication speed that is current used in the Pi is 100000Hz, and this is also the stable speed for the communication without and errors or issues.

Lastly, when the python script is executed, it will run threads p1, p2, and p3. However, after executing the script, it will not be able to be terminated since these threads are not currently in daemon mode. The daemon mode needs to be setup before threads start. By implementing daemon mode into threads, the entire python script can be killed by simply using Ctrl+C. This is a software safety's switch to stop sending data to robot.

To use the robot, the user needs to follow this checklist:

1. Connecting PS4 to Pi
2. Checking the I^2C communication connection to ensure that it can communicate with the sub-controllers by command:
   ```
   $ i2cdetect -y 1
   ```
3. Checking the speed of the I^2C communication is at 100000Hz by command:
   ```
   $ sudo nano /boot/config.txt
   ```
   Find and change 'i2c_arm_baudrate' into 100000. Reboot if it is not 100000Hz
4. Turn on the switch on the Robot to ensure that the robot is at its default position.
5. Running the python script by simply pressing F5.
6. To command the robot moves forward, press triangle button. For moving backward, pressing X button.

Additional resources on making the main program:

https://core-electronics.com.au/tutorials/using-usb-and-bluetooth-controllers-with-python.html
https://stackoverflow.com/questions/11815947/cannot-kill-python-script-with-ctrl-c
https://docs.python.org/3/library/threading.html
https://learn.sparkfun.com/tutorials/raspberry-pi-spi-and-i2c-tutorial

2.4.3 REMOTE CONTROL PROGRAM

One of the first things GMADA worked on this semester was a simple program that would be used to control the robot via a Bluetooth controller. Using a myriad of online resources, the most useful being core-electronics.com, GMADA could create a program that could read and interpret input from the controller. This was done by determining the data each button press gave, and corresponding it to an appropriate reaction. For the sake of testing the reaction was to print out a pre-specified letter upon button press. Next GMADA began working on multi-core threading so that the controller program would run separately main program and could pass information between them. Details on the setup and processes of establishing basic and complex remote commands like the ones used can be found using the resources listed within the appendix.

3. FUTURE DEVELOP DETAILS

While great progress has been achieved, the system is far from finished. This section aims to expedite future development by providing an overview of digital resources, works of caution, and relaying GMADA's thoughts on the project. An overview of the disc files available from Dr. Huff and of the Google drive account used by this project team are available in Section 3.1. General notes from GMADA's time with the system are presented in Section 3.2. The system faces mechanical, electrical and software challenges that must be resolved by future teams. These hurdles (as they are currently understood) are presented in Section 3.3.

3.1 OVERVIEW OF DIGITAL RESOURCES

At the time of writing, this project has been developed by three different teams (in 2011, 2012, and 2018). The available resources are:

- Data from 2011 Team
- Data from 2012 Team
- Data from 2018 Team
- 2018 Team Google Drive Account

Note: the Google Drive account contains all the previous team information in a reorganized format. Thus, it is advised that you primarily use this resource for quickly finding previous work.

Important resources:

- Solid model of the system (Solidworks 2017)
- Inverse kinematics (2012 Team, validated by 2018 team)
- Wiring and pinout diagrams for all electronics
- Previous teams' presentations and reports
- Software for sub controller and main controller

- Research material and help manuals

Access Details for Google Drive Account

- Username: hexapod.uta@gmail.com
- Password: Seniordesign

Overview of Google Drive Resources:

The '2011 Team (Mavapod)' and '2012 Team (Four Guys Robotics)' folders contain the reorganized and culled files from both senior design teams. The 'Solid Models' folder contains the most recent revision of the solid models. The '2018 Team (GMADA)' folder contains all GMADA's files and the majority of the important resources listed above. While the previous teams' data is provided, it is mostly outdated.

3.2 TECHNICAL DEVELOPMENT

This section provides a small amount of detail regarding several key tasks that GMADA considers important for the continuing development. This list also serves as a good list of current issues with the overall system.

3.2.1 MECHANICAL DEVELOPMENT

Redesign legs to allow for unpowered standing

As of August 2018, the robot could not maintain a standing position without power. This means that while the motors are unpowered they slowly lower the robot. To accomplish this task during a standard capstone course requires great familiarity with the current design and an outstanding solution for the new legs. GMADA possessed neither of these things at the onset of the project and thus did not pursue complete replacement. Efforts to reduce slop movement were largely successful and are detailed in the Spring 2018 report.

Update top fasteners to ease disassembly

The top of the carbon fiber housing (the lid of the robot) is secured to the edges with too many fasteners. This overall design is ugly and has asymmetrical hole placement that looks tacky. However, these fasteners are required for the structural integrity of the system. Thus, a carbon fiber endoskeleton might be feasible where the carbon plates can be removed without affecting the structural integrity of the hexapod.

Replace fasteners in leg A to increase mobility

Leg A possesses fasteners on J2 and J3 joints (refer to mechanical details on joint names) that impede the full range of motion of the joints. These must be replaced with less obtrusive fasteners.

Redesign brackets to hold new motor drivers

The elegant bracket solution that holds the sub controllers and motor drivers must be updated/redesigned. The motor drivers were salvage components. But the wiring disaster in summer of 2018 (refer to incident reports in the appendix) damaged enough of them that

new motor drivers will have to be procured. This will required adapting the current bracket. The solid models are available on the Google Drive.

Assess issue with tight joint

While running the walking gait program, GMADA noticed that leg F was not able to rotate the femur up at the same speed as the other legs and struggled to lift the leg to the correct position. Because the gain values are the same on every Teensy, this shouldn't be a programming issue. This issue should be investigated by future groups to determine the root cause. The most likely causes of this behavior include: excessive friction in the joint, a malfunctioning or damaged motor driver, and a damaged motor. These possibilities should be investigated so that leg F can function at the same level as the other legs.

3.2.2 SUB CONTROLLER SOFTWARE DEVELOPMENT

Fix TOF lockup error

The Time of Flight sensor currently has an issue when it is unable to see the ground. The sensor can lock up the entire Teensy program when the ground is past the range that the sensor can read with the current settings. This behavior should be investigated to determine what is causing the issue. Alternatively, different methods of reading the Time of Flight sensor can be investigated.

Stabilize program to prevent locking i2c bus

The Teensy sub-controllers currently have an issue that causes them to no longer communicate on the I^2C bus. This issue was previously caused by not using the volatile keyword in the structures. Currently, this issue typically occurs after the program has been running for 10 to 15 minutes or even longer.

3.2.3 MAIN CONTROLLER SOFTWARE DEVELOPMENT

Make additional/more complex gaits

The robot is capable of complex gaits involving up to 3 legs in the air at a time (refer to Spring 2018 report, Section 2.3.2). Currently, only the simplest of motions have been programmed (one legged gait, forward and backwards motion). Once other issues have been addressed, more robust walking gaits need to be implemented.

Make program faster (fix i2c bus)

The main I^2C bus is currently running at 100 kHz. This is slower than GMADA originally intended. GMADA attempted to increase the speed of the I^2C bus to 200 kHz, but the I^2C communication failed within a couple of minutes. This issue should be investigated and corrected. The I^2C bus may need to be monitored with an oscilloscope to ensure that the signals are being transmitted correctly, are reaching the required voltage, and are not being masked by noise. To correct the issue, it may be necessary to add filters to the main I^2C bus to reduce noise. Alternatively, the pull-up resistance may need to be changed.

Troubleshoot RC program restart bug

The remote control program on the Raspberry Pi is having an intermittent issue where the walking gait will suddenly return to its original position. This issue occurs when a button has been released and pressed again, or when the button for the opposite direction has been pressed. This issue may be caused by a slight logic error in the function that controls the walking phase timing.

Refine RC control to include more than just 2 commands

The current remote control program reads four buttons. However, the main program is not currently programmed to respond to two of the buttons being pressed. The remaining two buttons that are currently read by the RC program should be used to enable additional capabilities on the robot.

Troubleshoot print error in data transmission function (right number function)

The writeNumber() function has an issue where the entire Raspberry Pi gait program slows down when the print statement is commented out of the function. This is not the expected behavior. This should be investigated so that the overall program can have less unnecessary output.

3.2.4 ELECTRICAL DEVELOPMENT

Replace motors drivers to allow for full voltage to motors

The motor drivers that were originally used on the robot were not capable of driving the motors at the full 20V that the motors were capable of. Additionally, multiple motor drivers were damaged during the semester. For these reasons, the motor drivers on the robot should be replaced with models that are capable of handling higher voltages to allow the motors to function at their full potential.

Replace motors to allow the robot to stand in a wider range of positions

The current motors on the robot can keep the robot standing when the robot is in a highly ideal position. However, the robot is not able to stand on its own in the configurations used in the walking gait. For this reason, the geared motors on the robot may need to be replaced with stronger motors and gears. This would allow the robot to hold up more weight and potentially walk on its own.

Replace local kill switch with remote kill switch

The remote kill switch that was originally used on the robot was damaged when the high voltage system was originally tested. The remote kill switch was replaced by a mechanical switch to allow the voltage to the motors to be shut off without disconnecting the battery. This mechanical switch should be replaced by electronics that allow the robot to be shut off remotely. Shutting of the robot remotely is safer in case of a major malfunction.

Refine cable management

The current cable management inside the robot is less than ideal. The cables currently have some strain relief, but are somewhat messy due to the large number of wires. Future

groups should work to refine the cable management so that the cables are routed cleanly and without risk of disconnecting.

Mount RPi and low-voltage battery inside frame

The current robot frame design proved to be less than ideal for accessing batteries and control hardware. For these reasons, the Raspberry Pi and the low voltage batter cradle were moved to the top of the robot. Future groups should examine ways of making the control hardware and batteries more accessible so that these components can be moved inside the robot case.

The Raspberry Pi can be configured for remote access by using VNC. However, the Raspberry Pi has to be able to connect to a network when it turns on for this to be a practical solution. Future groups should work with OIT to enable the Raspberry Pi to connect to the network on startup to allow the Raspberry Pi to be accessed remotely and moved inside the robot frame.

Fuse on low voltage power grid

The low power system is currently designed without a fuse. In the future, it may be a good idea to add a fuse to the low power system to ensure that a short circuit in the wiring damages as few electronic components as possible.

Troubleshoot RPi power issue

The Raspberry Pi currently is unable to start when the 5V power wires are connected to the 5V pins on the Raspberry Pi. This may be caused by the voltage regulator being damaged when the high voltage power system shorted, or it could be caused by the voltage regulator being unable to provide enough current. The Raspberry Pi power issue should be investigated so that the Raspberry Pi can be powered by the low voltage power system on the robot.

4. SUMMARY AND CONCLUSIONS

UTA began the spring 2018 semester with a decaying hexapod robot originally intended to demonstrate the level of engineering practice and principles it taught. GMADA was tasked with refurbishing this piece of machinery so that it may finally walk. This engineering feat was split up into two semester's worth of work. The ultimate task that the team was supposed to complete was the complete functionality of the hexapod robot.

The first challenge was to completely tighten the leg joints by repairing the errors introduced by sloppy construction. The group was almost entirely successful by achieving a five-fold reduction in joint looseness. Next, the group researched and selected a variety of hardware to improve the robot with. These items include batteries, distance and force sensors, and control hardware. Every selection was based on solid research, leading to, if not optimal, then satisfactory hardware components to be installed on the robot at some future date. Behind this physical work, the group allotted resources to start designing the robot's gait, to some success. By far the most important items to test were the main and sub controllers. It was only after choosing the communication protocol that the brains of the robot began to take shape through the development of the software architecture. The technical members of the team were able to construct commands in Python and C to teach the main controllers and sub controllers how to speak to each other. This

culminated in the final breakthrough enabling full, two-way communication between the two controller types. Each one of the previous successes contributes to the project in its own way.

During the summer semester of 2018 GMADA ban working on realizing the ideal of having the robot walk. First GMADA began by creating mounts that would house the sensors and control hardware within the robot. Once the mounts were designed and printed GMADA began placing the pieces within the robot, and started deciding on the placement of the hardware and wires. Once GMADA decided on an internal architecture GMADA placed all the hardware in the mounts in their desired place. GMADA then began working on creating custom wires and connectors for the power and data connections. After all of the wire had been sized and fitted with connectors GMADA began the long process of wiring everything together. After the robot was all wired up GMADA directed its attention to designing the program to get the robot walking. It was around this time GMADA began some initial testing of the robot, unfortunately parts of the robot had been wired up incorrectly earlier. This lead to the frying of a lot of the robots sensitive hardware controllers, within the final week of the project.

Thus even the gait program for walking is complete, so to speak, GMADA were unable to test it and modify it to full functionality, due to a lack of parts.

5. REFERENCES

UBEC DC/DC Step-Down (Buck) Converter - 5V @ 3A output. Retrieved from https://www.adafruit.com/product/1385

Raspberry Pi FAQs - Frequently Asked Questions. (n.d.). Retrieved May 4, 2018, from https://www.raspberrypi.org/help/faqs/#topPower

LV-MaxSonar® -EZ™ Series. (2015). Retrieved from https://www.maxbotix.com/documents/LV-MaxSonar-EZ_Datasheet.pdf

Pololu - VL6180X Time-of-Flight Distance Sensor Carrier with Voltage Regulator, 60cm max. (2018). Retrieved from https://www.pololu.com/product/2489/resources

VL6180X - Proximity sensor, gesture and ambient light sensing (ALS) module - STMicroelectronics. Retrieved from http://www.st.com/en/imaging-and-photonics-solutions/vl6180x.html

UBEC DC/DC Step-Down (Buck) Converter - 5V @ 3A output. Retrieved from https://www.adafruit.com/product/1385

ZIPPY Flightmax 5800mAh 3S1P 30C. (n.d.). Retrieved May 4, 2018, from https://hobbyking.com/en_us/zippy-flightmax-5800mah-3s1p-30c.html?___store=en_us

Citations Created using Citethisforme.com and citationmachine.net

6. APPENDIX

6.1 MAIN CONTROLLER SOFTWARE

```
#! /usr/bin/python
import smbus
import time
import struct
from evdev import InputDevice, categorize, ecodes
#from multiprocessing import Process, Queue
from threading import Thread
from numpy import *
from numpy.linalg import inv
from math import *
bus = smbus.SMBus(1)
ADDRESS = [0xa, 0xb, 0xc, 0xd, 0xe , 0xf]
#define the index values in ADDRESS for the left legs
left_side = [0, 2, 4]
############Initial values ##############
########################################
btn_array_cur = [0,0,0,0]
btn_array_pre = [0,0,0,0]
#initialize contanst value
tau = 3.0
beta = 0.05*tau
h = 0.20 #meter
Z_0 = 0.34 #meter
Y_f = 0.3
Y_b = -0.3
X_0 = 0.32
phase_init = 0
t_0 = time.time()
time_t = time.time() - t_0 + phase_init

delta_min = 5*pi/180
delta_max = 25*pi/180
theorical_record  = open('recording_theorical_data.txt','w')
measured_record  = open('recording_measured_data.txt','w')
L_femur = 0.27978
L_tibia = 0.36830
C_1 = zeros((3,1), float)
L_1 = zeros((3,1), float)
num_of_legs = 6 #number of leg that GMADA want to control
leg_info = zeros((3,num_of_legs),float) # define feedback information matrix
cmd_position = zeros((num_of_legs,3),float)
data = [0,0,0]
# The user should always check the path that the PS4 are connected to.
# This step can by done by the command line below.
#python /usr/local/lib/python2.7/dist-packages/evdev/evtest.py
def RC_input():
       #python /usr/local/lib/python2.7/dist-packages/evdev/evtest.py
       #run the command above to check which event or channel the RC controller
       # will be used.
```

```
        gamepad = InputDevice('/dev/input/event4')
        #button code variables (change to suit your device)
        aBtn = 304
        bBtn = 305
        xBtn = 307
        yBtn = 308
        left = 312
        right = 313

        start = 315
        select = 314
        lTrig = 310
        rTrig = 311
        x = 0
        y = 0
        a = 0
        b = 0
        #prints out device info at start
        print(gamepad)
        #loop and filter by event code and print the mapped label
        for event in gamepad.read_loop():
            if event.type == ecodes.EV_KEY:
                if event.code == yBtn:
                    if event.value == 1:
                        y=1
                    elif event.value == 0:
                                y=0
            if event.type == ecodes.EV_KEY:
                if event.code == aBtn:
                    if event.value == 1:
                        a=1
                    elif event.value == 0:
                                a=0
               ############
            if event.type == ecodes.EV_KEY:
                if event.code == xBtn:
                    if event.value == 1:
                        x=1
                    elif event.value == 0:
                                x=0
                ##########
            if event.type == ecodes.EV_KEY:
                if event.code == bBtn:
                    if event.value == 1:
                        b=1
                    elif event.value == 0:
                                b=0
              global btn_array_cur
              btn_array_cur = [a,b,x,y]
#do not comment out the print(address) since it will make
#the program run slowly
#need to investigate it more
def writeNumber(address,x):
        print(address)
        bus.write_i2c_block_data(address, 0x01 , x)
```

```
	return -1
def get_data(address):
	#read data from sub-controller and convert it into floats
	#temp holds the 12 bits that bus.read_i2c_block_data() outputs
	#the third argument of bus.read_i2c_block_data(address, 0,12) sets
	#the number of bytes to read from the i2c interface
	temp=bus.read_i2c_block_data(address, 0,12);
	#print(temp)
	#temp2 changes the data in locations 0 to 11 into a character string
	temp2=''.join([chr(i) for i in temp[0:12]])
	#temp3 unpacks the character string into 3 float numbers ("<3f")
	temp3=struct.unpack("<3f",temp2)
	return temp3;
# convert input value => string => byte to send to sub-controller
def str_to_byte(value_to_byte):
	str_converter = str(value_to_byte)
	converted = []
	for b in str_converter:
		converted.append(ord(b))
	return converted
#############################################################
###### SENDING and RECEIVING VALUES TO THE SUB-CONTROL#######
#############################################################
def i2c_com(address, index ,input_x,input_y,input_z):
	global leg_info
	global data
	#rewrites the data that must be sent into the correct format using struct.
	#converts input values into floats using float()
	#packs floats into memory data using struct.pack()
	#unpacks memory data into integers using struct.unpack()
	#converts the tuple output from struct.unpack into a list using list()
	#currently set to use 3 floats ("!3f") which becomes 12 bytes ("!12b")
	byte_to_send =
list(struct.unpack("12b",struct.pack("<3f",float(input_x),float(input_y),float(input_
z))))
	# send to sub_controller.
	writeNumber(address , byte_to_send)
	#############################################################
	###### RECEIVING VALUES TO THE SUB-CONTROL and PRINT #######
	#############################################################
	## each column of the matrix will represent information of the leg
	## in order [gamma,phi,theta, force , proximity] respectively

	#print("Data from subcontroller: ")
	data = get_data(address)
	#print('data: ')
	#print(data)
	leg_info[:,index] = [ data[0] , data[1] , data[2] ]
#the desired position will calculate the move of each leg base on
# current time time_t. In this program, the desire position will have
#cyloid path in y-z plane and the x-position will be constant since
# GMADA assume that the leg is going to move along y-direction
def desired_position(phase, time_t):
	## X_0 , Y_f, Y_b , Y_0, tau, beta are all constant
	t = time_t + phase*tau
```

```
	global x,y,z
	global cmd_position
	######## Z - direction ###################################
	if t >= (-6*tau) and t <= (-5*tau + beta):
		nu = -6*tau - beta
		z = Z_0 - h/2 *(1 - cos((t-nu)*2*pi/(tau + 2*beta)))
	elif t >= (-5*tau + beta) and t <=  -beta:
		z = Z_0
	elif t >= (-beta) and t <= (tau + beta):
		nu = -beta
		z = Z_0 - h/2 *(1 - cos((t-nu)*2*pi/(tau + 2*beta)))
	elif t >= (tau+beta) and t <= 6*tau - beta:
		z =Z_0
	elif t >= (6*tau-beta) and t <= (7*tau + beta):
		nu = 6*tau - beta
		z = Z_0 - h/2 *(1 - cos((t-nu)*2*pi/(tau + 2*beta)))

	elif t >= (7*tau + beta) and t <= (12*tau - beta):
		z = Z_0
	elif t >= (12*tau - beta) and t <= (12*tau) :
		nu = 12*tau - beta
		z = Z_0 - h/2 *(1 - cos((t-nu)*2*pi/(tau + 2*beta)))
	############If statements Y-direction ########################
	#############################################################
	if t >= (-6*tau) and t <= (-5*tau):
		mu = -6*tau
		y = (Y_f - Y_b) / tau *(t - mu) + Y_b
	elif t >= (-5*tau ) and t <=  0:
		mu = -5*tau
		y = (Y_b - Y_f) / (5*tau)*(t- mu) + Y_f
	elif t >= 0 and t <= tau :
		mu = 0
		y = (Y_f - Y_b) / tau *(t - mu) + Y_b
	elif t >= (tau) and t <= 6*tau :
		mu = tau
		y = (Y_b - Y_f) / (5*tau)*(t- mu) + Y_f
	elif t >= (6*tau) and t <= (7*tau ):
		mu = 6*tau
		y = (Y_f - Y_b) / tau *(t - mu) + Y_b
	elif t >= (7*tau) and t <= (12*tau ):
		mu = 7*tau
		y = (Y_b - Y_f) / (5*tau)*(t- mu) + Y_f
	##################---X---################################
	######## X - direction is gonna be the same ###############
	x = X_0

	cmd_position[phase,:] = [x,y,z]
	#time.sleep(0.5)
	#print('Desired Position function: ')
	#print(cmd_position)
# This function convert the feedback of the teensy from angle into
# Cartisan coordinate system and return it as a P vector
def angle_to_coordinate(gamma, phi, theta):
	R_H_A = [[cos(phi) , -sin(phi), 0],[sin(phi) , cos(phi), 0],[0 , 0, 1]]
	R_H_A = reshape(R_H_A,(3,3))
```

```
	#print(R_H_A)
	R_L_H = [[cos(theta) ,0 , -sin(theta) ],[ 0 , 1 , 0],[sin(theta) , 0 ,
cos(theta) ]]
	R_L_H = reshape(R_L_H,(3,3))
	#print(R_L_H)
	R_B_L = [[-1, 0 , 0],[0 , 1 , 0],[0 , 0, -1]]
	R_B_L = reshape(R_B_L,(3,3))
	#print(R_B_L)
	R_C_B = [[cos(gamma), 0 , -sin(gamma) ],[0 , 1 , 0],[sin(gamma) , 0 ,
cos(gamma)]]
	R_C_B = reshape(R_C_B,(3,3))
	#print(R_C_B)
	L_1 = [[1,0,0]]
	L_1 = reshape(L_1,(3,1))
	C_1 = [1,0,0]
	C_1 = reshape(C_1,(3,1))
	#print('vector P: ')
	P = L_femur*dot(dot(R_H_A , R_L_H),L_1) + dot(L_tibia * dot(dot(R_H_A , R_L_H)
, R_B_L), dot(R_C_B , C_1))
	return P

# this function calculate the area that the leg allow to move.
# And this is only run once
def bounding_box(Z):
	# using Z_0 as Z
	# Z is variable, the height of the robot respect to the ground
	theta_min = asin(L_tibia / L_femur * cos(delta_min) - Z / L_femur )
	theta_max = asin(L_tibia / L_femur * cos(delta_max) - Z / L_femur )
	L1 = L_femur*cos(theta_max) + L_tibia*sin(delta_min)
	L2 = L_femur*cos(theta_min) + L_tibia*sin(delta_max)
	bound = [L1,L2]
	#print(bound)
	return bound
#This function is used to calculate the maximum Y position that the leg
# can move forward and backward.
# And this is only run once
def determine_Y_f_and_Y_b(Z) :
	# Z_0 = 0.34
	L_1 = (bounding_box(Z_0))[0]
	L_2 = (bounding_box(Z_0))[1]
	phi_max = acos(L_1/L_2)
	phi_min = phi_max*(-1) #same as phi_max but symmetric
	#print('phi_max: ' + str(phi_max*180/pi) + '   phi_min: ' +
str(phi_min*180/pi))
	# Y_f : Y_forward
	# Y_b : Y_backward
	Y_f = L_2*sin(phi_max)
	Y_b = L_2*sin(phi_min)
	Y_position = [Y_f , Y_b]
	#print( 'Y_f and Y_b')
	#print(Y_position)
	return Y_position
# this function is used to send information to 6 teensy and it also used
# to record calculated x,y,z position
def sending_and_receive_info():
```

```
	exit = False
	while not exit:
		theorical_record.write(str(time.time()) + ' ' + str(time_t) + ' ' +
str(cmd_position[0,0] )+ ' '+ str(cmd_position[0,1] ) + ' ' + str(cmd_position[0,2] )
+ '\n')
		for i in range(num_of_legs):
			input_x = cmd_position[i,0]
			input_y = cmd_position[i,1]
			input_z = cmd_position[i,2]
			#change y component to -y for left side to accomodate the
			#teensy program
			if (i == left_side[0]) or (i == left_side[1]) or (i ==
left_side[2]):
				input_y=-input_y
			( i2c_com(ADDRESS[i], i , (input_x),(input_y),(input_z) ) )
		#time.sleep(0.2)
#the loop_time_offset function is going to allow the robot know that
#when the user pressed the button and when it was released, which t_0, and phase_init
respectively
#Besides that, it also prevent the user to press both moving forward
#and backward at the same time. In the function, btn_array_pre is the array
# to check current button, and btn_array_pre is array fro previous button.
# The last if statement in the function is to ensure that if the leg's movement
# reach to 6*tau, the phase will reset back to zero
def loop_time_offset():
	global btn_array_pre
	global phase_init, t_0, time_t

	if (btn_array_cur[0] == 1):
		print('button X')
	elif (btn_array_cur[1] == 1):
		print('button cricle')
	elif (btn_array_cur[2] == 1):
		print('button triangle')
	elif (btn_array_cur[3] == 1):
		print('button square')
	#print('loop_time_offset')
	################____forward_____#######################
	if (btn_array_cur[2] == 1) and (btn_array_pre[2] == 0):
		t_0 = time.time()
		print(t_0)
	elif (btn_array_cur[2] == 0 and btn_array_pre[2] == 1):
		phase_init = time_t
		print(phase_init)
	elif (btn_array_cur[2] == 1 and btn_array_cur[0] == 1):
		print("nothing")
		t_0 = time.time()
	phase_init = time_t
	if (btn_array_cur[2] == 1 and btn_array_cur[0] != 1 ):
		time_t = time.time() - t_0 + phase_init
		print(time_t)
		if (time_t > 6*tau ) or (time_t < -6*tau):
			t_0 = time.time()
			phase_init = 0
	################____backward_____#######################
```

```
	if (btn_array_cur[0] == 1) and (btn_array_pre[0] == 0):
		t_0 = time.time()
		print(t_0)
	elif (btn_array_cur[0] == 0 and btn_array_pre[0] == 1):
		phase_init = time_t
		print(phase_init)
	elif (btn_array_cur[2] == 1 and btn_array_cur[0] == 1):
		print('nothing')
		t_0 = time.time()
		phase_init = time_t
	if (btn_array_cur[0] == 1 and btn_array_cur[2] != 1 ):
		time_t = -(time.time() - t_0) + phase_init
		print(time_t)
		if (time_t > 6*tau ) or (time_t < -6*tau):
			t_0 = time.time()
			print(t_0)
			phase_init = 0
	#time.sleep(0.1)
	btn_array_pre = btn_array_cur
	#print('previous button')
	#print(btn_array_pre)
#This main function will handle most of the calculation and recording data only
def main():
	exit = False
	while not exit:
		loop_time_offset()
		for i in range(num_of_legs):
			desired_position(i, time_t)
		feed_back =
angle_to_coordinate(leg_info[2,0]*pi/180,leg_info[2,1]*pi/180, leg_info[2,2]*pi/180)

		#print ('FEED_BACK: ')
		#print (str(feed_back[0,0]) + ' ' + str(feed_back[1,0]) + ' '+ str(-
feed_back[2,0]) )
		measured_record.write(str(time.time()) + ' ' + str(time_t) + ' ' +
str(feed_back[0,0] )+ ' '+ str(feed_back[1,0] ) + ' ' + str(-feed_back[2,0]) + '\n')
		#time.sleep(0.2)
print('First run')
print(bounding_box(Z_0))
print(determine_Y_f_and_Y_b(Z_0))
#time.sleep(3)
#There are 3 seperated threads, p1,p2,and p3.
#Since the threads are not in daemons mode, so it keep alive and run
#even if GMADA try to kill it. Therefore, GMADA need to make them be in
#daemon mode. This will allow GMADA to kill the program by simply use
#Ctrl + C
print('Second run')
p1 = Thread(target = RC_input )
p1.daemon = True
p2 = Thread(target = main )
p2.daemon = True
p3 = Thread( target = sending_and_receive_info)
p3.daemon = True
p1.start()
p2.start()
```

```
p3.start()
# the while loop here to keep the program looping.
while True:
       time.sleep(1)
```

6.2 SUB CONTROLLER SOFTWARE

```
#include <math.h>
#include <i2c_t3.h>   //needed to use multiple I2C channels
#include <VL6180X_2.h>  //Needed to use ToF. Modified version needed to use 2nd I2C
for ToF
//Set up for the Proximity sensor
VL6180X proxSensor;
//set the Teensy number of the teensy that is being programmed: A=1, B=2, C=3, D=4,
E=5, F=6
int teensyN=2;
//SLAVE_ADDRESS should be: 0xa for A, 0xb for B, 0xc for C, 0xd for D, 0xe for E, 0xf
for F
#define SLAVE_ADDRESS 0xb
//#include <OneWire.h>
#define PARAMETERS 3
#define FLOATS_SENT 3
#define enc1 2         // defines encoder1 pin   //The #define construct is a
compiler pre-processor
#define enc2 3         // defines encoder2 pin   //directive. The value is
substituted for the name,
#define enc3 4         // defines encoder3 pin   //wherever the name occurs in the
source code.
#define m11 23       //defines pin for motor 1 PWM direction 1
#define m12 22      //defines pin for motor 1 PWM direction 2
#define m21 9      //defines pin for motor 2 PWM direction 1
#define m22 10      //defines pin for motor 2 PWM direction 2
#define m31 20      //defines pin for motor 3 PWM direction 1
#define m32 21      //defines pin for motor 3 PWM direction 2
#define fsrpin 14   //define the pin for the force sensor
//*********************************************************
//********************prototype*************************
//*********************************************************
void p_controller_outline_v3(float,float,float);
void DriveMotor(float, int, int);
void receiveEvent(int);
void sendEvent();
void gettingData(float);
//*********************************************************
//********************Global Array*************************
//*********************************************************
float send_data[FLOATS_SENT];
float processData[PARAMETERS];
//*********************************************************
//********************Global Variable**********************
//*********************************************************
//constants used to calibrate the zero of the encoders
float q1c,q2c,q3c;
float q1d,q2d,q3d;     //desired angle variables
float v1,v2,v3;                         //analog voltage to the motors
float kp1=2.0,kp2=3.0,kp3=4.0;          //position gain for the controlls
int pwm_value;                          //pwm signal to send to the motor
float x, y, z;              //xyz coordinates of the actual position of the foot
relative to the hip joint measured in [m]
```

```
float xd,yd,zd;             //xyz coordinates of the desired position of the foot
relative to the hip joint measured in [m]
//length of the tiba and the femur (constants) measured in [m]
float Lf=0.27978;           //length of the femur measured in [m]. Measurement taken
from solid model
float Lt=0.36830;           //length of the tibia measured in [m]. Measurement taken
from solid model.
                            //tibia measured from joint on femur to foot joint (NOT
bottom of foot)
//*********************************************************
//********************Global Structures*******************
//*********************************************************
// define a struct to hold all ReturnFields to transmit
typedef struct ReturnFields{
  //position variables (q1 is the end segment rotation,
  //q2 is the horizontal hip rotation, q3 is the vertical hip rotation)
  float q1;
  float q2;
  float q3;
  float force;
  float groundProx;
  };
typedef struct ReceiveFields{
  //position variables x,y,z of the foot in [m],
  //position is for the foot joint which is a slight vertical distance above the
ground
  float x;
  float y;
  float z;
  };

//make a data type that can be read as either bytes or the actual data types
//this one is for data sent to the main controller
typedef union SendPacket
{
  ReturnFields Contents;                      //this part is read as floats
  byte ByteContents[sizeof(ReturnFields)];    //this part is read as bytes (legth set
to match the struct)
};

//make a data type that can be read as either bytes or the actual data types
//this one is for data received from main controller
typedef union ReceivePacket
{
  ReceiveFields Contents;                      //this part is read as floats
  byte ByteContents[sizeof(ReceiveFields)];    //this part is read as bytes (legth
set to match the struct)
};
SendPacket LegCurDat;         // make a variable instance of SendPacket
ReceivePacket LegDesPos;     // make a variable instance of SendPacket for test
reading
void setup() {
//set the slave address and angle correction factors based on which teensy is being
programmed. See
//the Summer 2018 documentation to see the procedure to obtain correction factors.
```

```
  if(teensyN>1)
  {
    q1c=-17.84+90,q2c=-111.88,q3c=-303.66;
  }
  else if (teensyN==2)
  {
    q1c=-78.22+90,q2c=-174.99,q3c=-44.56;
  }
  else if (teensyN==3)
  {
    q1c=-125.77+90,q2c=-296.81,q3c=-55.46;
  }
  else if (teensyN==4)
  {
    q1c=22.94+90,q2c=-364.04,q3c=-235.72;
  }
  else if (teensyN==5)
  {
    q1c=72.95+90,q2c=-100.99,q3c=-358.51;
  }
  else if (teensyN==6)
  {
    q1c=-7.03+90,q2c=-325.55,q3c=-242.75;
  }

  Serial.begin(9600); // start serial for output
  Wire.begin(SLAVE_ADDRESS);
  Wire.onReceive(receiveEvent);
  Wire.onRequest(sendEvent);
  Wire.setClock(200000);    //set I2C speed to 400 kHz
Wire1.begin(SLAVE_ADDRESS);
Wire1.setClock(400000);    //set I2C speed to 400 kHz
Wire1.setOpMode(I2C_OP_MODE_IMM);
  analogWriteResolution(10);        //Changes the PWM resolution to 10 bit (0-1023)
  analogWriteFrequency(m11, 21000); // Changes the PWM frequency to about 20kHz
pinMode(enc1,INPUT);      //set encoder reading pins to INPUT pins
  pinMode(enc2,INPUT);
  pinMode(enc3,INPUT);
//set up the ToF sensor
  proxSensor.init();
  proxSensor.configureDefault();
  proxSensor.setTimeout(20);
  proxSensor.setScaling(2);
    LegDesPos.Contents.x=-1000, LegDesPos.Contents.y=-1000, LegDesPos.Contents.z=-
1000;
  }
void loop() {
 if ( LegDesPos.Contents.x != -1000 && LegDesPos.Contents.y != -1000 &&
LegDesPos.Contents.z != -1000 ){  //actual code
  p_controller_outline_v3();
 }}
/*****************************************
 * ***************************************
 * **********RECEIVING DATA FROM MASTER********
 * ***************************************
```

```
 * ****************************************
 */

void receiveEvent(size_t byteCount){
  int s = Wire.available();
  byte b = Wire.read();
  //write the contents of the sent data into LegDesPos.ByteContents
  for(int i = 0; i < s - 1; i++) {LegDesPos.ByteContents[i] =  Wire.read();}
  }
/****************************************
 * ****************************************
 * **********SENDING DATA TO MASTER************
 * ****************************************
 * ****************************************
 */
// callback for sending data
void sendEvent(){
//  Serial.println("sizeof(LegCurDat.ByteContents) = ");
//  Serial.println(sizeof(LegCurDat.ByteContents));
  Wire.write(LegCurDat.ByteContents, sizeof(LegCurDat.ByteContents));
}
void gettingData(float data[]){
    Serial.println("Output from getting data function: ");
    Serial.println(data[0]);
    Serial.println(data[1]);
    Serial.println(data[2]);
}
void p_controller_outline_v3(){
 Serial.println(millis());
      //read the three joint angles
      LegCurDat.Contents.q1 = 180-pulseIn(enc1,HIGH,10000)*360.0/4096.0+q1c;
//LegCurDat.Contents.q1 is the angle of the end segment of the leg
      if(LegCurDat.Contents.q1>180) LegCurDat.Contents.q1=360-LegCurDat.Contents.q1;
      else if (LegCurDat.Contents.q1<-180)
LegCurDat.Contents.q1=360+LegCurDat.Contents.q1;      //constrain
LegCurDat.Contents.q3 to be between -180 and 180 degrees
      LegCurDat.Contents.q2 = pulseIn(enc2,HIGH,10000)*360.0/4096.0+q2c;
//LegCurDat.Contents.q2 is the angle of the
      if(LegCurDat.Contents.q2>180) LegCurDat.Contents.q2=360-LegCurDat.Contents.q2;
      else if (LegCurDat.Contents.q2<-180)
LegCurDat.Contents.q2=360+LegCurDat.Contents.q2;      //constrain
LegCurDat.Contents.q3 to be between -180 and 180 degrees
      LegCurDat.Contents.q3 = pulseIn(enc3,HIGH,10000)*360.0/4096.0+q3c;
      if(LegCurDat.Contents.q3>180) LegCurDat.Contents.q3=360-LegCurDat.Contents.q3;
      else if (LegCurDat.Contents.q3<-180)
LegCurDat.Contents.q3=360+LegCurDat.Contents.q3;      //constrain
LegCurDat.Contents.q3 to be between -180 and 180 degrees
//Serial.print("after encoders millis: ");
//Serial.println(millis());
      //Read the force sensor.
      //When I was testing originally, a 100ish reading was 0-5lb of force, 900 was
20 lb of force
      //The equation below assumes that the slope is constant for every sensor (need
to verify), and that the feet
```

```
      //have been adjusted so that one pound of force will compress the spring and
exert force on the sensor.
      //The feet haven't all be adjusted this way, it is a placeholder.
      //Slope is (20-5)lb/(900-100)reading=0.01875, the offset is the one pound that
the spring supposedely holds
      //Below one pound would be held by the spring and the sensor would not see any
force.
      LegCurDat.Contents.force = (analogRead(14)-100)*0.01875+1.0;
//Serial.print("\tafter FSR sensor millis: ");
//Serial.println(millis());
      //read the proximity sensor
      LegCurDat.Contents.groundProx = proxSensor.readRangeSingleMillimeters();
      if(proxSensor.timeoutOccurred())
      {
          Serial.println("ToF Timeout");
      }
//Serial.print("after ToF sensor millis: ");
//Serial.println(millis());



//run function to convert xyz to q1 q2 q3 values
      leg_pos_des(LegDesPos.Contents.x,LegDesPos.Contents.y,LegDesPos.Contents.z);
      //software controllers for the three joints
      v1= kp1*(q1d-LegCurDat.Contents.q1);
      v2= kp2*(q2d-LegCurDat.Contents.q2);
      v3= kp3*(q3d-LegCurDat.Contents.q3);
      if(v3>0) v3=v3+3.0;         //increase v3 to compensate for gravity
//Serial.print("after voltage calculations millis: ");
//Serial.println(millis());
//actually send voltages/PWM to motors
      DriveMotor(v1,m11,m12);
      DriveMotor(v2,m21,m22);
      DriveMotor(v3,m31,m32);
      troubleshooting();
//      Serial.print("\n");
}
//fuction to send PWM signal to the motor drivers
void DriveMotor(float voltage, int motorpin1, int motorpin2){
      if(voltage<0)     //reverse direction: send PWM to motor
      {
        pwm_value=constrain(abs(voltage)*1023/12,0,1023);  //forces pwm1 to be within
the allowable range for the analog write command
                                                           //max voltage set to 12V, 10
bit PWM range (0-1023)
        analogWrite(motorpin1, pwm_value);     // commands motor to move, 100% = 1023
        analogWrite(motorpin2, 0);        // MUST BE SET TO ZERO!!!
        //Serial.print("pwm = + ");    //output diagnostic info
        //Serial.print(pwm_value);
      }
      else if(voltage>0)  //forward direction: send PWM to motor
      {
        pwm_value=constrain(abs(voltage)*1023/12,0,1023);  //forces pwm1 to be within
the allowable range for the analog write command
        analogWrite(motorpin2, pwm_value);    // commands motor to move, 100% = 1023
```

```
        analogWrite(motorpin1, 0);    // MUST BE SET TO ZERO!!!
        //Serial.print("pwm = - ");    //output diagnostic info
        //Serial.print(pwm_value);
}
      else  //have the motor either coast or stop
      {
        //need to replace with pin to disable the motors if possible
        analogWrite(motorpin1, 0);     // BOTH NEED TO BE ZERO. DO NOT SET BOTH TO
FULL!!!
        analogWrite(motorpin2, 0);     // MUST BE SET TO ZERO!!!
        //Serial.print("pwm = 0");
      }
}
void troubleshooting(){
//display the force sensor reading
      Serial.print("force = ");
      Serial.print(LegCurDat.Contents.force);
//display the proxy sensor reading
      Serial.print("Proxy Distance = ");
      Serial.println(LegCurDat.Contents.groundProx);
//      //display the current angles on the serial monitor
      Serial.print("LegCurDat.Contents.q1 = ");
      Serial.print(LegCurDat.Contents.q1);
      Serial.print("\tLegCurDat.Contents.q2 = ");
      Serial.print(LegCurDat.Contents.q2);
      Serial.print("\tLegCurDat.Contents.q3 = ");
      Serial.print(LegCurDat.Contents.q3);

//      //display the desired coordinates on the serial monitor
      Serial.print("xd = ");
      Serial.print(LegDesPos.Contents.x);
      Serial.print("\tyd = ");
      Serial.print(LegDesPos.Contents.y);
      Serial.print("\tzd = ");
      Serial.print(LegDesPos.Contents.z);
//      //Display the desired angles on the serial monitor
}
//function to calculate the angles q1d,q2d,and q3d from the desired xyz coordinates
//The function calculates values in degrees
void leg_pos_des(float x_l, float y_l, float z_l){   //set up local x y and z
variables for the function
q2d=atan(y_l/x_l)*180.0/M_PI;            //calculate the horizontal rotation of the
leg/hip/femur
q1d=acos((pow(x_l,2)+pow(y_l,2)+pow(z_l,2)-pow(Lf,2)-pow(Lt,2))/(-
2*Lf*Lt))*180.0/M_PI;  //calculate the angle between the femur and tibia
q3d=180.0-q1d-atan(z_l/sqrt(pow(x_l,2)+pow(y_l,2)))*180/M_PI-
asin((Lf*sin(q1d*M_PI/180.0))/sqrt(pow(x_l,2)+pow(y_l,2)+pow(z_l,2)))*180/M_PI;
}
```

6.3 COMPONENT SPECIFICATIONS

6.3.1 ANGLE ENCODER

The MA3 that was used in the project is a miniature rotary absolute shaft encoder that reports the shaft position over 360° with no stops or gaps The MA3 is available with an analog or a pulse width modulated (PWM) digital output. Analog output provides an analog voltage that is proportional to the absolute shaft position. Analog output is only available in 10-bit resolution. PWM output provides a pulse duty cycle that is proportional to the absolute shaft position. PWM output is available in 10-bit and 12-bit resolutions. While the accuracy is the same for both encoders, the 12-bit version provides higher resolution. Three shaft torque versions are available: high torque (-D option), low torque (-N) and ball-bearing (-B). The high and low torque versions have a stainless-steel shaft and brass bushing lubricated with grease to provide ideal torque for panel mount, human-interface applications. The ball-bearing version has a brass shaft and miniature precision ball bearings suitable for high speed and ultra-low torque applications. The ball-bearing version is only available with a 1/8" shaft diameter. Connecting to the MA3 is simple. The 3-pin high retention snap-in 1.25mm pitch polarized connector provides for +5V, output, and ground."

Specification	Sleeve Bushing	Ball Bearing
Moment of Inertia	4.1 x 10^-6 oz-in-s²	4.1 x 10^-6 oz-in-s²
Max. Shaft Speed (1)	100 RPM	15000 RPM
Max. Acceleration	10000 rad/sec²	250000 rad/sec²
Max. Shaft Torque	0.5 ± 0.2 in-oz (D - torque option) 0.3 in-oz (N- torque option)	0.05 in-oz
Max. Shaft Loading	2 lb. dynamic 20 lb. static	1 lb.
Bearing Life (2)	> 1,000,000 revolutions	$L_{10} = (18.3/F_r)^3$ Where L_{10} = bearing life in millions of revs, and F_r = radial shaft loading in pounds
Weight	0.46 oz.	0.37 oz.
Max. Shaft Total Indicated Runout	0.0015 in.	0.0015 in.
Technical Bulletin TB1001 - Shaft and Bore Tolerances		Download

(1) The chip that decodes position uses sampled data. There will be fewer readings per revolution as the speed increases. The formula for number of readings per revolution is given by:

10-bit PWM:

n = 625200 / rpm

12-bit PWM / Analog:

n = 156600 / rpm

(2) only valid with negligible axial shaft loading

Figure 12. Encoder Information

6.3.2 FLEXIFORCE SENSOR

A201 sensors are trimmable and have a 3-pin male connector. The sensors are available in three force ranges: Low 4.4 N (0 - 1 lb), Medium 111 N (0 - 25 lb) and High 445 N (0 - 100 lb)*. The force ranges stated are approximations. The dynamic range of this small, versatile force sensor can be modified by changing the drive voltage and adjusting the resistance of the feedback resistor."

Force Sensor Length:	190.5 mm
	152.4 mm
	101.6 mm
	50.8 mm
Sensing Size (Diameter/Width):	9.7 mm
Standard Force:	4 N
	111 N
	445 N
	4448 N
Temperature Range (Low):	-40 °C
Temperature Range (High):	60 °C

Figure 13. Flexiforce Specifications

6.3.3 TIME OF FLIGHT

The VL6180 also includes an ambient light sensor, or ALS, that can measure the intensity of light with which it is illuminated. Ranging and ambient light measurements are available through the sensor's I²C (TWI) interface, which is also used to configure sensor settings, and two independently-programmable GPIO pins can be configured as interrupt outputs.The VL6180X is a great IC, but its small, leadless, LGA package makes it difficult for the typical student or hobbyist to use. It also operates at voltages below 3 V, which can make interfacing difficult for microcontrollers operating at 3.3 V or 5 V. GMADA's breakout board addresses these issues, making it easier to get started using the sensor, while keeping the overall size as small as possible.

The carrier board includes a low-dropout linear voltage regulator that provides the 2.8 V required by the VL6180X, which allows the sensor to be powered from a 2.7 V to 5.5 V supply. The regulator output is available on the VDD pin and can supply almost 150 mA to external devices. The breakout board also includes a circuit that shifts the I²C clock and data lines to the same logic voltage level as the supplied VIN, making it simple to interface the board with 3.3 V or 5 V systems, and the board's 0.1″ pin spacing makes it easy to use with standard solderless breadboards and 0.1″ perfboards. The board ships fully populated with its SMD components, including the VL6180X, as shown in the product picture."

- Dimensions: 0.5″ × 0.7″ × 0.085″ (13 mm × 18 mm × 2 mm)
- Weight without header pins: 0.5 g (0.02 oz)
- Operating voltage: 2.7 V to 5.5 V

- Supply current: 5 mA (typical; varies with configuration, target, and environment)
- Output format (I^2C): 8-bit distance reading (in millimeters), 16-bit ambient light reading
- Distance measuring range: up to 10 cm (4″) specified; up to 60 cm (24″) possible with reduced resolution. See the graph at the right for typical ranging performance.
- Ranging beyond 10 cm is possible with certain target reflectances and ambient conditions but not guaranteed by specifications. By default, the sensor can report distances up to 20 cm, or it can be configured to measure up to 60 cm with reduced resolution.

6.3.4 KILL SWITCH

Dorman offers a comprehensive line of Toggle Switches for almost any automotive repair. All Toggle Switches are constructed of high-quality materials for long-lasting durability. Refer to product specification available in the references.

Figure 14. Kill Switch

6.3.5 LOW VOLTAGE WIRING DIAGRAM

Below is the wiring schema for the low voltage grid on the robot.

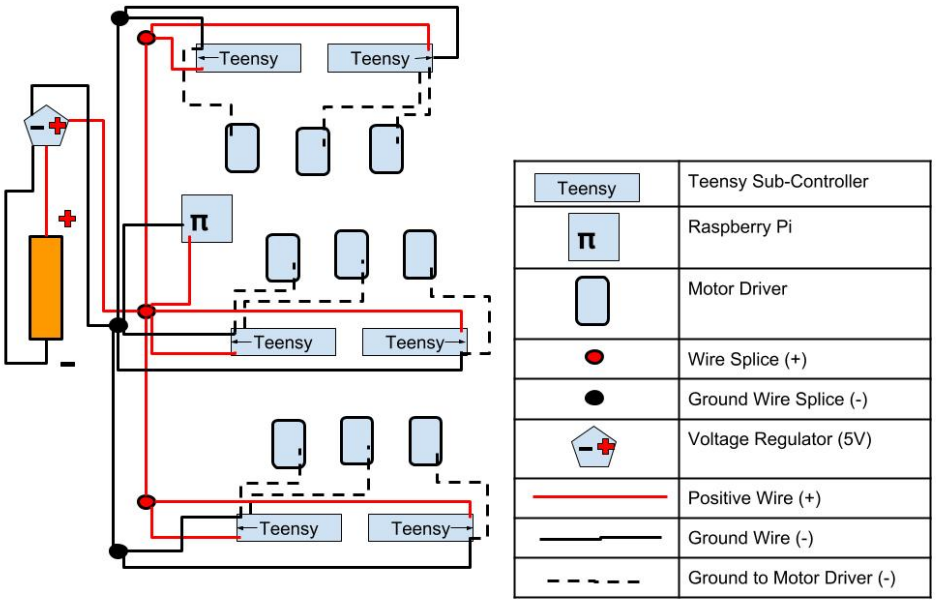


Figure 15. Low Voltage Power Grid Diagram

6.3.6 HIGH VOLTAGE WIRING DIAGRAM

Below is the wiring schema for the high voltage grid on the robot.

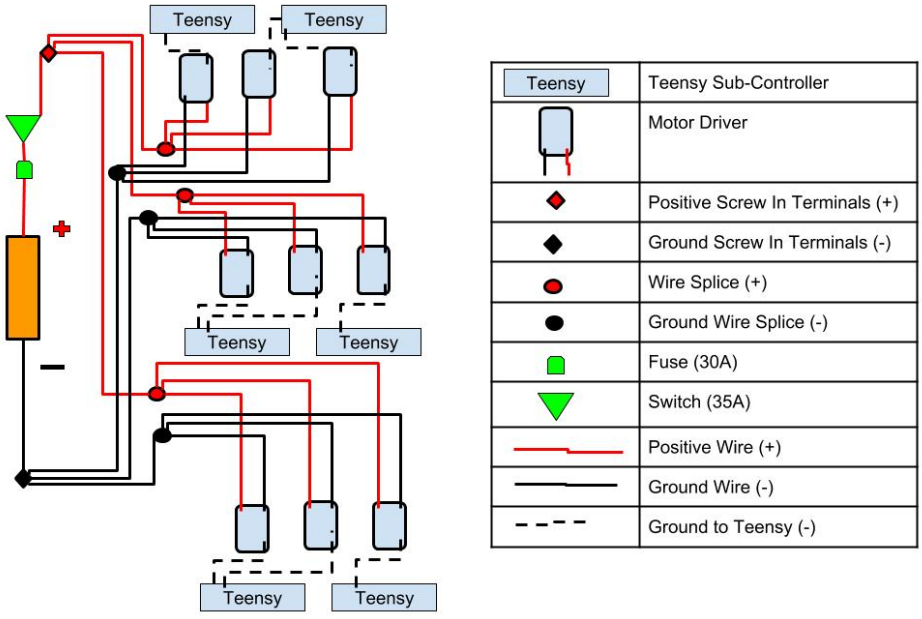


Figure 16. High Voltage Power Grid Diagram

6.3.6 DRIVE BELT SPECIFICATIONS

The hip joints (J1) are powered using reinforced drive belts. These belts were originally present in the system. Two broke during troubleshooting late in the Spring 2018 semester. Efforts to identify and replace the belt were successful. The belts are 112MXL012 Drive Belts purchased from McMaster-Carr during the Summer 2018 Semester. A product link and other details are available in the BOM for the Spring 2018 Semester.

6.3.7 COMMUNICATION ARCHITECTURE

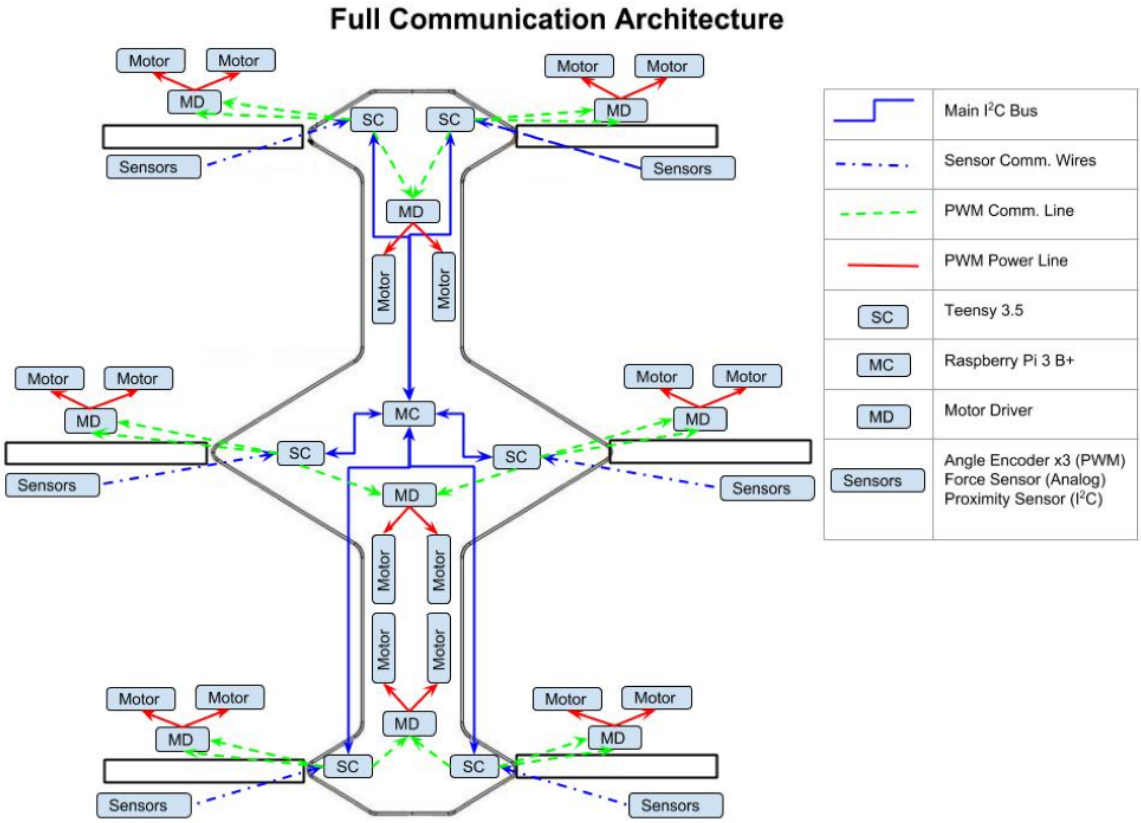


Figure 17. COMMUNICATION SCHEMA

6.3.8 COMPONENT PINOUT DIAGRAMS

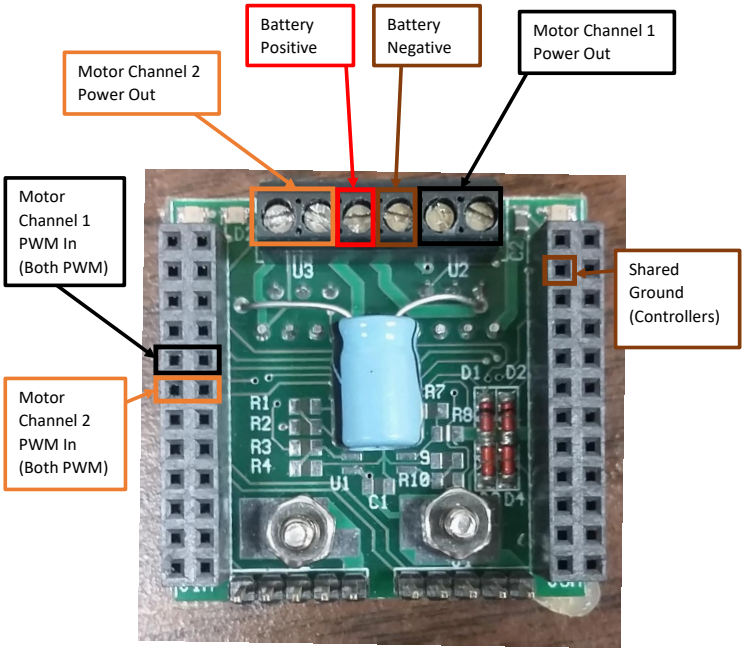


Figure 18. Motor Driver Pinout

Main Power Ground	GND			Vin (3.6	Voltage Regulator (+ 5V)
	0			Analog (	Force Sensor Circuit Analog Ground
	1			3.3V (25	Force Sensor Circuit Power
Tibia Encoder PWM In	PWM	2		23 A9	Tibia PWM out (Hip Driver)
Hip Encoder PWM In	PWM	3		22 A8	Tibia PWM Out (Hip Driver)
Femur Encoder PWM In	PWM	4		21 A7	Femur PWM Out (Leg Driver)
	PWM	5		20 A6	Femur PWM Out (Leg Driver)
	PWM	6		19 A5	Main I²C (SCL0)
	PWM	7		18 A4	Main I²C (SDA0) [currently red wire]
	PWM	8		17 A3	
Hip PWM Out (Leg Driver)	PWM	9		16 A2	
Hip PWM Out (Leg Driver)	PWM	10		15 A1	
		11		14 A0	Force Sensor Analog Input
		12		13 (LED)	
	3.3V			GND	Force Sensor Circuit Digital Ground
		24		A22	
		25		A21	
		26		39 A20	
		27		38 A19	ToF Sensor I²C (SDA1)
		28		37 A18	ToF Sensor I²C (SCL1)
	PWM	29		36 A17	
	PWM	30		35 A16	
	A12	31		34 A15	
	A13	32		33 A14	

Image Source: https://www.pjrc.com/teensy/pinout.html

Encoder Wires (old cables)	
Red	3.3V in
Black	Ground
Green	PWM out

Teensy Program Motor Designations			
Motor Number	Segment Controlled	Angle Variable	Angle in Original Derivation
1	Tibia	q1	γ
2	Hip	q2	ϕ
3	Femur	q3	θ

Figure 19. Teensy Pinout

6.4 ELECTRICAL FAILURE INCIDENT REPORT

The electrical system of the hexapod robot was partially destroyed the week before the final presentation. This incident was caused by the positive and negative wires being plugged into the motor driver backwards on at least one set of motor drivers. The total damage that was attributed to this incident included four Teensy 3.5 that no longer work, one kill switch, and four motor drivers that have at least one nonfunctional channel or that have a ground pin that doesn't connect to ground.

The motor drivers were wired into a remote kill switch that was designed to allow GMADA to turn the power to the motors off remotely. This was accomplished by routing the positive voltage line through the kill switch before it went to the motor drivers. There were two motor drivers

attached to three of the kill switch channels and three motor drivers attached to the fourth channel. This was done to ensure that the individual channels of the motor driver didn't exceed the power rating of the channel. There was no drawn out wiring diagram of the overall high voltage power system, but the design had been discussed verbally and agreed upon. Additionally, diagrams had been made that showed how the wires needed to be connected to an individual motor driver.

When GMADA initially tested the high voltage power system, GMADA connected the battery, but all the kill switch channels were turned off. GMADA pressed one of the buttons to turn on a channel of the kill switch and noticed some slight popping sounds. GMADA immediately disconnected the power. Next, GMADA checked the parts for damage, but GMADA didn't notice any visible damage. GMADA thought that something may have been odd with the kill switch instead of the main wiring of the high voltage power system. GMADA decided to try to turn the system on again to see if it was just the one channel of the kill switch that had an issue or if GMADA were imagining the issue. When GMADA connected the power for the second time, GMADA noticed that smoke was coming from the kill switch. When GMADA removed the kill switch, GMADA saw the damage shown in Figure 20.

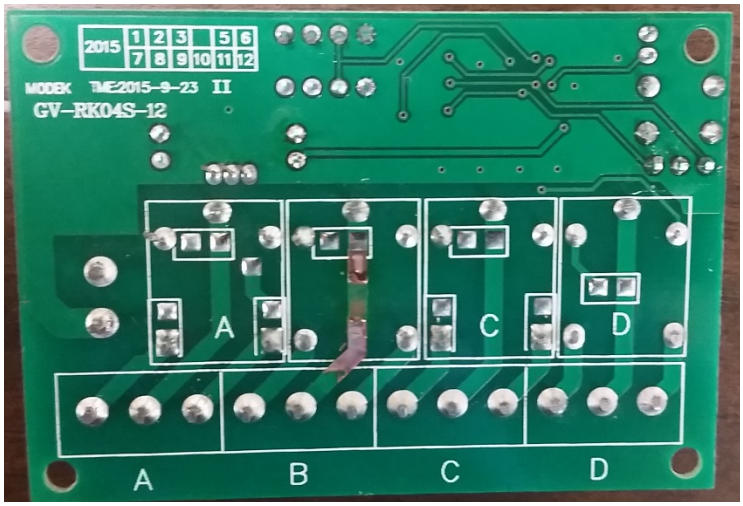


Figure 20. Burned Out Trace On Kill Switch

After the incident, GMADA decided to disconnect and rebuild the electrical system because it would have been very difficult to discover the issue with the large number of wires that were present. Initially, GMADA did not expect the cause to be reversed inputs into one of the motor drivers. After GMADA had disconnected some of the motor drivers and most of the data cables, GMADA noticed that the motor drivers were not plugged in according to the wiring diagram. You can see the incorrect wiring in Figure 21.

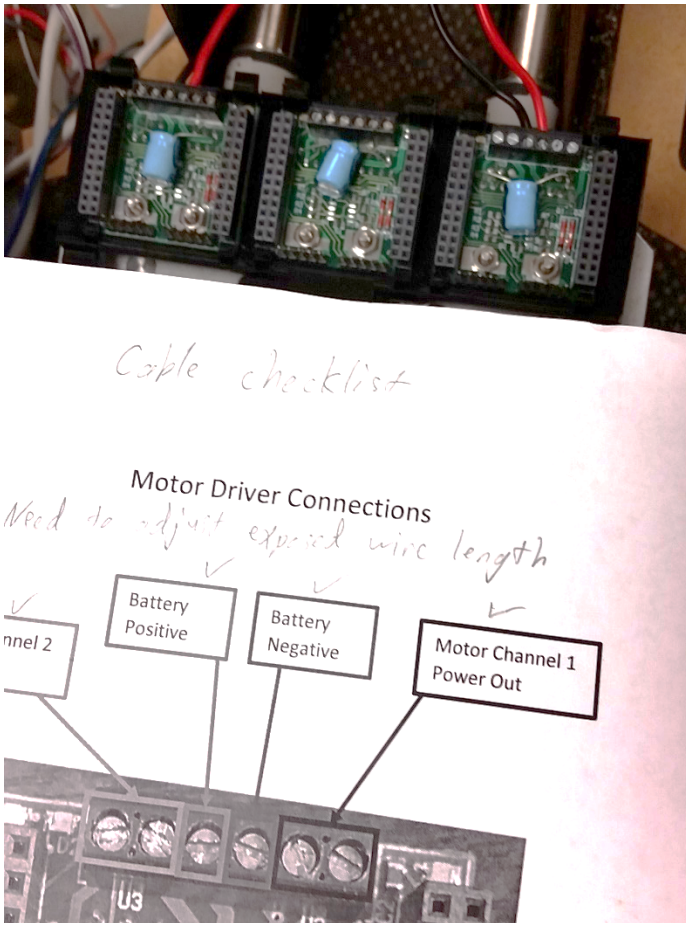


Figure 21. Incorrect Motor Driver Wiring Next to Wiring Diagram

While investigating the incident, four Teensy 3.5 sub-controllers, that were known to be working, were found to be unresponsive when connected to a computer via USB. Additionally, four motor drivers were found to have some type of issue. The motor drivers either had at least one channel that was unresponsive, or the motor driver ground pin showed a nonzero voltage compared to the ground power input. Therefore, the total hardware loss from the incident was 4 Teensy 3.5 sub-controllers, 4 motor drivers, and 1 kill switch. Each Teensy 3.5 cost approximately $31.25 while the motor drivers cost approximately $70.00. Lastly, the kill switch cost approximately $16.00. This brings the hardware cost of the incident to $421.00 dollars.

The incident also set the project back a significant amount of time. It took approximately 50 man hours to fix the electrical system. For an hourly cost of $100.00 per man hour, the labor cost would come to $5000.00. This brings the total cost of the incident to $5421.00.

In order to avoid this type of incident in the future, a complete power wiring diagram that shows the connections to the positive and negative terminals of the battery should be drawn out ahead of time so that any potential short circuits or other issues can be identified. Additionally, the wiring diagrams for individual components should be referenced while connecting the wires. This is especially true for any device that is directly connected to the high voltage power system. Finally, the direct connections to the high voltage power system should be double checked before the power is turned on for the first time to ensure that all wires are fully plugged in and are inserted to the correct terminal on the device. Lastly, a fuse should be added to any higher voltage systems to ensure that the fuse will burn out before more expensive electronics. The practices that are

recommended in this report were used while rebuilding the high voltage power system in the hexapod robot. These practices helped enable GMADA to get the high voltage power system rebuilt without losing any further hardware.

6.5 INVERSE KINEMATICS AND GAIT DEVELOPMENT

The analysis in this document relies heavily on the material in the textbook *Vector Mechanics: A Systematic Approach, Second Edition*, by Dr. Alan Bowling.

It is typical in dynamic systems to define the system's kinematics and dynamics in terms of the inertial reference frame. The relationship between the components and the observer is of paramount importance to the analysis of the system. However, in the case of the hexapod robot, it is more important to focus on the components' movement with respect to each other. The accumulated numerical values of the robot's actual displacement and rotation with respect to the inertial reference frame do not significantly affect the gait algorithm. It is only necessary to store the changes in the body's displacement and rotation with respect to the environment for as long as necessary before these values can be discarded and updated.

This is because the robot in its current form is effectively blind to the workings of the environment. The only sensory input it currently has to work with is the feedback angles of its legs, the force on the feet, and potentially the distance said feet are above the ground. This is not an issue because the robot is directly controlled by an operator at a distance. This human operator is expected steer the robot so that it avoids environmental hazards. It is for these reasons that the kinematic analysis of the robot focuses on the frame and point attached to the main body.

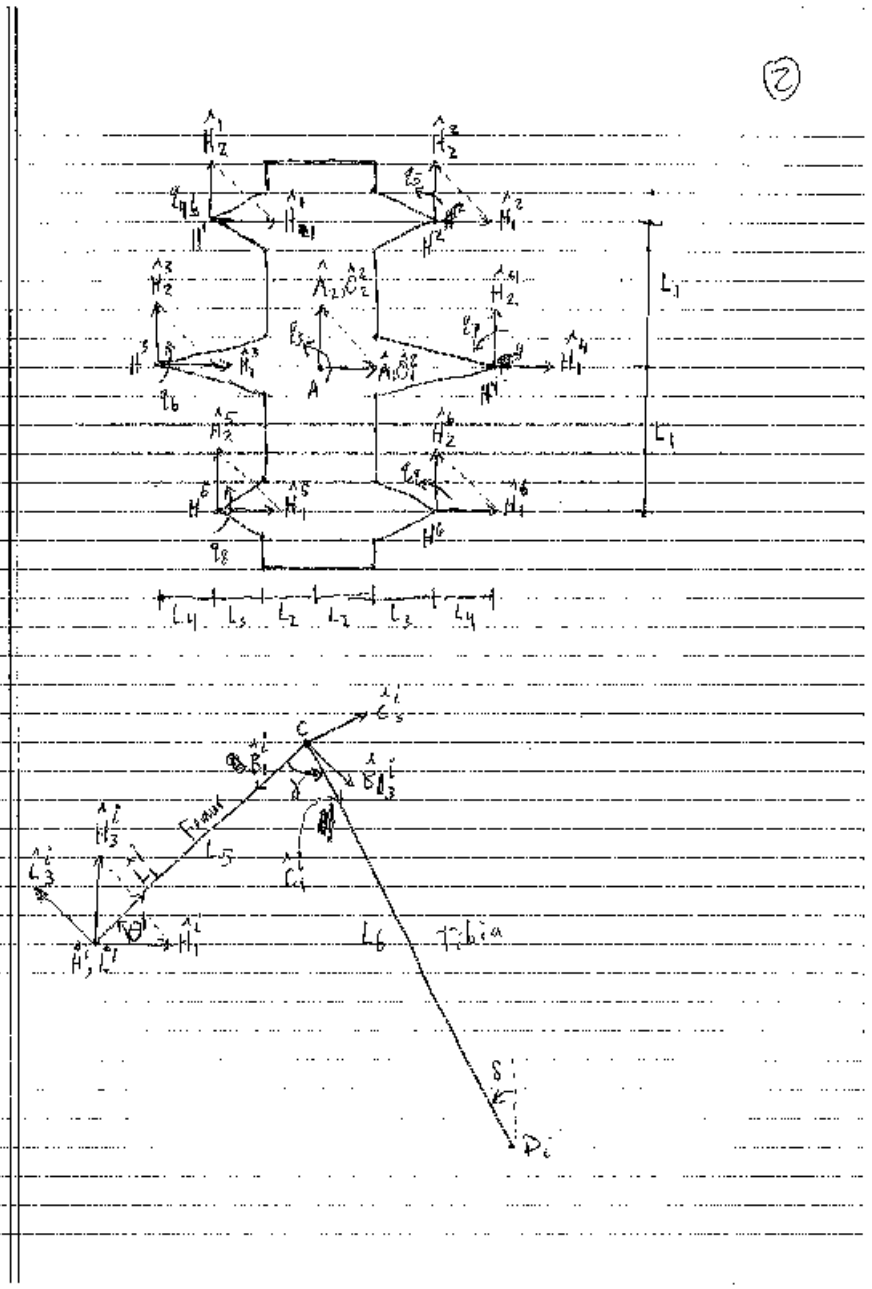
One begins the location description of the robot by defining the body attached point and frame. These are labeled point A and frame A. From there, the frames that are connected to the hips, the frames connected to each knee, and finally how these relate to the inertial reference frame and point are then defined. Each step of this analysis must focus on how all of the components affect the displacement of point A and rotation of frame A.

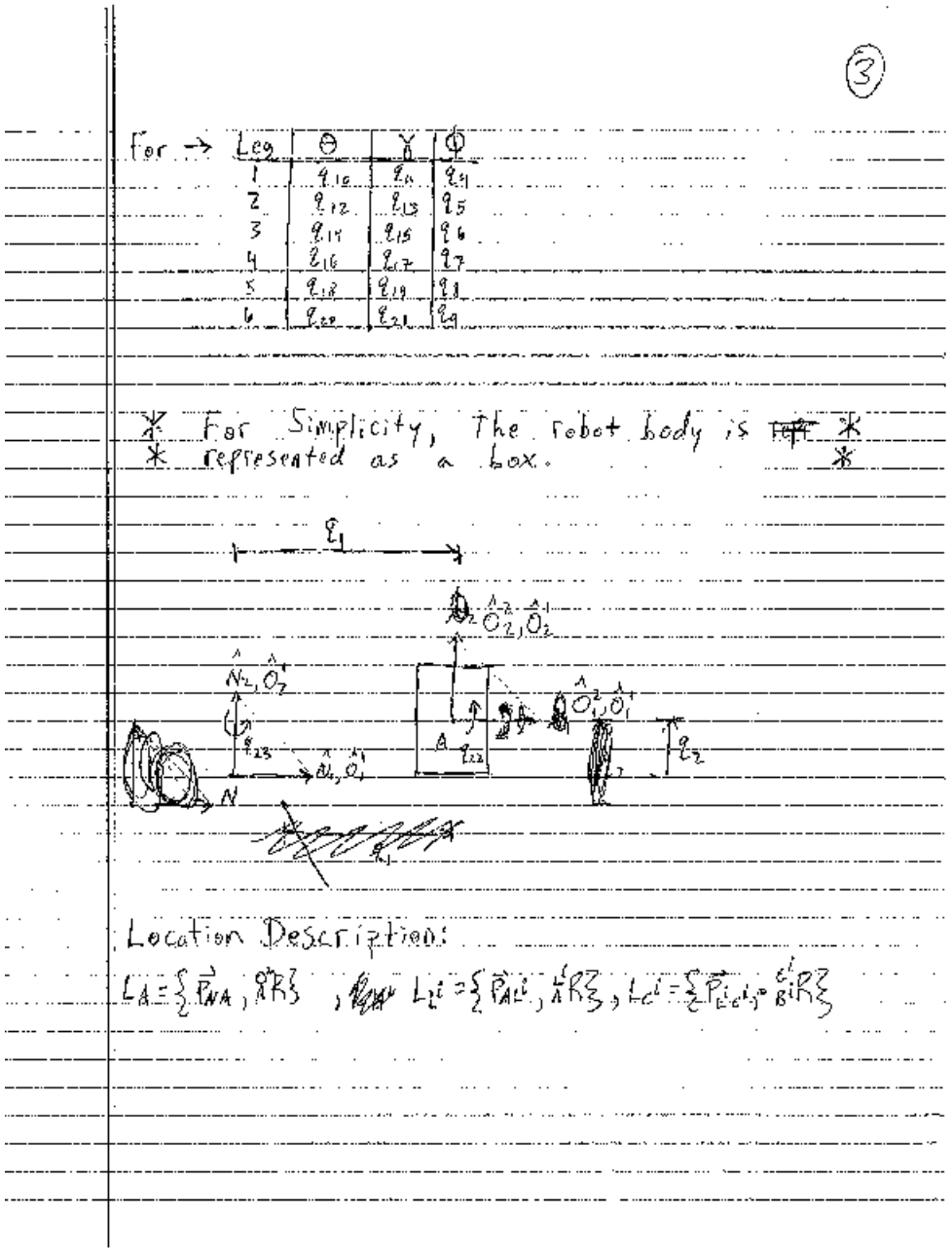
There are six legs on the hexapod that must each have their own set of frames and points. It is thus necessary to set up a systematic notation to prevent confusion between which set of frames corresponds to which leg. Each leg is assigned a number from one to six. The subscript on each frame vector is a tag that indicates for time and all eternity which vector is associated with a particular orthogonal direction. (Bowling, 2016) There is an additional superscript added to each frame vector to indicate which leg it is associated with. For example, the point and frame rotating around the vector normal to the top face of the robot, and attached hip one, are written as follows:

$$Point\ H^1$$

$$Frame\ H^1 = (\hat{H}_1^1, \hat{H}_2^1, \hat{H}_3^1)$$

The full location description is as follows.


(2)
Femur
Tibia



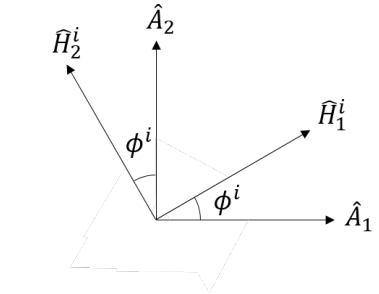
Location Description | **Position Vector** | **Rotation matrix**

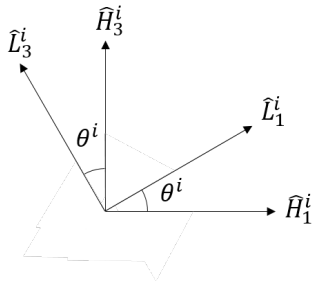
$$L_A = \left\{\vec{P}_{NA} \mid {}^{O^2}_{A}R\right\} + geom. \qquad \vec{P}_{NA} = q_1\hat{A}_1 + q_2\hat{A}_2 \qquad {}^{O^2}_{A}R = \begin{bmatrix} c_3 & -s_3 & 0 \\ s_3 & c_3 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

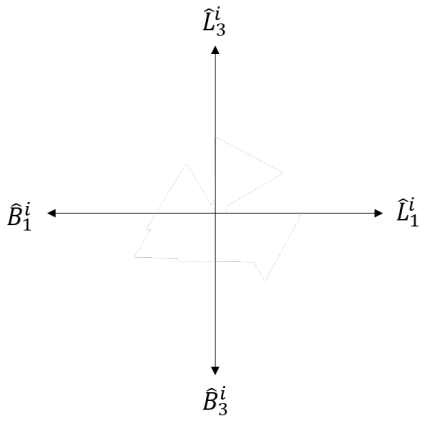
$$L_{L^i} = \left\{\vec{P}_{AL^i} \mid {}^{L^i}_{H^i}R\right\} + geom. \qquad \vec{P}_{AL^i} = L_{x_i}\hat{A}_1 + L_{y_i}\hat{A}_2 \qquad {}^{L^i}_{H^i}R = \begin{bmatrix} c_{\theta^i} & 0 & -s_{\theta^i} \\ 0 & 1 & 0 \\ s_{\theta^i} & 0 & c_{\theta^i} \end{bmatrix}$$

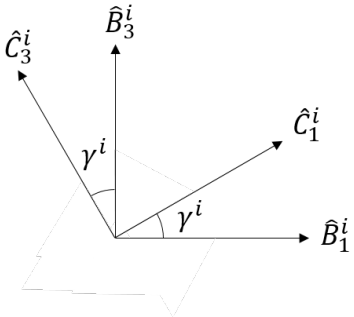
$$L_{C^i} = \left\{ \vec{P}_{L^i C^i} \mid {}^{C^i}_{B^i}R \right\} + geom. \qquad \vec{P}_{L^i C^i} = L_{f_i} \hat{L}^i_1 \qquad {}^{C^i}_{B^i}R = \begin{bmatrix} c_{\gamma^i} & 0 & -s_{\gamma^i} \\ 0 & 1 & 0 \\ s_{\gamma^i} & 0 & c_{\gamma^i} \end{bmatrix}$$

Simple Rotation









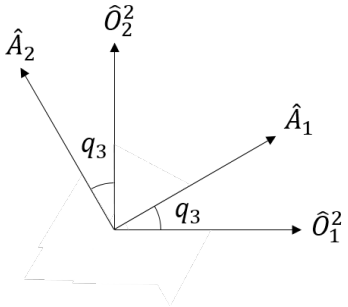
Frame Definition Equations

$$Frame\ H^i = \begin{cases} \hat{H}^i_1 &= c_{\phi^i}\hat{A}_1 + s_{\phi^i}\hat{A}_2 \\ \hat{H}^i_2 &= -s_{\phi^i}\hat{A}_1 + c_{\phi^i}\hat{A}_2 \\ \hat{H}^i_3 &= \hat{A}_3 \end{cases}$$

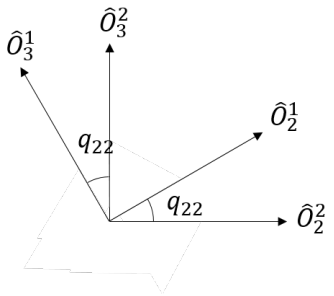
$$Frame\ L^i = \begin{cases} \hat{L}^i_1 &= c_{\theta^i}\hat{H}^i_1 + s_{\theta^i}\hat{H}^i_3 \\ \hat{L}^i_2 &= \hat{H}^i_2 \\ \hat{L}^i_3 &= -s_{\theta^i}\hat{H}^i_1 + c_{\theta^i}\hat{H}^i_3 \end{cases}$$

$$Frame\ B^i = \begin{cases} \hat{B}^i_1 &= -\hat{L}^i_1 \\ \hat{B}^i_2 &= \hat{L}^i_2 \\ \hat{B}^i_3 &= -\hat{L}^i_3 \end{cases}$$

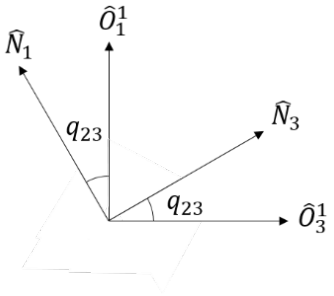
$$Frame\ C^i = \begin{cases} \hat{C}^i_1 &= c_{\gamma^i}\hat{B}^i_1 + s_{\gamma^i}\hat{B}^i_3 \\ \hat{C}^i_2 &= \hat{B}^i_2 \\ C^i_3 &= -s_{\gamma^i}\hat{B}^i_1 + c_{\gamma^i}\hat{B}^i_3 \end{cases}$$



$$Frame\ A = \begin{cases} \hat{A}_1 &= c_3\hat{O}_1^2 + s_3\hat{O}_2^2 \\ \hat{A}_2 &= -s_3\hat{O}_1^2 + c_3\hat{O}_2^2 \\ \hat{A}_3 &= \hat{O}_3^2 \end{cases}$$



$$Frame\ O^1 = \begin{cases} \hat{O}_1^1 &= \hat{O}_1^2 \\ \hat{O}_2^1 &= c_3\hat{O}_2^2 + s_3\hat{O}_3^2 \\ \hat{O}_3^1 &= -s_3\hat{O}_2^2 + c_3\hat{O}_3^2 \end{cases}$$



$$Frame\ N = \begin{cases} \hat{N}_1 &= c_{23}\hat{O}_1^1 - s_{23}\hat{O}_3^1 \\ \hat{N}_2 &= \hat{O}_2^1 \\ \hat{N}_3 &= s_{\gamma^i}\hat{O}_1^1 + c_{\gamma^i}\hat{O}_3^1 \end{cases}$$

A frame relation diagram shows the how the frames in a dynamic system connect. This is a bookkeeping tool that enables one to keep track of frames even when dealing with more complicated systems.

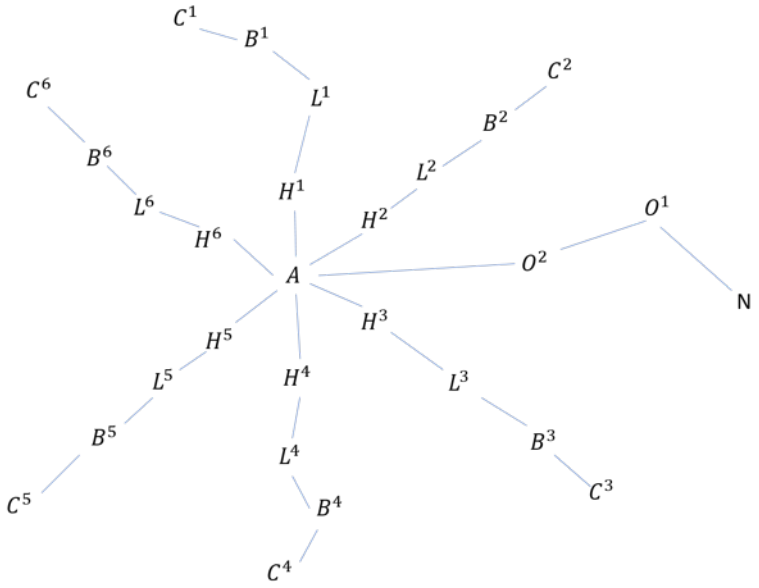


Figure 22. Frame Relationship Diagram

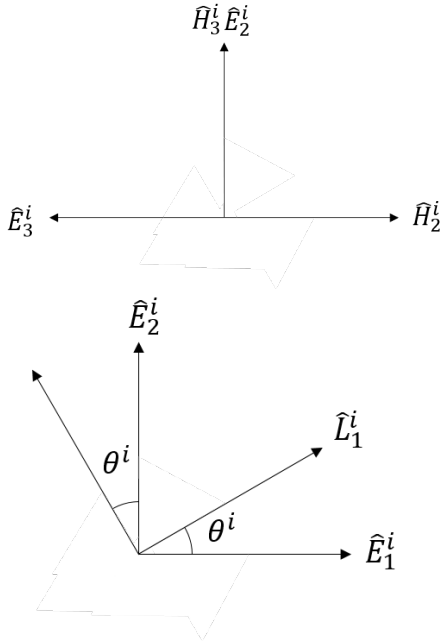
Each node represents a frame used in the location description. The frame relation diagram allows one to quickly determine which rotation matrices are needed to convert from some starting frame and a corresponding ending frame. A line connecting adjacent nodes implies that there exists at least one invertible rotation matrix between the adjacent nodes. For example, nodes H^4 and L^4 are connected with a straight line. This means that the rotation matrix required to travel from node H^4 to node L^4 is ${}^{L^4}_{H^4}R$, and that one can also convert frame L^4 to frame H^4 using ${}^{H^4}_{L^4}R$.
If there is an arrow connecting two nodes, then the rotation matrix between those two frames is not invertible. The direction of transformation follows the direction of the arrow. Had nodes H^4 and L^4 been connected with an arrow starting at node $L^4$and ending at node H^4 then there would be no way for any of the kinematic information contained in frame H^4 to affect the kinematics of frame C^4, which describes the angular position of tibia number 4.
It is recommended that frame relation diagrams be constructed simultaneously with, or immediately after, the initial free body diagram has been drawn up. Frame relation diagrams can be drawn up before the components of the rotation matrices have been defined. This helps protect from errors caused by improperly connecting the frames to each other, and leaves the rest of the location description readily open to future modifications.

See previous reports for derivation of the inverse kinematic equations. Note that for them to be compatible with the current location description, the equation for ϕ must be changed to

$$\phi = \tan^{-1}\left(\frac{x}{y}\right)$$

It is likely that this discrepancy can be rectified by adding an intermediate frame in between H^i and L^i like so:

Simple Rotations (Proposed)



Frame Definition Equations (Proposed)

$$Frame\ E^i = \begin{cases} \hat{E}_1^i &= \hat{H}_1^i \\ \hat{E}_2^i &= \hat{H}_3^i \\ \hat{E}_3^i &= -\hat{H}_2^i \end{cases}$$

$$Frame\ L^i = \begin{cases} \hat{L}_1^i &= c_{\theta^i}\hat{E}_1^i + s_{\theta^i}\hat{E}_2^i \\ \hat{L}_2^i &= -s_{\theta^i}\hat{E}_1^i + c_{\theta^i}\hat{E}_2^i \\ \hat{L}_3^i &= \hat{E}_3^i \end{cases}$$

There are a variety of methods used by animals and machines to orient themselves relative to the terrain. These methods vary from accelerometers in the human ear and their machine analogs to more sophisticated techniques based on image recognition. The robot currently does not have any of these capabilities. This calls for a different approach in order for it to have the ability to orient itself properly with respect to the terrain.

When the robot's legs are on the ground, the force sensors are reporting a force with a given value. The actual value is not important as long as it is above the threshold that defines when the foot is in the air or on the ground. If one assumes that the ground is reasonably flat, then there is a purely geometrical method to approximate the orientation of the robot relative to the ground. The derivation of this process is as follows.

Assume actual shape of the terrain can be well approximated by a plane. The general equation of a plane is

$$ax + by + cz + d = 0$$

Because the feet are on the ground, their (x,y,z) coordinates can be interpreted as points that lie on the plane of the terrain. Fortunately, the objective is to derive a relationship between the orientation of the ground and the robot body only; the vertical shift is not important for determining orientation. Thus, one can divide the plane equation by d like so:

$$\alpha_1 x + \alpha_2 y + \alpha_3 z + 1 = 0$$
$$\vec{\alpha} \cdot \vec{P}_{HC} = -1$$

where

$$\vec{\alpha} = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix}, \qquad \vec{P}_{HC} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

The position vectors of each foot are defined in terms of (x,y,z) coordinates. These coordinates can be stored in the position vector $\vec{P}_{H_iC_i} = [x_i \quad y_i \quad z_i]^T$. The variable n to represents the quantity of legs in contact with the terrain, and $\vec{\alpha}$ is a vector normal to the terrain.

There is an issue with this model, however. If the robot is employing a three-legged gait, then one can solve for the orientation vector directly, but if the robot is using a two-legged or one-legged gait then the modified plane equation is over defined. The method of least squares has been chosen to extend the plane approximation to gaits with an arbitrarily large number of feet in contact with the terrain.

The method of least-squares is a curve fitting technique that minimizes the squared residual between the desired curve and the data points ("Least Squares," n.d.) The desired curve for this problem is a plane. The components of $\vec{\alpha}$ are the unknown variables that will be used to minimize the squared residual. The (x,y,z) coordinates of the feet are already known from the robot's geometry and the feedback from the encoders. There is now enough information to determine the coefficients of $\vec{\alpha}$.

Let ${r_i}^2 = (\vec{\alpha} \cdot \vec{P}_{H_iC_i} + 1)^2$ be the squared residual between the coordinates of a given leg and the terrain plane.

The sum of these squared residuals, S, is the quantity to be minimized. Let the gradient operator be

$$\nabla = \begin{bmatrix} \frac{\partial}{\partial \alpha_1} & \frac{\partial}{\partial \alpha_2} & \frac{\partial}{\partial \alpha_3} \end{bmatrix}^T = [\partial_{\alpha_1} \quad \partial_{\alpha_2} \quad \partial_{\alpha_3}]^T$$

because α_1, α_2, and α_3 are the unknown variables that affect the least-squares error. The values of the components of $\vec{\alpha}^T$ can now be calculated as follows.

$$S = \sum_{i=1}^{n} {r_i}^2 = \sum_{i=1}^{n} \left(\vec{\alpha} \cdot \vec{P}_{H_i C_i} + 1\right)^2$$

$$\nabla S = \nabla \sum_{i=1}^{n} {r_i}^2 = \nabla \sum_{i=1}^{n} \left(\vec{\alpha} \cdot \vec{P}_{H_i C_i} + 1\right)^2 = 0$$

$$\sum_{i=1}^{n} \nabla {r_i}^2 = \sum_{i=1}^{n} \nabla \left(\vec{\alpha} \cdot \vec{P}_{H_i C_i} + 1\right)^2 = 0$$

$$\sum_{i=1}^{n} 2\nabla r_i = \sum_{i=1}^{n} 2\left(\vec{\alpha} \cdot \vec{P}_{H_i C_i} + 1\right)\left(\nabla\left(\vec{\alpha} \cdot \vec{P}_{H_i C_i}\right) + \nabla 1\right) = 0$$

To simplify further, use the identity $\nabla\left(\vec{\alpha} \cdot \vec{P}_{H_i C_i}\right) = (\nabla \vec{\alpha}^T)\vec{P}_{H_i C_i} + \left(\nabla \vec{P}_{H_i C_i}^T\right)\vec{\alpha}$.

$$\Rightarrow \sum_{i=1}^{n} \left(\vec{\alpha} \cdot \vec{P}_{H_i C_i} - 1\right)\left((\nabla \vec{\alpha}^T)\vec{P}_{H_i C_i} + \left(\nabla \vec{P}_{H_i C_i}^T\right)\vec{\alpha}\right) = 0$$

$$\sum_{i=1}^{n} \left(\vec{\alpha} \cdot \vec{P}_{H_i C_i} + 1\right)\left(\left(\begin{bmatrix} \partial_{\alpha_1} \\ \partial_{\alpha_2} \\ \partial_{\alpha_3} \end{bmatrix}[\alpha_1 \quad \alpha_2 \quad \alpha_3]\right)\vec{P}_{H_i C_i} + \left(\begin{bmatrix} \partial_{\alpha_1} \\ \partial_{\alpha_2} \\ \partial_{\alpha_3} \end{bmatrix}[x_i \quad y_i \quad z_i]^T\right)\vec{\alpha}\right) = 0$$

$$\sum_{i=1}^{n} \left(\vec{\alpha} \cdot \vec{P}_{H_i C_i} + 1\right)\left(\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}\vec{P}_{H_i C_i} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}\vec{\alpha}\right) = 0$$

$$\sum_{i=1}^{n} \left(\vec{\alpha} \cdot \vec{P}_{H_i C_i} + 1\right)\left(\vec{P}_{H_i C_i} + \vec{0}\right) = 0$$

$$\sum_{i=1}^{n} \left(\vec{P}_{H_i C_i}\right)\left(\vec{\alpha} \cdot \vec{P}_{H_i C_i} + 1\right) = 0$$

$$\sum_{i=1}^{n} \vec{P}_{H_i C_i}\left(\vec{P}_{H_i C_i}^T \vec{\alpha} + 1\right) = 0$$

$$\sum_{i=1}^{n} \left(\vec{P}_{H_i C_i} \vec{P}_{H_i C_i}^T\right)\vec{\alpha} + \vec{P}_{H_i C_i} = 0$$

$$\left(\sum_{i=1}^{n} \vec{P}_{H_i C_i} \vec{P}_{H_i C_i}^T\right)\vec{\alpha} = -\sum_{i=1}^{n} \vec{P}_{H_i C_i}$$

$$\vec{\alpha} = -\left(\sum_{i=1}^{n} \vec{P}_{H_i C_i} \vec{P}_{H_i C_i}^T\right)^{-1} \sum_{i=1}^{n} \vec{P}_{H_i C_i}$$

$$\vec{\alpha} = -\left(\sum_{i=1}^{n} \vec{P}_{H_i C_i} \vec{P}_{H_i C_i}^T\right)^{-1} \sum_{i=1}^{n} \vec{P}_{H_i C_i}$$

where		
n	N/A	Number of feet in contact with the ground
$\vec{\alpha}$	$= [\alpha_1 \quad \alpha_2 \quad \alpha_3]^T$	Vector pointing approximately normal to the terrain
$\vec{P}_{H_i C_i}$	$= [x_i \quad y_i \quad z_i]^T$	Vector storing (x,y,z) coordinates of an individual foot in contact with the ground

Note: $\vec{\alpha}$ does not represent the orientation of the robot relative to the ground. The above analysis was done with the assumption that the (x,y,z) coordinates of each foot are expressed in the frame attached to the robot's main body. It is a consequence of using the body frame that the robot body is implicitly assumed to be the level reference surface. Thus, $\vec{\alpha}$ contains orientation information of the ground relative to the robot, and not the more intuitive converse.

This vector will provide all of the necessary information to level the robot with the terrain. Figure [whatever – initial orientation vector relative to body frame A] shows the starting condition of the orientation vector. The goal of the following analysis is to create rotation matrices that will align $\vec{\alpha}$ with $\hat{A}_3$. Assume that the robot is stationary.

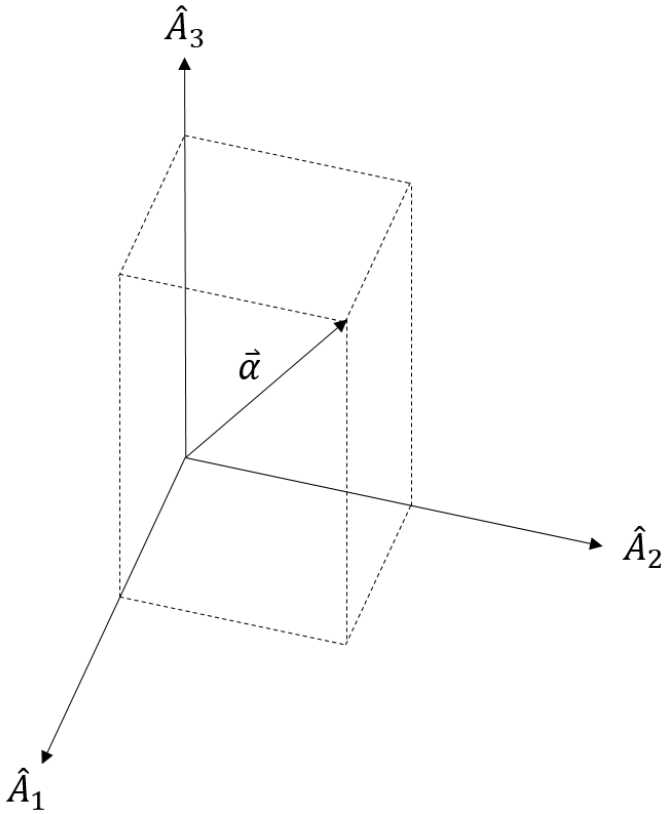


Figure 23. initial orientation vector relative to body frame A

The first matrix can be derived by projecting $\vec{\alpha}$ into the plane formed by $\hat{A}_2$ and $\hat{A}_3$. Figure 24 provides details on exactly what information is available, and how to use that information to get the rotation matrices.

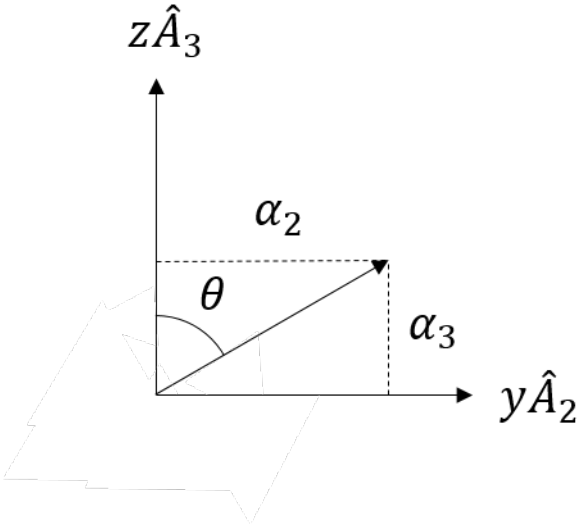


Figure 24. rotate alpha into A^_2 A^_3.plane

The angle θ can be found by noting that $\tan(\theta) = \frac{\alpha_2}{\alpha_3}$. It is also obvious that this is a rotation around $\hat{A}_1$. Thus, the first rotation matrix is

$$R_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix}$$

where

$$\theta = \tan^{-1}\left(\frac{\alpha_2}{\alpha_3}\right)$$

Multiplying $\vec{\alpha}$ by R_1 leads to a new vector, $\vec{\beta} = R_1\vec{\alpha}$. This step is necessary because while $\vec{\beta}$ and $\vec{\alpha}$ have the same magnitude, their orientations are very obviously different in general. This means that their components will have differing values. Thus, the next rotation matrix can be defined from Figure [whatever – rotate $\vec{\beta}$ into $\hat{A}_3$].

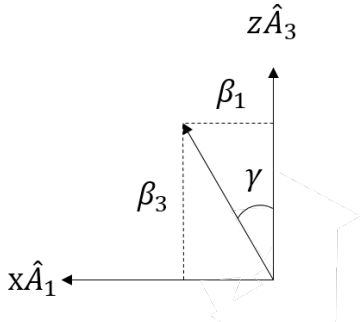


Figure 25. rotate β ⃗ into A^_3

Note that $\tan(\gamma) = \frac{\beta_1}{\beta_3}$. From here, the rotation matrix R_2 must be of the following form, as the transformation shown in Figure [whatever – rotate $\vec{\beta}$ into $\hat{A}_3$] is around $\hat{A}_2$:

$$R_2 = \begin{bmatrix} \cos(\gamma) & 0 & \sin(\gamma) \\ 0 & 1 & 0 \\ -\sin(\gamma) & 0 & \cos(\gamma) \end{bmatrix}$$

where

$$\gamma = \tan^{-1}\left(\frac{\beta_1}{\beta_3}\right)$$

The final step to leveling the robot with the terrain is to multiply each position vector corresponding to the legs like so: $R_2 R_1 \vec{P}_{H_i D_i}$. This will ensure that the robot is parallel to the ground, at least when it is stationary. More work needs to be done in this area to ensure that the robot will remain level even when commanded arbitrary velocities and angular velocities.

Derivation of no-slip constraints for the hexapod's feet

A no slip constraint is as it sounds: a constraint that forces some point to move at the same velocity as the surface it is in contact with. In the case of the hexapod robot, the no slip constraint applies only to the feet on the ground. It implies that whatever the form of the equation governing the ground feet kinematics, the velocity associated with that foot must be zero when in contact with stationary terrain.

Suppose that

$$^N\frac{d\vec{P}_{ND_i}}{dt} = 0$$

Break this velocity into its component vectors:

$$^N\frac{d\vec{P}_{ND_i}}{dt} = {}^N\frac{d(\vec{P}_{NA} + \vec{P}_{AH^i} + \vec{P}_{H^iD^i})}{dt} = 0$$

Where

$$\vec{P}_{NA} = q_1\hat{A}_1 + q_2\hat{A}_2$$
$$\vec{P}_{AH^i} = L_{1_{H^i}}\hat{A}_1 + L_{2_{H^i}}\hat{A}_2$$
$$\vec{P}_{H^iD^i} = x_i\hat{A}_1 + y_i\hat{A}_2 + z_i\hat{A}_3$$

Using the information in the location description, note the following:

$$^A_N\vec{\omega} = {}^{O^1}_N\vec{\omega} + {}^{O^2}_{O^1}\vec{\omega} + {}^A_{O^2}\vec{\omega}$$

Apply the transport theorem to calculate the velocity of $\vec{P}_{ND_i}$. (Bowling, 2016) The general formula for $^N\frac{d\vec{P}_{ND_i}}{dt}$ is unwieldly, tedious to derive, and not particularly instructive. Thus, the following simplifications will reduce $^N\frac{d\vec{P}_{ND_i}}{dt}$ into a form that is more manageable. The overall theme of the simplifying assumptions is to pretend that the robot is walking with its top face parallel to the ground with a constant height.

1. $q_{22} \approx q_{23} \approx 0$
2. $\cos(q_{22}) \approx \cos(q_{23}) \approx 1$

3. $\sin(q_{22}) \approx \sin(q_{23}) \approx 0$
4. $\dot{q_{22}} \approx \dot{q}_{23} \approx 0$
5. $\dot{z} \approx 0$

Applying these five assumptions implies that the desired velocities have the following form:

$$^{N}\frac{d\vec{P}_{NA}}{dt} = (\dot{q_1} - q_2\dot{q_3})\hat{A}_1 + (\dot{q_2} - q_1\dot{q_3})\hat{A}_2$$

$$\frac{d\vec{P}_{AD_i}}{dt} = \left(\dot{x_i} - \left(L_{2_{H^i}} + y_i\right)\dot{q_3}\right)\hat{A}_1 + \left(\dot{y_i} + \left(L_{1_{H^i}} + x_i\right)\dot{q_3}\right)\hat{A}_2$$

The two constraint equations generated by each foot are

$$\dot{q_1} - q_2\dot{q_3} + \dot{x_i} - \left(L_{2_{H^i}} + y_i\right)\dot{q_3} = 0$$

$$\dot{q_2} - q_1\dot{q_3} + \dot{y_i} + \left(L_{1_{H^i}} + x_i\right)\dot{q_3} = 0$$

The only unknowns in these two equations are $\dot{x_i}$ and $\dot{y_i}$. The robot operator has specified $\dot{q_1}, \dot{q_2}, \dot{q_3}$ which also allows the robot to be able to calculate to calculate q_1, q_2, q_3 from the specified velocities. x_i and y_i will be known as well if the robot the equations are being evaluated at $t = t_0 + \Delta t$. This implies that one can solve for $\dot{x_i}$ and $\dot{y_i}$. Now that the new derivative are known, multiply them by Δt to get the new Δ(x,y,z) for the particular foot in question. Add the new incremental (x,y,z) value to the current (x,y,z) values to find the new ground foot position at $t = t_0 + \Delta t$.

Trip avoidance formula

The goal is to derive a formula that will solve for the (x,y) coordinates where two given legs have the potential to collide. It is assumed that all coordinates are in the body frame attached to the top of the robot. There is a temporary coordinate system to facilitate the derivation. It does not affect the kinematics or overall operation of the robot. It is only a simplification used to derive the collision points.

The (x,y) coordinates represent a position vector projected into the plane parallel to the top of the body. It is defined with respect to some arbitrary point in the plane parallel to the top of the robot body. Extra care must be taken during implementation so that there are no errors regarding where the collision points are located. It is advisable to fix point A or point B as the origin of the system.

The Δf term is the distance between the hip joints for the two legs under consideration. It is a fixed parameter based on the geometry of the robot. Points A and B are the fixed points that define Δf.

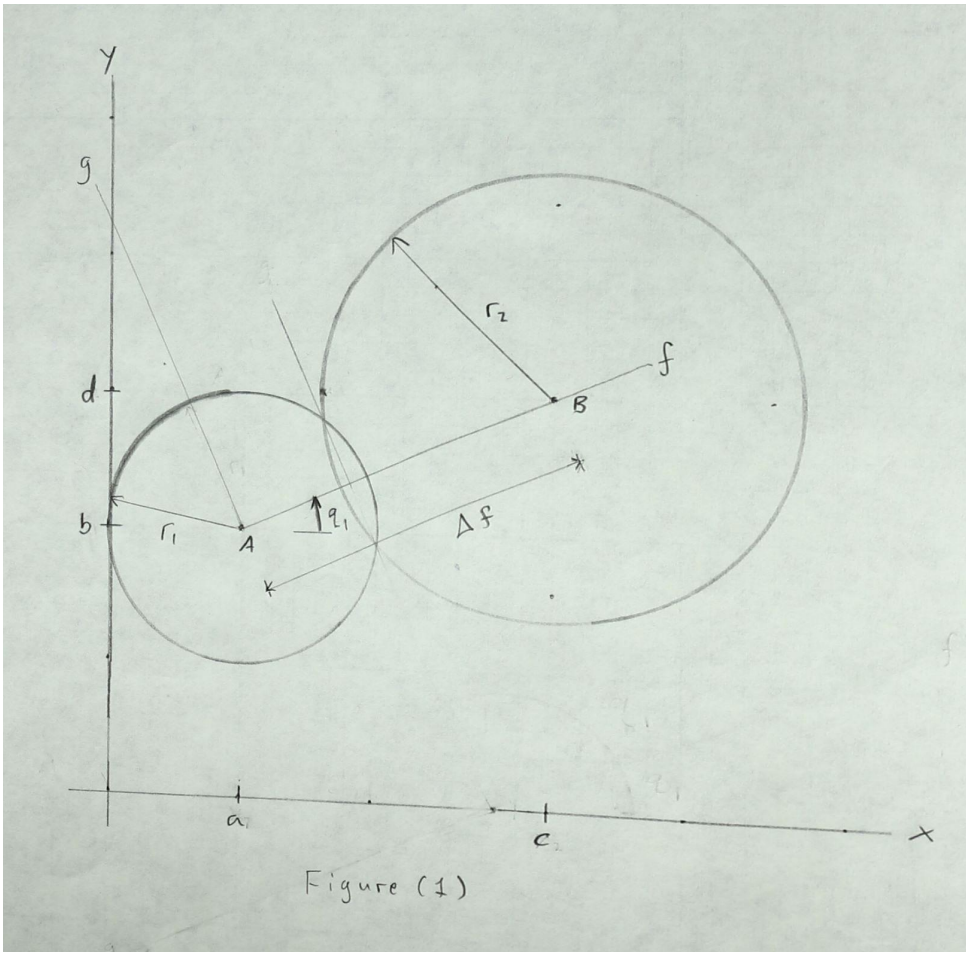


Figure (1)

Define Circle A as the set of points $A(f,g)$ that satisfy $f^2+g^2-r_1^2=0$, and Circle B as the set of$B(f,g)$ that satisfy $(f-\Delta f)^2+g^2-r_2^2=0$. The first task is to determine the location of any intersection points between $A(f,g)$ and $B(f,g)$.

$$0=0$$
$$f^2+g^2-r_1^2=(f-\Delta f)^2+g^2-r_2^2$$
$$f^2+g^2-r_1^2=f^2-2f\Delta f+\Delta f^2+g^2-r_2^2$$
$$-r_1^2=-2f\Delta f+\Delta f^2-r_2^2$$
$$2f\Delta f=r_1^2-r_2^2+\Delta f^2$$
$$\Rightarrow f=\frac{r_1^2-r_2^2+\Delta f^2}{2\Delta f}$$

Now that f is known, insert it back into the equation for circle A to determine the value of g.

$$\left(\frac{r_1^2-r_2^2+\Delta f^2}{2\Delta f}\right)^2+g^2-r_1^2=0$$
$$\Rightarrow g=\pm\sqrt{r_1^2-\left(\frac{r_1^2-r_2^2+\Delta f^2}{2\Delta f}\right)^2}$$

The next step is to transform

The next step is to transform the solutions found in (f, g) coordinates into more convenient (x, y) coordinates. First, relate Δf to the geometry of the robot. Set $a < c$ and $b < d$. This allows one to use the Pythagorean Theorem to obtain the not-so-surprising result:

$$\Delta f = \sqrt{(c-a)^2 + (d-b)^2}$$

Next, observe that (f, g) can be viewed as points rotated around the z-axis, plus a horizontal and vertical offset. This implies that we can convert the solutions in terms of (f, g) coordinates to the desired (x, y) coordinate solutions. This can be done via the following transformation:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \cos(q_1) & -\sin(q_1) \\ \sin(q_1) & \cos(q_1) \end{bmatrix} \begin{bmatrix} f \\ g \end{bmatrix} + \begin{bmatrix} a \\ b \end{bmatrix}$$

The angle q_1 can be found with $\tan(q_1) = \frac{d-b}{c-a}$. This is sufficient to transform the intersection point values so that they are meaningful in (x, y) coordinates.

The equations needed to find the potential intersection points between two hexapod legs can by evaluating the below equations is this particular order:

$$\Delta f = \sqrt{(c-a)^2 + (d-b)^2}$$

$$f = \frac{r_1^2 - r_2^2 + \Delta f^2}{2\Delta f}$$

$$g = \pm\sqrt{r_1^2 - \left(\frac{r_1^2 - r_2^2 + \Delta f^2}{2\Delta f}\right)^2}$$

$$q_1 = \tan^{-1}\left(\frac{d-b}{c-a}\right)$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \cos(q_1) & -\sin(q_1) \\ \sin(q_1) & \cos(q_1) \end{bmatrix} \begin{bmatrix} f \\ g \end{bmatrix} + \begin{bmatrix} a \\ b \end{bmatrix}$$

The last step needed to find the intersection point is a conditional statement to choose the correct value of g. This must be guided by the geometry of the robot. None of the legs can move so far back that they can cross the line formed by the two hip joints. It then stands to reason that if the leg that is further forward on the right side of the robot corresponds to Circle A then one must take the positive value for g. As for the left side of the robot, the relevant point of intersection will be in the negative g direction as long as the leg that is further forward still represents Circle A.

Deriving the bounding box

Each joint on the robot has a maximum and minimum angular displacement before they risk damaging themselves. The knee and two hip joints are currently equipped with absolute angle encoders that provide real-time data about the angles for each of these three joints. However, the ankles are currently not equipped with similar sensors. This presents a problem for developing the hexapod robot's gait because the risk of damaging the robot that is associated with overextending the ankle joints is just as dire as the risks posed by the other leg joints.

The range of allowable ankle angles is known, however. The allowable angles are given by

$$\delta_{min} < \delta < \delta_{max}$$

Physical measurements of the robot's ankles indicate that $\delta_{min} = 0° = 0$ radians, and that $\delta_{max} = 30° = \frac{\pi}{6}$ radians. The bounds on the hip angle ϕ are found by both physical measurements and the trip avoidance algorithm.

Figure [whatever – bounding box top] depicts the planar area described above. Figure [whatever – Side view of leg inside radial bounds]

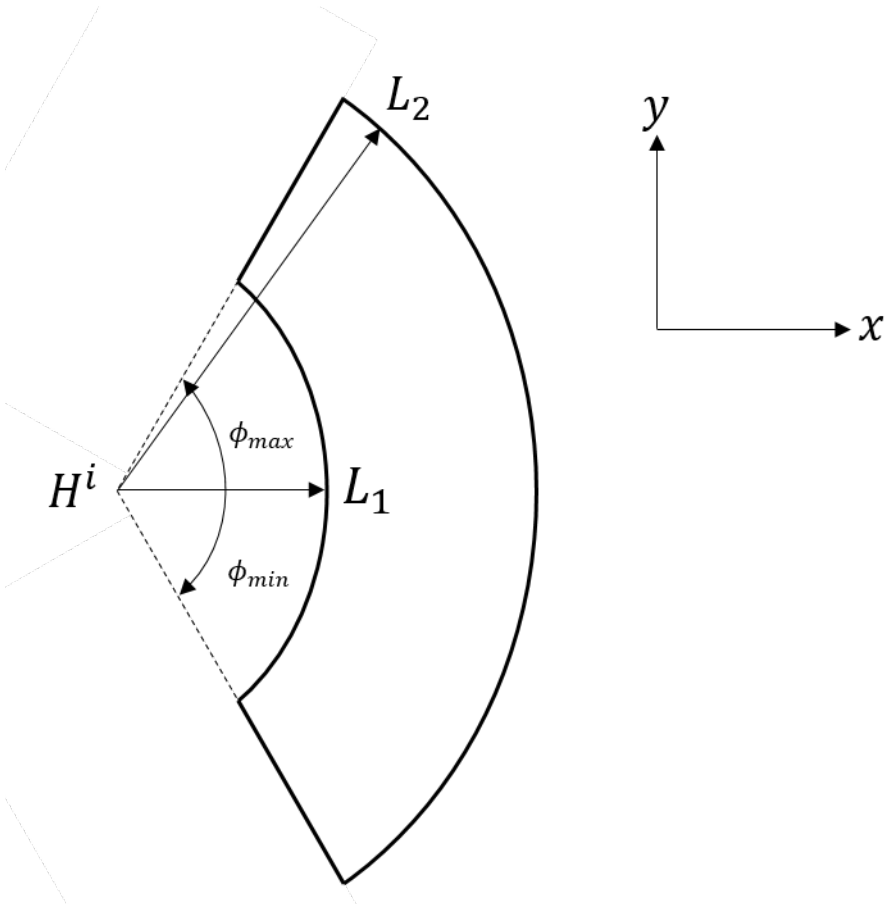


Figure 26. Bounding Box Top

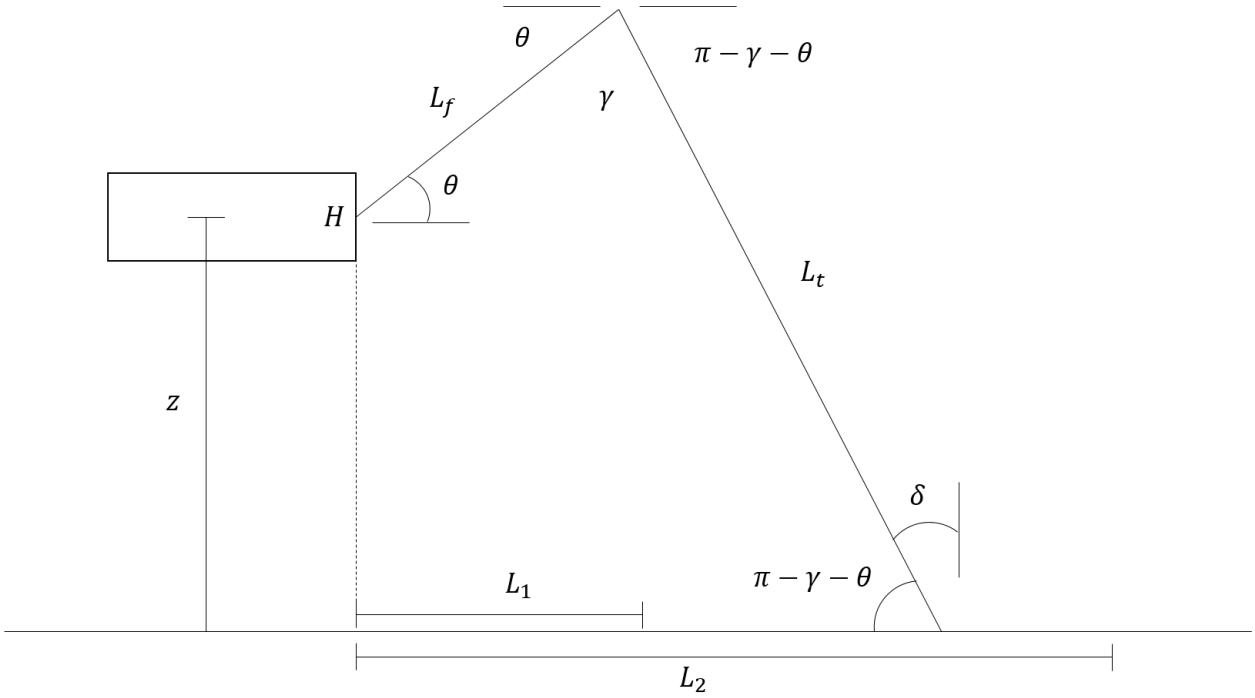


Figure 27. Side view of leg inside radial bounds

The derivation of the bounding box of safe operation for the leg begins with the definition of terms. See table [Bounding box equation terms]

L_1	Inner radius of bounding box
L_2	Outer radius of bounding box
ϕ	Angle of leg rotation parallel to the top robot surfce
θ	Commanded femur angle
γ	Commanded tibia angle relative to the femur
δ	Angle of ankle displacement
L_f	Length of Femur
L_t	Length of Tibia

Combining the bounds on δ with the geometry of the leg leads to the following result:

$$\frac{\pi}{2} + \delta_{min} < \theta + \gamma < \frac{\pi}{2} + \delta_{max}$$

Next, apply a loop constraint to the leg to get the following starting equation in the z-direction:

$$L_T \cos(\pi - \theta - \gamma) - z - L_f \sin(\theta) = 0$$

Solve for γ, and combine with the modified bound equation to find the first set of equations that give a range of angles satisfying the bounding box criteria. This method is necessary because there is a non-zero chance that some of the z displacements will damage the joints if not closely monitored. After exhausting all combinations that could cause trouble, the values of L_1 and L_2 will be the ones that satisfy all external bounds on the range of motion of the hexapod's joints.

Derivation and analysis of the aerial leg path/curve

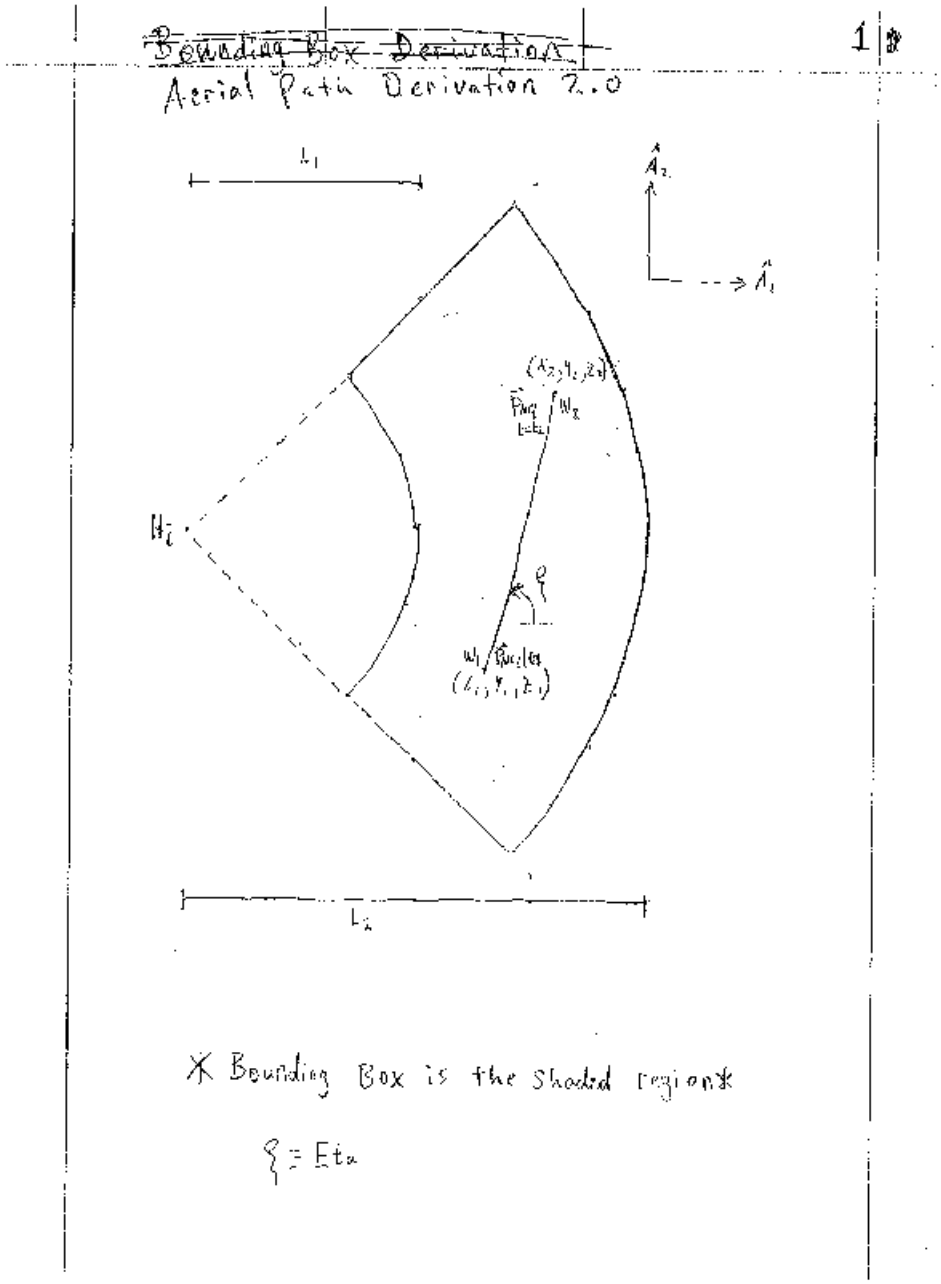


Figure 28. Diagram showing interplay between the bounding box and the potential path of the aerial leg's cycloid curve.

A cycloid is a curve that is generated by following the path of a point attached to a circle as the circle rolls across the ground. ("Cycloid," n.d.) If for no other reason, this is advantageous for walking because the cycloid allows the foot to transition between rapid aerial movement to contacting the terrain so that the no slip constraint can be met. The parameterized components of the cycloid also behave very nicely in terms of velocity, acceleration, jerk, and jolt due to their

sinusoidal nature. Theoretically, there are no rough transitions or spikes between the derivatives when the velocity of the generating circle is constant.
The canonical equations for a cycloid are

$$x = r(t - \sin(t))$$
$$y = r(1 - \cos(t))$$

Where r is the radius of the generating circle, t is the angle through which the generating circle is rotating through at some instance, x is the x-coordinate of the cycloid for a given angle t, and y is the y-coordinate of the cycloid for a given angle t. Note that in the canonical formulas, t represents an angle, not time. In order to correctly apply the cycloid to the hexapod gait, the following modifications have been made to the canonical equations:

1. The variable t now represents time.
2. The lowercase Greek letter eta (η) now represents the angle through which the generating circle is rotating through at some instance.
3. $\overrightarrow{\mathcal{W}}(t)$ is the equivalent horizontal component of the cycloid.
4. $\vec{\mathcal{V}}(t)$ is the equivalent vertical component of the cycloid.

The derivation begins with finding the amount of time that the aerial leg has to move from its lifting point to its landing point. This time interval is a function of how much time remains before the other legs on the ground reach the edge of their respective bounding boxes. It is assumed that t_{lift} corresponds to the moment in time that the soon-to-be aerial leg has reached the edge of its bounding box. In order to calculate t_{land}, the moment in time that the aerial leg reestablishes contact with the terrain, all one needs to do is solve

$$\overrightarrow{B_i}(x, y, z) = \int_{t_{lift}}^{t_{b_i}} {}^{A}\frac{d\vec{P}_{H_iA_i}}{dt}\,dt$$

for t_{b_i}, where the index i corresponds to the i^{th} leg in contact with the ground; ${}^{A}\frac{d\vec{P}_{H_iA_i}}{dt}$ is the time derivative of the position vector from a given hip to its associated foot relative to the main robot body, i.e. with respect to Frame A; t_{b_i} is the moment in time where a given leg on the ground will be at the edge of its bounding box; and $\overrightarrow{B_i}(x, y, z)$ is the function that defines the aforementioned bounding box. It may be easier in practice to solve the general form of this equation numerically rather than analytically, but the validity of that statement remains to be seen.
The minimum value of $\Delta t_i = \left(t_{b_i} - t_{lift}\right)$ will be the amount of time that the aerial leg is given to travel through the cycloid curve. If a different value of t_{b_i} is selected, then the robot is guaranteed to risk damaging itself. Thus, $t_{land} = \min(t_{b_i})$.
The above process can be greatly simplified if the velocity of the robot body is also simplified. If the robot's velocity is constant and is always pointing in a direction parallel to one of the vectors composing the frame attached to the robot body, then the integral in

$$\overrightarrow{B_i}(x, y, z) = \int_{t_{lift}}^{t_{b_i}} {}^{A}\frac{d\vec{P}_{H_iA_i}}{dt}\,dt$$

Is reduced to the equation for a line. This is a much easier problem to solve than the general case, though it may still be challenging to solve depending on how pathological $\overrightarrow{B_i}(x, y, z)$ is.

The simplest variable in the canonical cycloid equation to define is η. Linear interpolation, with respect to time, will be used to calculate the values of η because the derivative of a linear function is constant. Having a constant $\frac{d\eta}{dt}$ greatly simplifies the aerial leg path derivation. η is a function of time only, thus the interpolating variable will be time.
The cycloid curve will start with an initial angle $\eta = 0$ radians and a final angle of $\eta = 2\pi$ radians. The known values for interpolation will be t_{lift} and t_{land} as calculated previously. Thus, the formula for η is

$$\eta = 2\pi\left(\frac{t - t_{lift}}{t_{land} - t_{lift}}\right)$$

The next task is to derive a relationship between the radius, r, of the cycloid and the kinematics of the robot. First, define the vector

$$\vec{w} = \vec{P}_{ND_i}^{land} - \vec{P}_{ND_i}^{lift}$$

Note that magnitude of $\vec{w}$ is the distance between the two cusps of the cycloid. This can be expressed as

$$\|\vec{w}\| = \left\|\vec{P}_{ND_i}^{land} - \vec{P}_{ND_i}^{lift}\right\|$$

From the definition of the cycloid, the distance between two cusps is the circumference of the generating circle. This implies that $2\pi r = \|\vec{w}\|$. Rearranging terms gives

$$r = \frac{\|\vec{w}\|}{2\pi}$$

These are the last two components needed to derive the equation for $\vec{\mathcal{W}}(t)$, the equivalent horizontal component of the cycloid. Note that $\frac{\vec{w}}{\|\vec{w}\|}$ is a unit vector that is parallel to the equivalent horizontal component. It changes the orientation of the cycloid while leaving the magnitude untouched. The derivation is as follows:

$$\vec{\mathcal{W}}(t) = \frac{\vec{w}}{\|\vec{w}\|} r(\eta - \sin(\eta))$$

$$\vec{\mathcal{W}}(t) = \frac{\vec{w}}{\|\vec{w}\|}\frac{\|\vec{w}\|}{2\pi}(\eta - \sin(\eta))$$

$$\vec{\mathcal{W}}(t) = \frac{\vec{w}}{2\pi}(\eta - \sin(\eta))$$

Defining the equivalent vertical component, $\vec{\mathcal{V}}(t)$, is more involved than defining $\vec{\mathcal{W}}(t)$. The fact that $\vec{\mathcal{V}}(t)$ must be orthogonal $\vec{P}_{ND_i}^{land}$ and $\vec{P}_{ND_i}^{lift}$ is a direct consequence of how $\vec{\mathcal{W}}(t)$ is defined. The simplest way to generate a perpendicular vector is with the cross product. The following definition for $\vec{\mathcal{V}}(t)$ ensures that it is both perpendicular to the lift and landing points, and that it will also point in a direction away from the terrain and toward the robot body. The robot will risk serious damage if the vertical displacement of the cycloid moves into the ground instead of away from it.
Define the vector $\vec{v}$ such that

$$\vec{v} = \begin{cases} \vec{w} \times \vec{P}_{ND_i}^{lift} & , \quad \left(\vec{w} \times \vec{P}_{ND_i}^{lift}\right) \cdot \hat{A}_3 > 0 \\ -\vec{w} \times \vec{P}_{ND_i}^{lift} & , \quad \left(\vec{w} \times \vec{P}_{ND_i}^{lift}\right) \cdot \hat{A}_3 < 0 \end{cases}$$

The derivation of $\vec{\mathcal{V}}(t)$ follows a similar pattern to the derivation of $\vec{\mathcal{W}}(t)$. Note that $\frac{\hat{v}}{\|\hat{v}\|}$ is a unit vector pointing in the equivalent vertical direction. Finally, define the equivalent vertical component of the cycloid to be

$$\vec{\mathcal{V}}(t) = \frac{\hat{v}}{\|\hat{v}\|} r(1 - \cos(\eta))$$

$$\vec{\mathcal{V}}(t) = \frac{\hat{v}}{\|\hat{v}\|} \frac{\|\vec{w}\|}{2\pi} (1 - \cos(\eta))$$

Equation # here it is

Though the path of the cycloid relative to the robot may necessarily not lie in a plane, it will always lie in a plane with respect to the observer in this analysis because $\vec{w}$ and $\hat{v}$ are assumed to be static, unchanging quantities. If the robot receives commands directly through a computer or from a joystick, then $\vec{P}_{ND_i}^{land}$ will no longer be constant. It is recommended that a computer algebra system be used to manipulate $\vec{\mathcal{W}}(t)$ and $\vec{\mathcal{V}}(t)$ and their derivatives.

Derivation of misc. identities

Derive the identity: $\nabla(\vec{u} \cdot \vec{v}) = (\nabla \vec{u}^T)\vec{v} + (\nabla \vec{v}^T)\vec{u}$, where $\nabla = \begin{bmatrix} \frac{\partial}{\partial x} & \frac{\partial}{\partial y} & \frac{\partial}{\partial z} \end{bmatrix}^T = [\partial_x \quad \partial_y \quad \partial_z]^T$

$\nabla(\vec{u} \cdot \vec{v})$	$=$	$\begin{bmatrix} \partial_x \\ \partial_y \\ \partial_z \end{bmatrix} (\vec{u} \cdot \vec{v})$	Definition of gradient
	$=$	$\begin{bmatrix} \partial_x(\vec{u} \cdot \vec{v}) \\ \partial_y(\vec{u} \cdot \vec{v}) \\ \partial_z(\vec{u} \cdot \vec{v}) \end{bmatrix}$	Definition of gradient
	$=$	$\begin{bmatrix} (\partial_x \vec{u}) \cdot \vec{v} + (\partial_x \vec{v}) \cdot \vec{u} \\ (\partial_y \vec{u}) \cdot \vec{v} + (\partial_y \vec{v}) \cdot \vec{u} \\ (\partial_z \vec{u}) \cdot \vec{v} + (\partial_z \vec{v}) \cdot \vec{u} \end{bmatrix}$	Product rule for dot product
	$=$	$\begin{bmatrix} (\partial_x \vec{u}) \cdot \vec{v} \\ (\partial_y \vec{u}) \cdot \vec{v} \\ (\partial_z \vec{u}) \cdot \vec{v} \end{bmatrix} + \begin{bmatrix} (\partial_x \vec{v}) \cdot \vec{u} \\ (\partial_y \vec{v}) \cdot \vec{u} \\ (\partial_z \vec{v}) \cdot \vec{u} \end{bmatrix}$	Separate terms
	$=$	$\begin{bmatrix} (\partial_x \vec{u})^T \vec{v} \\ (\partial_y \vec{u})^T \vec{v} \\ (\partial_z \vec{u})^T \vec{v} \end{bmatrix} + \begin{bmatrix} (\partial_x \vec{v})^T \vec{u} \\ (\partial_y \vec{v})^T \vec{u} \\ (\partial_z \vec{v})^T \vec{u} \end{bmatrix}$	Transform the problem using $\vec{a} \cdot \vec{b} = \vec{a}^T \vec{b}$

	$=$	$\begin{bmatrix}(\partial_x \vec{u})^T \\ (\partial_y \vec{u})^T \\ (\partial_z \vec{u})^T\end{bmatrix}\vec{v} + \begin{bmatrix}(\partial_x \vec{v})^T \\ (\partial_y \vec{v})^T \\ (\partial_z \vec{v})^T\end{bmatrix}\vec{u}$	Factor out $\vec{u}$ and $\vec{v}$
	$=$	$\begin{bmatrix}\partial_x(\vec{u}^T) \\ \partial_y(\vec{u}^T) \\ \partial_z(\vec{u}^T)\end{bmatrix}\vec{v} + \begin{bmatrix}\partial_x(\vec{v}^T) \\ \partial_y(\vec{v}^T) \\ \partial_z(\vec{v}^T)\end{bmatrix}\vec{u}$	Factor out the partial derivatives
	$=$	$\left(\begin{bmatrix}\partial_x \\ \partial_y \\ \partial_z\end{bmatrix}\vec{u}^T\right)\vec{v} + \left(\begin{bmatrix}\partial_x \\ \partial_y \\ \partial_z\end{bmatrix}\vec{v}^T\right)\vec{u}$	Decompose the partial derivative matrices into original factors
$\nabla(\vec{u}\cdot\vec{v})$	$=$	$(\nabla\vec{u}^T)\vec{v} + (\nabla\vec{v}^T)\vec{u}$	Simplify the partial derivative vectors. The derivation is now complete