NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF MECHANICAL ENGINEERING
SECTION OF MECHANICAL DESIGN & AUTOMATIC CONTROL
Control Systems Lab

Diploma Thesis

DESIGN AND ROS-BASED CONTROL OF AN ARTICULATED MONOPOD AND ITS EXPERIMENTAL SETUP

Spyridon Garyfallidis

Supervisor: E. G. Papadopoulos

ATHENS 2017

Abstract

Robotics field, and especially the one of articulated biomimetic robots, is one of the most rapidly developing scientific fields, in which one is always required to test new ideas. For that purpose, specialized setups are created where the designer can individually test various leg configurations but also different control systems or controllers. Those setups are called monopod robots and this thesis regards the one installed in our laboratory.

Since progress on the field is fast, both in the hardware and the software subcategories, there is a need to adapt in recent developments equally fast.

This of course requires swift and easy renewal and replacement of structural elements, parts and codes, standardization of the most frequently used and fundamental functions, as well as their decoupling from each programmer and setup, leading to interchangeability between, for example, different controllers, but also their recycling and reusal in different versions of the robot. This task is to be carried out by the upcoming software platform ROS, which was incorporated both in the monopod and the lab's treadmill.

After the control system's creation, there comes a need to somehow test its function, the tools it offers and its various parameters, since the most popular simulation tools, like Matlab, are not ideal for this purpose. This problem is resolved with the introduction of Gazebo, a simulation software with the capability to interact with ROS, but also with various secondary incorporated useful tools and libraries.

For the execution of simple tasks, like the position control of a motor, and considering what we mentioned before about standardizing procedures, there was an evaluation of the available options and resources that ROS provides, and specifically the *ros_pid* and *ros_control* packages. There is also a brief reference of the path planning capabilities that are available.

Finally, in the frame of the general experimental leg testing setup renewal, there was a transition from the first leg design, based on the classic SLIP model with one revolute hip joint and a prismatic knee, to a second biomimetic articulated leg with two revolute joints, according to a new innovative method conceived in our lab. In this thesis we will provide some basic mechanical design concepts and ideas.

Περίληψη

Ο τομέας της ρομποτικής, και πιο συγκεκριμένα των αρθρωτών βιομιμητικών ρομπότ είναι ένας εκ των ταχύτερα αναπτυσσόμενων, στον οποίο κανείς είναι υποχρεωμένος συνεχώς να δοκιμάζει νέες ιδέες. Για το σκοπό αυτό δημιουργούνται διατάξεις στις οποίες ο σχεδιαστής μπορεί να δοκιμάσει μεμονωμένα διάφορες μορφές ποδιών αλλά και διαφορετικά συστήματα ελέγχου. Οι διατάξεις αυτές ονομάζονται μονόποδα ρομπότ και η εργασία αυτή αφορά το μονόποδο το οποίο είναι εγκατεστημένο στο εργαστήριό μας.

Εφόσον η πρόοδος στο χώρο είναι αλματώδης, τόσο στον τομέα των υλικοτεχνικών διατάξεων (hardware) όσο και λογισμικού (software), δημιουργείται άμεσα η ανάγκη γρήγορης προσαρμογής στις διάφορες εξελίξεις.

Αυτό φυσικά προϋποθέτει γρήγορη και εύκολη ανανέωση και αντικατάσταση δομικών στοιχείων και κωδίκων, τυποποίηση των πιο συνηθισμένων και βασικών διαδικασιών και απεμπλοκή τους από τον εκάστοτε προγραμματιστή και εγκατάσταση, οδηγώντας σε εναλλαξιμότητα μεταξύ, για παράδειγμα, διαφορετικών ελεγκτών, αλλά και επαναχρησιμοποίησή τους σε διαφορετικές εκδοχές και εκδόσεις της εγκατάστασης - ρομπότ. Το έργο αυτό έρχεται να επιτελέσει η ανερχόμενη πλατφόρμα ανάπτυξης λογισμικού ROS, η οποία ενσωματώνεται τόσο στο μονόποδο ρομπότ όσο και στον κυλιόμενο διάδρομο του εργαστηρίου.

Μετά την δημιουργία ενός συστήματος ελέγχου στο ROS, δημιουργείται κατευθείαν η ανάγκη αυτό να δοκιμαστεί, καθώς και τα εργαλεία που προσφέρει και οι διάφοροι παράμετροί του. Τα συνήθη προγράμματα που χρησιμοποιούνται στην πλειοψηφία των περιπτώσεων, π.χ. Matlab, δεν προσφέρονται για αυτό το σκοπό. Το πρόβλημα έρχεται να λύσει το λογισμικό προσομοίωσης Gazebo, με δυνατότητα ενσωμάτωσης και συνεργασίας με το ROS, αλλά και με διάφορες χρήσιμες βοηθητικές λειτουργίες και βιβλιοθήκες.

Για την επιτέλεση βασικών εργασιών, όπως για παράδειγμα τον έλεγχο θέσης ενός κινητήρα, και με βάση αυτά που αναφέραμε πριν για την τυποποίηση εργασιών, γίνεται διερεύνηση των διαθέσιμων επιλογών και δυνατοτήτων που μας παρέχει το ROS, και πιο συγκεκριμένα τα πακέτα *ros_control* και *ros_pid*. Γίνεται επίσης μια σύντομη αναφορά στη δυνατότητα για ενσωμάτωση σχεδιασμού τροχιάς.

Τέλος, στα πλαίσια της γενικότερης ανανέωσης της πειραματικής διάταξης δοκιμής ποδιών, περνάμε στο εργαστήριο από το πρώτο στάδιο ποδιού βασισμένο στο κλασικό SLIP μοντέλο, μίας περιστροφικής άρθρωσης στο γοφό και πρισματικού γονάτου, σε ένα περισσότερο βιομιμητικό αρθρωτό πόδι δύο περιστροφικών αρθρώσεων, σύμφωνα με νέα καινοτόμο μέθοδο υπολογισμού προερχόμενη από το εργαστήριό μας. Στα πλαίσια αυτής της εργασίας θα δοθούν κάποια στοιχεία για τον μηχανολογικό σχεδιασμό και διάταξη.

To my family

Preface – Acknowledgements

For anyone relative to the field of legged robots, it is clear that a monopod robot can be used as the basic test unit for a multi legged robot, e.g. a quadruped. It is a very reliable and safe way to test leg properties, various leg configurations and control algorithms, as there is no need for the researcher to deal with leg coordination and body balance. One can instead focus on dynamics and control issues.

To this day, there have been two monopod versions in our lab. The first, featuring a PC-104 and Linux OS, managed to function properly [8]and a dynamic hopping experiment was conducted. In the second updated version the PC-104 was replaced by a Beaglebone and a new set of electronics [32]. Unfortunately that second version never operated the way it should have. That was partly because after the first successful attempt, same codes for same functions had to be rewritten for the new computational system and the work that was actually recycled was next to zero. Same holds for simulations and models that were created for both versions. Therefore, the actual purpose was to start building on a new solid basis, upon which other researchers and lab members could continue building, add features, utilize them on other applications and generally benefit from, and not rediscover the wheel every single time on the beginning of a new project.

To this end, a software platform that can provide said basis was tested and incorporated on the experimental setup. Robot Operating System (ROS) is a generic platform that offers various libraries and tools that promote code distribution, standardization and recycle. This software is rapidly gaining ground in terms of popularity, so its utilization was somewhat imposed. However there still exists a necessity to test the control systems we create in ROS, but also to simulate the physical setup. Gazebo, the simulation platform in which our system was modeled and tested, allows for simulating the robot's behavior using the exact same control system that will be used on the actual hardware, making the transition between simulation and experiment as easy as the push of a button. It is exactly this capability to connect with ROS that led to its utilization.

Finally, after properly redesigning the software aspect, and where it was required the electronics, a new articulated biomimetic leg was designed. Segment lengths will be defined later using the new leg design method for performance running. As it is a work in progress, a diploma thesis of another lab member, we shall not mention any details and we will deal with a modular joint design instead.

At this point I would like to thank professor E. Papadopoulos for the guidance throughout this whole process, but also for his trust and the opportunity he offered me to work on the highest level possible. Next, everyone else that contributed to this work. PhD candidates K.

Koutsoukis and K. Machairas for the general support, advice, useful ideas and most of all patience, Phd candidate T. Mastrogeorgiou for his assistance in software debugging and K. Assimakopoulos for the support in electronics. Finally, every member of the legged lab for making those long hours seem less long and tiresome with the amazing atmosphere they created.

Contents

List of Figures

List of Tables

List of Symbols

QEI	Quadrature Encoder Interface
CPR	Counts Per Revolution
SLIP	Spring Loaded Inverted Pendulum
PWM	Pulse Width Modulation
eQEP	enhanced Quadrature Encoder Pulse
DAC	Digital to Analog Converter
ERP	Error Reduction Parameter
CFM	Constraint Force Mixing
LR	Loop Rate
MB	Mounting Board
PDB	Power Distribution Board
RB	Regulator Board

1 Introduction

1.1 Motivation and objectives

The field of legged robotics is only showing recent development, expanding on the last four decades. Our demands increase as technology advances. For example, no one could have considered planet exploration using a legged robot a viable option twenty years ago, although it could have been a distant objective. Wheeled robots were, and still are, simple to build and easy to control, and as such they represented the only option for mobile robot applications, no matter what their deficiencies were, e.g. limited ability to move on rough terrains and slopes, overcome obstacles. Such areas are hardly accessible by wheeled robots, as there is a large possibility for them to get stuck, damage their-perhaps sensible- surroundings or even damage themselves.

On the other hand, a legged robot can selectively define its gait and its foot placement to maintain its balance, avoid sensitive or dangerous terrain areas, jump over obstacles or holes and even develop high velocities, should that be required. Also the torso's motion is uncoupled of the leg motion, in contrast with a simple wheeled robot. This could prove to be useful in cases of movement in uneven terrain, where the cargo, perhaps a sensory system or a camera, has to remain steady while the robot is moving.

The above however, remained a distant dream, up until recently when the accumulated know-how started showing reliable results. In any case, and since challenges and requirements continuously arise, new concepts and ideas need to be tested and evaluated constantly. Of course these experiments cannot be exacted on a functional robot or with all the legs installed. For example, constructing four experimental legs and testing them in an actual quadruped would be extremely costly, in terms both of time and money. To this end, experimental setups are created where the researcher can modify a single leg, hence the name monopod robots, and test mechanical properties, different configurations and control algorithms, undistracted by coordination and balancing issues.

This thesis' main purpose is to update the monopod robot of the CSL with the latest development in software, the Robot Operating System (ROS). ROS is a platform for software development that promotes code recycling from version to version, standardization of processes and, most importantly, code sharing between different developers who share this common basis.

To test and simulate the behavior of the system, the developer needs not only to solve the usual dynamic equations but integrate elements such as communication between different components, sampling rates, etc. For these purposes, the simulation software platform Gazebo is preferred. A model of the monopod robot was created and the same control system that would be applied to the real robot. This way the total system is simulated instead of just

the dynamic behavior, without having to derive any dynamic equations, by defining only the geometry and physical properties.

Finally, in the context of total renovation of the experimental setup, and in order to move towards a more biomimetic design, a new articulated leg had to be designed. This leg, with revolute hip and knee joints, resembles nature and can realize the design theory developed in our lab concerning the optimization of segment lengths for various gaits or tasks.

1.2 Literature review

As was already mentioned, monopod robots are the most fundamental testing units for legs and control algorithms that will later be adapted in multi legged robots. This is why there are many recorded cases of such robots, both during the first years of legged robotics field development, but also recently.

While browsing through the literature, what is really noticeable is the significant quantitative difference between the several control strategies, algorithms and methods, and the number of successful experiments with those techniques as well as the number of functional monopods. Whereas there are many different control theories (e.g. [25], [5], [27]), including our own lab's work ([7], [31]) on controlling a 3 DOF monopod and its motion on rough compliant terrains, no experimental confirmation is mentioned. Of course those concepts are quite hard to implement themselves; it only gets more difficult when software and hardware debugging issues are included. The existence of a fully functional, efficient and reliant experiment platform is priceless for such purposes.

There are of course some examples of such functional setups. MIT's 3D hopper machine with a prismatic actuated knee joint [23] based on the SLIP model, is one of the most important; the ARL Monopod versions I and II [1] being quite similar. This model considers the leg as an inverted pendulum while the foot is in contact with the ground (stance phase). Based on this model, a large variety of controllers is developed; also quadruped robots featuring such legs ([24], [8]). After a while, leg design leaned towards more biomimetic shapes, like the MIT's Monopod [43]. This robot uses a boom link to connect to a fixed base, a method that we also employed for our lab's first single-legged robot [8]. Other examples include the Kenken single-legged hopping robot [14], which is hydraulically actuated and articulated, the Uniroo [33], motivated by kangaroo's locomotion, the JennaHopper [26], and the leg of the biped Sugoi-Neco, described in [19] and tested in [28]. Moving towards more recent attempts, the OSU's ATRIAS biped has its own monopod design. For its successor Cassie, by Agility Robotics [44], there also seems to be a one-legged module on an introductory video, although the project is still new and few details are available. We shall discuss this leg design and a few others in Chapter 5.

1.3 Thesis outline

This thesis is comprised of six chapters. In the first introductory chapter, a brief discussion is presented on the motivation and the target of this work. There is also a presentation of the various leg experimental setups available in the literature. Finally the thesis' structure is presented.

Chapter 2 introduces the various software platforms that were used. At first, we present ROS, along with its advantages. A description is given of its basic tools and structure, as well as a mention to some of its more advanced capabilities. The simulation software Gazebo is also presented and described. There is also a mention of the path planning software Moveit!.

In Chapter 3, our lab's monopod robot is presented. Its previous design, as well as the problems associated with it are mentioned. Then we proceed with the work that was done, including the redesign of the electronic subsystem, the design of the ROS nodes, as well as the modeling in Gazebo and the simulations. Finally, the experiments that were conducted to verify the setup's functionality are described. For those experiments, we describe and define the most fundamental experimental and simulation parameters.

In Chapter 4, we repeat the work that was done for the monopod for the case of the lab's treadmill. Redesign of the whole electronic subsystem, ROS structure and velocity control experiments are described in detail.

In Chapter 5, the various leg designs of many known quadruped robots are reviewed, and an initial attempt to realize the leg segment optimization theory is being made. A first design of the modular knee and hip joints is analytically presented and tested.

Finally in Chapter 6, we summarize the results and assess the work done. We also offer some ideas for potential future work.

2 ROS, Gazebo and MoveIt!

2.1 Introduction to ROS

Ever since the creation of the first robotic systems, there was a big discussion concerning the software platform that should be employed. At first, everybody was using their own custom software and electronics, which largely complicated the software design. Codes that would be created and used for one specific combination of central processor and motor drive, for example, could not be transferred easily and reliably to the next design version, that might have a different combination of said parts. As a result the code had to be rewritten, taking into consideration the new hardware and its characteristics, something that was very time consuming. And all that because of the lack of a more general platform that different components would just connect and communicate with each other.

After a while, it was decided that this could continue no more, and the Robot Operating System started developing. ROS is a platform that includes a large database of drives for many devices and sensors, and allows easy cross communication with each other. This removes the need for codes to handle communication issues, which is extremely time consuming as mentioned earlier. The database is updated continuously with new elements allowing for fast incorporation of a new part in an existing system. Also common processes, such as position control of a motor are included and no longer have to be rewritten, but rather only properly configured from one application to another.

Another advantage of ROS is that it is running under Linux, usually Ubuntu. This makes it easily transportable between devices that can run Linux, like Raspberry Pi, Beagleboard and PC-104. Again, there is no need for rewriting or even modifying codes to fit the new environment. Also there are tools included in order to distribute conveniently the computational load by allocating executables in different devices; ROS handles their communication. This would be really useful in case, for example, someone wanted to build a Raspberry Pi stack instead of using an expensive PC-104, or run the PID control loops on the robot and the rest user interface on a central control station.

To be more precise, ROS' main features include [35]:

I. Friendly and easy to use **messaging system**. This is one of the first challenges a developer faces while designing a new robot. ROS provides a built-in and tested messaging system that proves to be really handy, as it manages all the communication's small details relieving the developer from the need to set up communication protocols, define and refine data exchange rates and of course, the debugging of that system. Specifically, the system that ROS is providing is an anonymous, asynchronous publish/subscribe mechanism where there is a

publisher and a subscriber node (ROS' form of executable files, written either in C++ or Python). The first one publishes information on a named bus over which nodes exchange messages called topic, and the second subscribes and receives it as published. What is interesting is that the publisher node is not aware who is receiving the messages that it is publishing, all it does is just use a specific type of message. After that, a topic can have multiple subscribers, practically every node that is interested or requires the relevant information. All they have to do is subscribe using the same type of message, removing the requirement to set up different communication channels between distributed devices to exchange a type of information that everyone needs. This goes both ways, as it is possible for more than one nodes to publish information on a topic. Another direct byproduct of this system is that it enforces developers to implement clear nodes interfaces, leading to less complex and readable algorithms, and also promoting code reuse.

II. **Services**. The aforementioned communication system regards one way message exchange as, like we mentioned, there is no response when publishing a message or a report about whether it was received and accordingly processed. However in many applications there is a necessity to send a message to a process, e.g. to execute a task, and receive a reply to know if the execution was successful or not. The service performs exactly that function. It uses the same messages as before, only now there are two, one for the request and one for the reply. Specifically a node offers a service, a second node sends to it a request to activate it, and then awaits for a reply.

III. **Message monitoring**. The publish/subscribe messaging system allows for easy real time monitoring using plots (*rqt* tool), if the data is plottable, or simply screen printing of published data in case of more complex messages or strings. The data can also be easily recorded for further processing in another software, e.g. Matlab, using the *rosbag* tool. We can use rosbag to record messages in a rosbag file from any topic of our choice, for as much time as we want. This file can be replayed to reproduce the result. For example, let's assume we have two nodes, A and B. Node A produces series of messages published on topic C, to which Node B subscribes and reads. If we record topic C while A publishes, we can then shut down A, replay the rosbag file and reproduce B's response. In other words, Node B does not realize the difference between data coming from A or a rosbag file. This is of course a result of the communication structure, as in both cases, all that node B sees is a topic in which it has to subscribe, without caring about its origin.

IV. **Global Parameter Server**. In most cases, ROS provides a convenient library of configuration values available for all nodes at runtime. This way all nodes can

have access and modify the system's state. It is advised to be used only for static, non-binary data as it is not particularly fast or efficient.

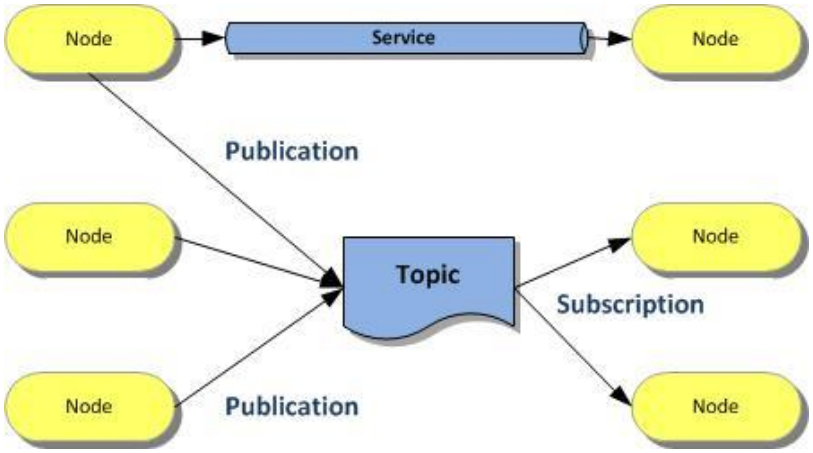


Figure 2-1. ROS messaging and service structure.

Those are only a few and most basic features ROS has to offer. Other, more advanced options include:

- Standard robot messages, like poses, transforms, various vectors, IMU sensors, lasers, etc. It also includes message definitions for navigation applications, like odometry, paths and maps.
- Robot description language. In some packages, like *ros_control*, a robot description is required in a format compatible with ROS. Of course ROS is providing a format called Unified Robot Description Format (URDF), i.e. an XML file in which the user defines the geometry (links, joints) and their properties (masses, inertias), but also visuals and sensors. This format can be used to visualize the robot and its motion in Rviz but also to simulate a controller in Gazebo.
- Robot geometry library. In cases of robots with many links and joints it is useful, although not easy, to know every part's position with respect to each other. This is most significant in cases where there are many sensors and we need a common reference frame. For those purposes, ROS provides the *tf* (transform) library to allow for easy transformation of sensor data from one system to another.
- Actions, an improved version of services. While with services one could request the execution of a task and receive a response once it was completed, with actions the user can actually monitor the task while it is being executed. For example, we could command a robot to move from point A to point B, monitor its whole motion and even redirect it if necessary.

- Diagnostic tools for messages, command-line tools and convenient visualization tools (*Rviz* and *rqt*) perform several tasks and facilitate the debugging process.

Last but not least, ROS is providing some limited, for now, real time tools. Real time is becoming steadily a trend nowadays, and ROS is starting to incorporate some features, like a real time publisher. Note that the standard version of ROS is not real time, as it is based on Linux, which is not a real time operating system either. Therefore, there is no guarantee that processes will be timed and executed as required. This fact creates issues in high frequency control applications, and subsequently real time control is required.

Examples of ROS' widespread utilization include from various hobby, homemade robots (especially using Raspberry Pi) to industrial applications and research projects. For example in our laboratory, both the quadruped robot Laelaps [47] and the space robot Cepheus [48] are using ROS for motor control, path planning and navigation purposes. Specifically modified to be used in industry applications comes *ROS-Industrial*, with interfaces for commonly used end effectors, grippers, etc. There are also several software libraries for path planning and sensor calibration.

Lately, there has also been an attempt to standardize robot components, like sensors, actuators, processors, etc, in a way that would be ROS compatible and easily interchangeable; even if made by different manufacturers. That would reduce the development of a new robot configuration into a simple plug and play process, where parts would just connect and handle their low level communication and cooperation, leaving only the high level programming and task assignment to the human operator. Damage repair and hardware debugging, processes as time consuming and as hard as the software debugging itself, would greatly simplify, also. This effort was named *H-ROS* [36] and it is currently on its very beginning, to be seen how and if it will be embraced by the community.

In the corresponding page [49] there are over 120 robot cases listed which use ROS. The majority regards wheeled platforms and manipulators. There are also some legged robots, mostly bipeds and grippers, human-like hands, and a few quadcopters.

Finally we should mention the various ROS versions. When we started working on this project, the most stable ROS version was Indigo, paired with Gazebo 2. After a while, ROS versions Jade (with Gazebo 5) and Kinetic (with Gazebo 7) were released. Since we seek continuous support and maintenance for all packages, and Gazebo 7 proved to be more convenient, we decided to proceed with the latest version, Kinetic, the tenth ROS release with end of life estimated date in May, 2021. Jade was released on an odd numbered year (2015), and those releases are only supported for two years instead of five.

2.2 Gazebo simulation software

Gazebo is a simulation software for dynamic systems that specializes in robotic systems' simulation, providing a large variety of useful tools concerning robot modelling [37]. Its close collaboration and relation with ROS has established it as the most utilized simulation software, with V-Rep, Webots, ADAMS, etc. following [30].

Gazebo is offering multiple advantages in robot simulation. It supports all four major physics engines, *ODE, Bullet, Simbody* and *DART*. It has enhanced visualizing and rendering capabilities, utilizing OGRE, an open-source graphic rendering engine. It also allows a user to run simulations on remote servers, using TCP/IP transport. However, probably the biggest advantage, just like in the case of ROS, is the large database of robot and sensor models that are provided. Cameras, lasers, Kinect, Lidar and force torque sensors are just a few examples, with the option to add noise in order to make simulation even more realistic. Same holds for numerous custom plugins that allow users to manipulate models exactly as they wish, as its open source nature implies. An additional advantage lies in the fact that in contrast with Adams and Webots, it is actually free to download, and yet there is a large, continuously growing community for support and guidance. For that purpose there is also a set of tutorials that cover most essential terms and functions.

Its simulation capabilities are impressive. It offers the opportunity even to run fluid dynamics simulations, although it has declared them outdated and advices to use them with caution. Also there are hydrodynamic and aerodynamic plugins to simulate the behavior of underwater and aerial objects. However most relevant to our case and indicative of the field's tendency to turn to Gazebo for reliable simulations is its excessive use in the Virtual Robotics Challenge, a part of the larger DARPA (Defense Advanced Research Projects Agency) Robotics Challenge (DRC). The later was a prize competition funded and organized by DARPA to develop autonomous robots that would compete in completing several tasks in harsh environments. The challenge included the DRC Simulator, based on Gazebo, where each team had to perform several tasks in the same simulated environment using their controllers applied to the same simulated robot, Boston Dynamics' Atlas. Gazebo includes a set of tutorials that guide the user through the manipulation of such a complex robot, its teleoperation, as well as grasping and navigation. The Atlas robot, as well as the simulated environment are presented in Figure 2-2.

(a) (b)

Figure 2-2. (a) DRC simulated environment - property of DARPA [38]. Atlas had to get in and drive the vehicle for a certain distance avoiding obstacles. (b) Atlas anthropomorphic robot - property of Boston Dynamics.

Despite its ability to closely cooperate with ROS and the fact that they are both maintained by OSRF (Open Source Robotics Foundation), Gazebo actually is a standalone package, where users can build models and simulate their dynamic behavior autonomously. It is suggested to use Linux Ubuntu, but there has been an effort to expand its usage to include Windows. However, there is not yet full support and it is not recommended.

We previously mentioned the various plugins that are available. The one that makes Gazebo so precious and important to ROS users is the one that allows data to flow in and out of the simulation using ROS topics. For example revolute joint angle measurements or joint hip torque commands are easily received and applied respectively just by using a properly modified plugin. This of course means that the same nodes that will be utilized on the actual hardware can be directly tested in simulation, including their cross communication system. This is a great improvement from just writing a controller in Matlab and running a simulation in it, which is only an approximation, as it does not include anything like node data exchange and node loop rates, parameters that are difficult to model and test. The only thing that has to change between a Gazebo simulation and an actual experiment with the real hardware is the interface between the sensors and the actuators, for example instead of the plugin there must be a node that will receive measurements and send commands.

Gazebo has integrated design capabilities. One can insert basic geometric structures (spheres, boxes and cylinders) and define joints (prismatic, revolute, spherical, etc.). As it is clear from the description above, designing in Gazebo is quite limited. However, there is an exporter that can take Solidworks assemblies and translate them into SDF files, the primary file format Gazebo uses to define models. On this exporter we can define link chains, joints and visuals. The created models can be attached to world files. Those files contain the models we want to simulate as well as the solver definition and various solver parameters.

We selected to use the Gazebo 7 version, since it is the most recent version, pairs well with ROS Kinetic, and their end of life dates are the same.

2.3 Path planning using MoveIt!

MoveIt! is a modern path planning tool for robotic arms, wheeled vehicles and legged robots, with over than 65 recorded applications [39].

The main reason we experimented with path planning methods was the desire to maximize the proportional and derivative gains of the PID control that was eventually used on the monopod's hip joint. When path planning is used, setpoints are sent gradually until the desired final value, in contrast with the case of a step input. This allows for larger gains, since the error of each loop is smaller, therefore we can reach the motor limits, which is always a requirement, especially in such demanding applications.

MoveIt! works in a similar manner with Gazebo. One has to create a model using the URDF format. The model and its entire configuration is inserted using the incorporated MoveIt! Setup Assistant, for which there are extended tutorials and instructions. There one can define many characteristics, like kinematic chains, end effectors, virtual joints (connections with the environment), non actuated -passive- joints and even predefined configurations, like the home position for manipulators.

This package comes with a plugin for Rviz. There, the user can graphically define the initial and final desired configurations, and then either just plan and visualize motion in Rviz or plan and execute, by sending the trajectory to the controllers in Gazebo or the actual hardware.

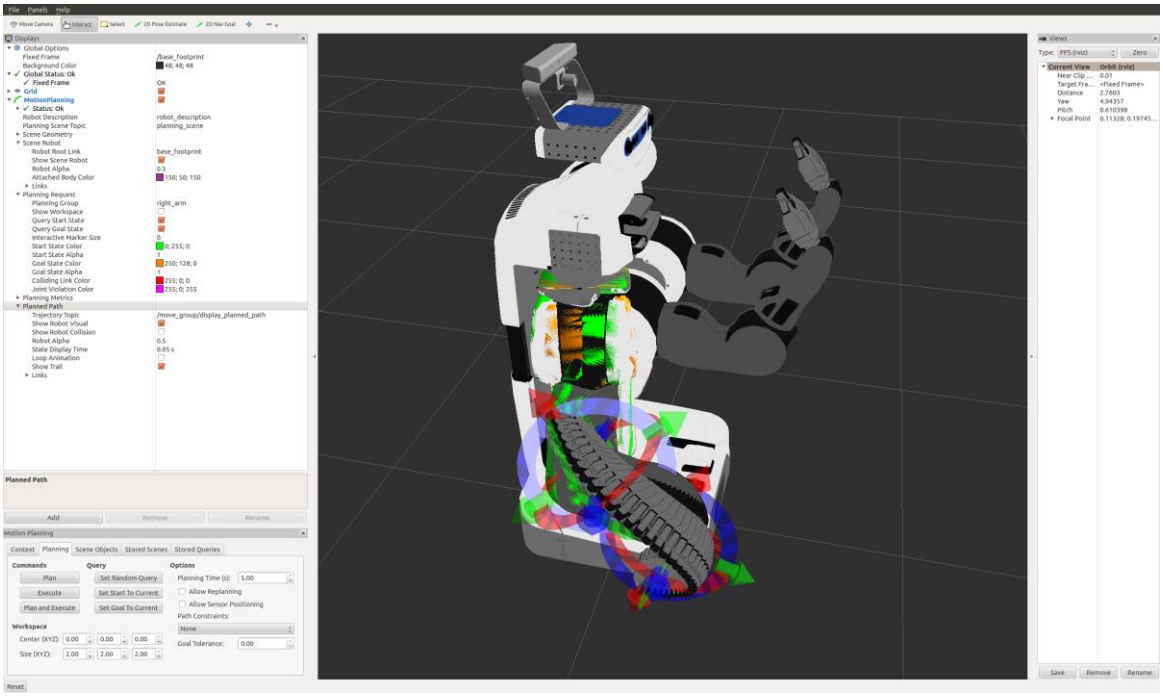


Figure 2-3. MoveIt! Rviz plugin interface of the PR2 robot.

One also has other options like obstacle detection and avoidance, self-collision detection, and checks to ensure the arm is always inside the reachable workspace.

This platform is quite new and still developing. At the time of its examination, it was not compatible with the other parts of software, ROS Kinetic and Gazebo 7, and therefore it was deemed wise not to incorporate it on the current single actuated monopod version, as it is quite simple and can be handled using standard techniques and methods.

3 Monopod design and control using ROS

3.1 Hardware setup description

The first monopod robot version that has ever been manufactured and run in our lab is the one pictured in the Figure 3-1 below. This setup included a fixed central base that would connect to the robot and allow for rotational motion around its axis. This connection with the robot was established through an aluminum bar, fixed on the robot and attached with a revolute joint on the base, thus allowing vertical hopping, additional to the previously mentioned circular motion. On the contrary, the fixed connection between the body and the bar would not allow pitch motion. That was necessary in order for the robot to be as close as possible to the 2D SLIP model mentioned previously. This setup was almost made entirely of aluminum, in order to reduce weight and inertia. However, the leg that is receiving most of the structural and impact loads is made of steel.

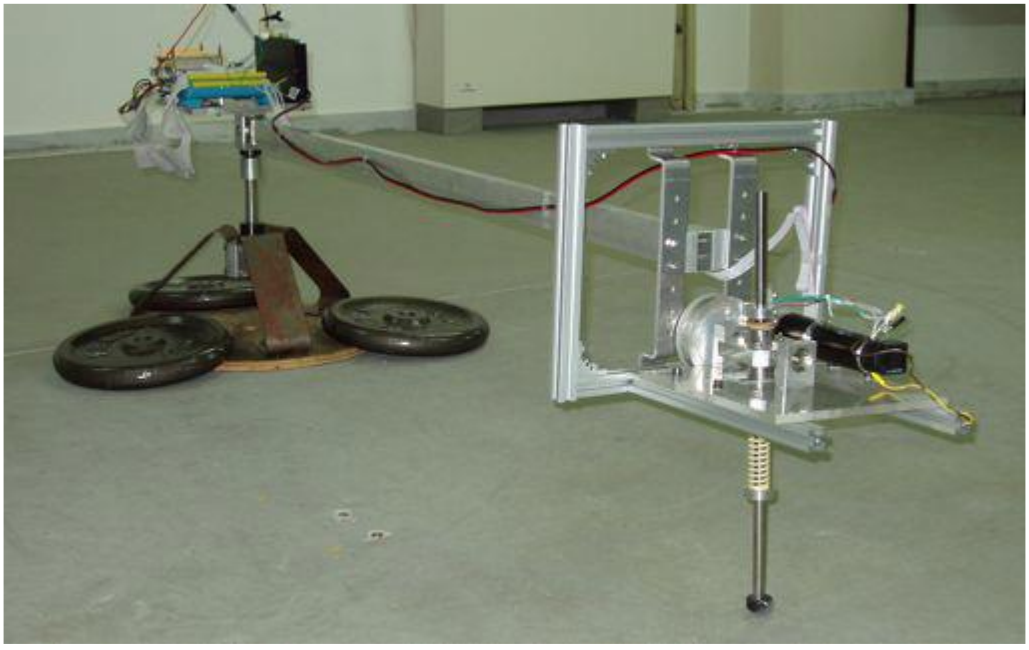


Figure 3-1. Previous experimental setup.

Thanks to the bar's large length and the relatively small size of each stride, we can easily assume that a setup like the one we are describing is appropriate to realize the monopod's motion on a 2D sagittal plane, according to the SLIP model that was mentioned earlier. One can easily find similar examples in literature. Nevertheless, no matter how long the bar is, this setup is and will remain a close approximation that was mainly used because a structure that could support the robot to run on a treadmill, like most modern legged robot applications, was not yet designed and manufactured. After the construction of such a system [22], we were able to mount the modified monopod, as we shall describe next, on the treadmill. This solution is preferred not only because it is an accurate realization of planar 2D motion but also because of the reduced amount of space that it requires.

The mounting mechanism consists of two rails and wagons in the vertical (Z) direction and a one wagon-rail in the X direction, along the treadmill. We used two rails on the Z axis, because the rail is cylindrical and it would allow for yaw rotation. The rails were mounted on aluminum bars to keep the system lightweight, pictured in Figure 3-2.

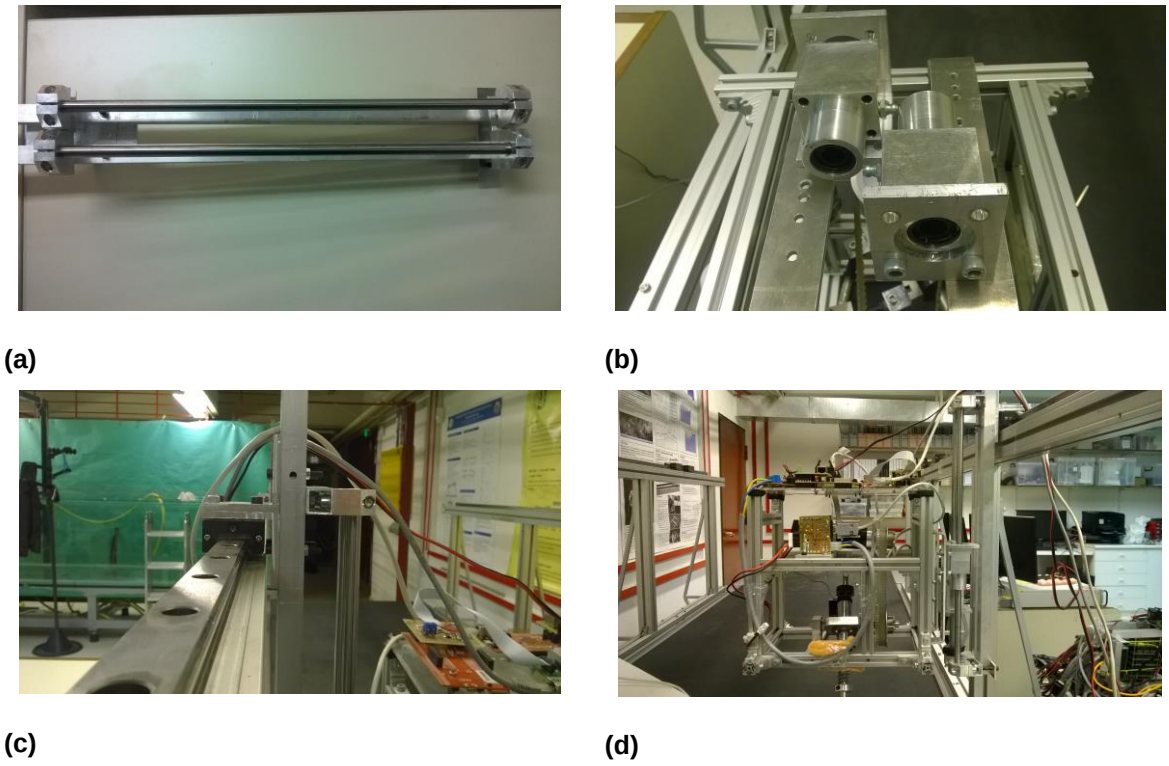


Figure 3-2. Mounting mechanism. (a)-Z axis rails, (b)-Z axis wagons, (c)-X axis rail and wagon, (d)-final assembly.

The monopod robot's hip joint is actuated by an electric DC current motor with an appropriate gearhead. The motor is made by Maxon (type RE35, 90 W, maximum continuous DC current 3.36 A, nominal voltage 24V and maximum continuous torque 0.0933 Nm). The planetary gearhead is also made by Maxon (type GP42C) and is mounted to the motor with reduction ratio equal to 26:1. Also, to transmit motion to the hip joint axis, a belt drive was employed, with reduction ratio 2:1, bringing up the total ratio to 52:1.

The motor was previously driven by a DZRALTE-012L080 amplifier made by Advanced Motion Controls (AMC). Its main advantage is its ability to provide high current in combination with its small size and weight. Software provided by AMC allows a user to program the amplifying gains, avoiding use of mechanical switches and potentiometers. It can also operate in position, velocity or current mode. However, this drive was considered as too complex for our application and thanks to massive changes occurring simultaneously at our laboratory's quadruped robot design, Laelaps, it was replaced by an AZBDC12A8 drive, made also by AMC. This drive can supply the same current as the previous one, up to 6A of continuous current and 12A intermittent (for 2 seconds), but has no need for programming. To

successfully use it, one sends an enabling signal to the corresponding pin, provides direction on the direction pin (HIGH or LOW) and sends a PWM signal, properly configured according to for the desired current. By default, 100% duty cycle corresponds to 12 A, although as mentioned this amount can only be supplied for 2 seconds (modulated by the drive itself, so there is no need to set our own software limitation to avoid drive damage). As one can easily deduce from above, in our application this drive simply functions in current mode.

Those drives are combined with mounting boards made in our lab for our quadruped robot Laelaps [4]. Since two different power levels are needed on the same drive, a low one for the control signals and a high one for the power supply, this board was designed with an optocoupler to isolate these two levels as a safety precaution. The optocoupler requires 3.3 to 5.5V of supply at both of its sides. Therefore, the board requires three ports in total:

- A low voltage and signal input to receive enable, direction and PWM signals, as well as 5V and GND for the optocoupler's first side,
- A high voltage input (24V) to supply the motor as well as 5V for the optocoupler's second side.
- A motor power supply output.

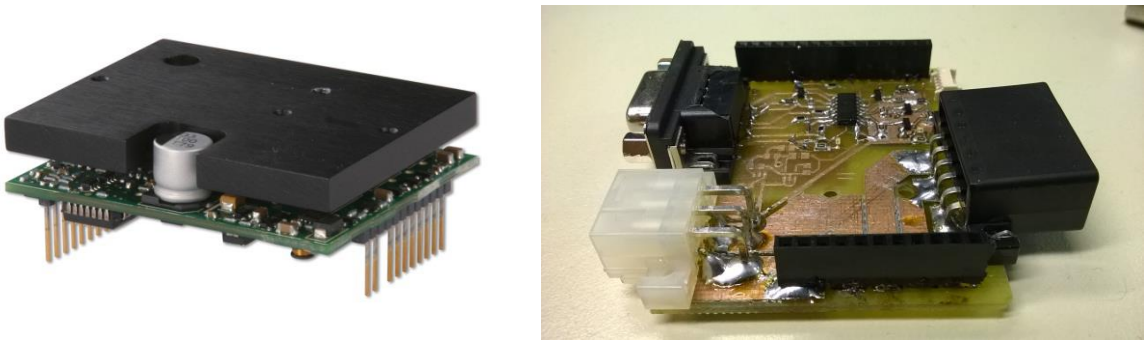


Figure 3-3. AZBDC12A8 drive and mounting board.

As far as the sensory system is concerned, there are two incremental encoders, charged with measuring the leg's angle with the vertical axis (hip joint angle) and the spring's displacement (knee prismatic joint). As usual, both encoders provide three channels, A, B and index. They produce 500 counts per revolution (CPR) with equivalent resolution of 2000 CPR when in quadrature mode. The hip encoder is mounted on the motor, type HEDS-5540, made by HP. The second, same model and manufacturer, is mounted on a custom 3-bar mechanism called quasi-knee that allows to measure the prismatic joint's displacement, pictured on Figure 3-4. Practically, this mechanism connects the fixed with the moving part of the prismatic joint with two links connected with a revolute joint, while the encoder is mounted on its axis. The displacement is calculated indirectly by measuring the angle between the two links, taking into consideration the geometry. This part was present in the first setup, but the original encoder

was replaced in order to increase the resolution (500 CPR instead of 360) and to resolve some wiring and connectivity issues.

Figure 3-4. Quasi-knee mechanism.

Additionally, to record the body's velocity and acceleration, an inertial measurement unit (IMU) is employed, (ADIS16375 Analog Devices). This sensor has 3 gyroscopes and 3 accelerometers, thus it can count translational accelerations and X, Y, Z angular velocities.

Figure 3-5. ADIS16375 IMU sensor with breakout board.

To measure impact forces and torques, but also for the controller to be able to tell when the robot is on stance or flight phase, a force sensor is needed as it is the most common and reliable option. Impact can be determined also by measuring the spring compression, and this is how it was actually done, before a force sensor was installed. The disadvantage of this method is that we can only measure forces along the leg axis, so a switch to force sensor measurements was called for. The one finally installed was a Rikudo series sensor, by BOTA Systems. Rikudo is a miniature force/torque sensor utilizing strain gauges across all six axis. The sensor is available pre-calibrated from the manufacturer, while the calibration matrix is provided to be used with the accompanying software. We used the Rikudo-4243-S, which is the standard version. Made of aerospace aluminum alloy, the sensor is rigid, strong and lightweight enough – it weighs only 80 grams – to be placed at the toe position. Also the sensor is ROS-ready which greatly reduces the integration to a simple plug and read process, in a machine that can run Ubuntu, e.g. a Raspberry Pi. No other external power supply is required.

Figure 3-6. Rikudo-4243-S force sensor.

To mount the force sensor on the toe, a 3D printed adaptor was utilized displayed on the figure below. Also to avoid cosmetic and structural wear, a silicon hemispherical cover was constructed, using silicon casting on a plaster mold.

Figure 3-7. Rikudo force sensor with and without the silicon cover.

3.2 Electrical and electronic subsystems

3.2.1 Previous setup

The previous electronic subsystem was using a Beagleboard-xM as a main computer. Since this could only send and receive signals up to 1.8V, there was an additional board designed to convert signals from 1.8 to 3.3V and vice versa. There was also a DAC, MAX517 by Maxim, to accept a digital signal from Beagleboard and produce the voltage output required by the old motor drive.

All the electronic elements of this setup required 5V supply. For this purpose, a voltage regulator (LT1085, Linear Technologies) was utilized. This regulator needed a supply of its own, a voltage above 6.5V. To summarize, the old setup required:

- 24V to supply the motor
- 5V to supply the PIC microcontrollers responsible for reading the encoders (see Section 3.2.2), the Beagleboard and its amplifier, the DAC and the encoders
- 6.5V to supply the voltage regulator

To make all those different voltages available but also to mount the microcontrollers and the drive, a main board was designed and constructed; an auxiliary board to create the 5V supply was added.

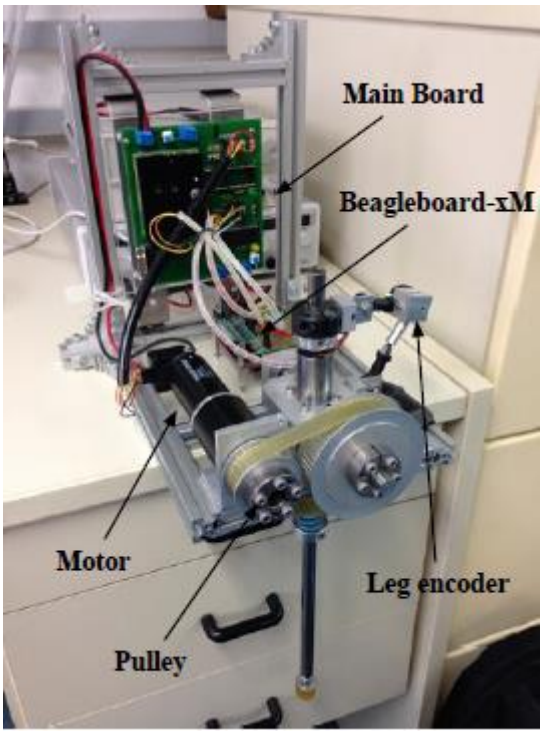


Figure 3-8. Monopod previous setup.

3.2.2 Redesign and current setup

The method with which the encoders would be read was the first decision to be made. This selection of method subsequently affects the whole redesign of the electric and electronic subsystems.

The older monopod version was using two Microchip PIC18F4331 microcontrollers, capable both for reading encoders with their incorporated QEI module and for producing PWM signals. Those microcontrollers required a 5V supply, as well as the construction of a breakout board that would connect and redirect each pin properly.

Figure 3-9. Microchip PIC18F4331 microcontroller.

Considering our new requirement for ROS utilization, as well as the specification for remote control, it was desirable for the new system to have the ability to be easily accessible and controllable from any computer on a local network that is using ROS; therefore there must be a way to send encoder measurements to said computer. It would be unwise and inconvenient to send A, B and I signals directly to the main computer, both because of the fragility of the cable type the encoder uses and the noise that would be inserted, even with a line driver and receiver. In any case, it is generally preferable to read the encoders locally and just transmit the angle value.

To this end, an Ethernet port could prove extremely useful. However the PIC18F4331 microcontroller does not offer one. Although perhaps there was a possibility to program it to perform this function, this solution was rejected as too complex and non-worthy of the time it would consume, since the microcontroller market was flooding with options, offering both enhanced incorporated Ethernet connectivity as well as augmented processing power, if in the future a requirement would arise to perform more functions locally, e.g. low level PID motor control.

We should clarify that probably every microcontroller with GPIO pins can read encoder signals using interrupts. This way, the amount of encoders a board is able to read simultaneously is only limited by the number of input pins it has. However, this method was tested and is was found out that little by little, for a repetitive swinging leg motion, the zero point started moving towards the sides, instead of remaining steady as it should in the vertical position. This is probably a result of some lost counts during the motion, which in our case was particularly fast. In cases of slower motion this phenomenon seemed to diminish or even disappear. This method was obviously rejected and it was decided to limit our microcontroller research to devices that offer one or multiple QEI modules.

A QEI module is an interface that accepts channels A, B and I and can calculate signed velocity and relative position. The way it works is that it detects whether pulse A or pulse B is leading, and defines the direction accordingly. It also offers increased resolution as from 500 CPR for channels A and B, now there are four different combinations between them, summing up to a total of 2000 equivalent CPR. Many microcontrollers offer libraries with convenient functions that can automatically return relative position and velocity values without any additional user editing.

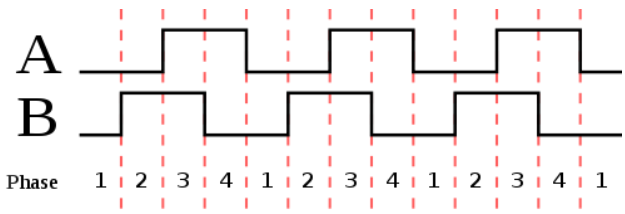


Figure 3-10. QEI module pulse succession.

Finally, the TM4C1294 Connected LaunchPad from Tiva C series by Texas Instruments was selected. The Beaglebone Black, with its incorporated eQEP module, similar to QEI, was also considered but it was deemed expensive (about 60$) and it was more powerful than necessary just for encoder reading. The Tiva board only costs 20$, it has an 120MHz 32-bit ARM Cortex-M4 CPU, 1 MB flash memory, an Ethernet Connection and it includes one QEI module. It also includes TivaWare, a set of useful libraries and is debugged using the Code Composer Studio, also provided by Texas Instruments.

Figure 3-11. Tiva C Series TM4C1294 Connected LaunchPad, Texas Instruments.

For the new design things were significantly simplified. We still need to create a 5V level to supply the three Tiva boards (one for the knee encoder, one for the hip encoder, and one for the IMU) and the two encoders. We also need the same 24V supply as before to power the motor. Since it is simpler and more compact to have a single power input for the whole system, a voltage regulator is required to convert the high voltage to low. For this purpose, a step-down regulator or buck converter was utilized. This type of regulator offers greater power efficiency, reaching 95%, compared to the classic linear voltage regulator, which also generates a large amount of heat. The regulator we selected is a Polulu D24V60F5, capable of handling currents up to 6A, more than enough to power our low power components. This part was mounted on a board with two outputs, to the motor drive and a second board we shall present below, and a power supply input (24V). The Regulator Board's (RB) schematic and final construction are presented in Figure 3-13 and Figure 3-14.

Figure 3-12. Polulu D24V60F5 5V Step-Down Voltage Regulator.

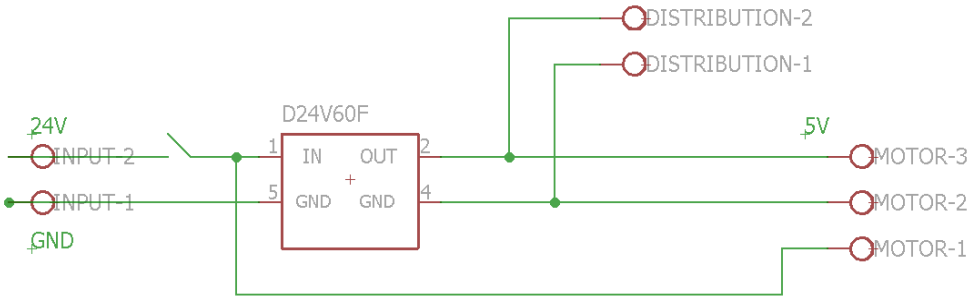


Figure 3-13. RB schematic.

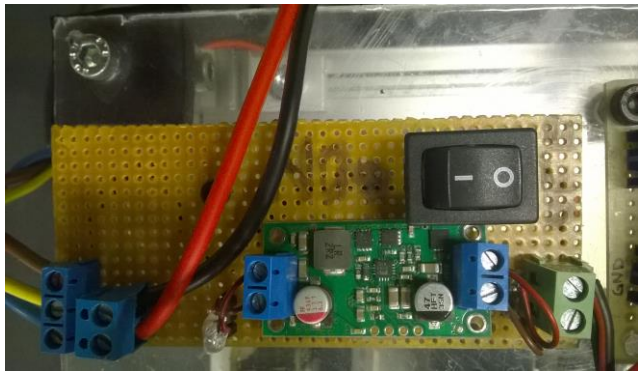


Figure 3-14. Constructed RB.

This voltage has to be distributed to the three Tiva boards through a *Power Distribution Board* (PDB). We chose to power the encoders directly from the Tiva boards and not use the master power board to keep the design compact and simple. The motor drive also is supplied and enabled by a Tiva board. Of course this nullifies the advantage of isolation between the two sides of the optocoupler as both sides will essentially have one common ground. This is not a great issue as in the unlikely case of an accident there is nothing expensive to be damaged, as there is on Laelaps (PC-104). Note that in our case the central computer only communicates with the robot through a rooter and therefore its electrical isolation is guaranteed.

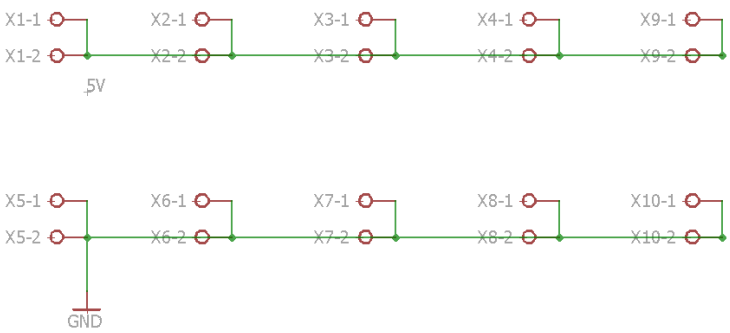


Figure 3-15. PDB schematic.

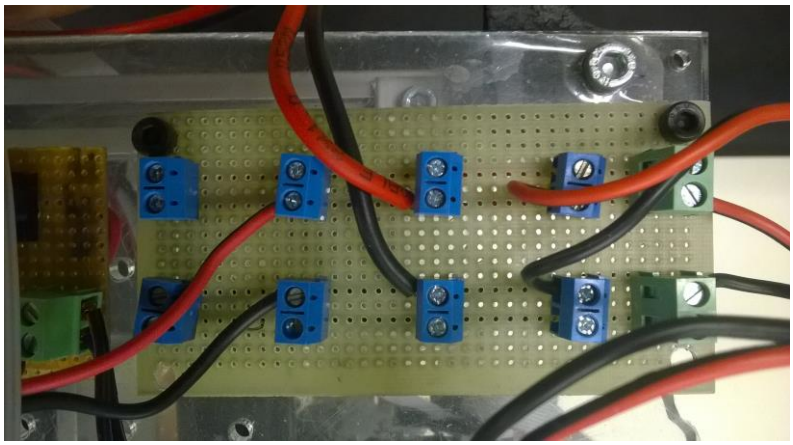


Figure 3-16. Constructed PDB.

To avoid using small cables and wires that would detach during impacts, but also to supply the Tiva boards from the Boosterpack instead from the debug port, a *Mounting Board* (MB) was constructed. It includes a small capacitor to normalize any voltage fluctuations as well as a standard Ethernet adaptor and a clip adaptor for the encoder flat cable, to ensure it will stay connected even during impacts. Its schematic, the final design and the constructed board are presented on Figure 3-17, Figure 3-18 and Figure 3-19 respectively. The construction was carried out using the lab's LPKF machine.

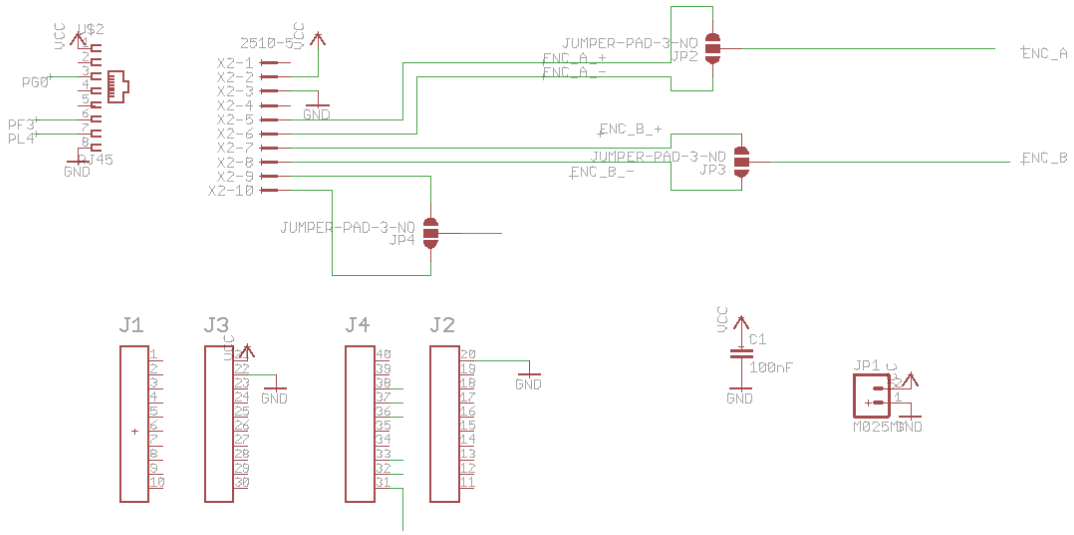


Figure 3-17. Tiva MB schematic.

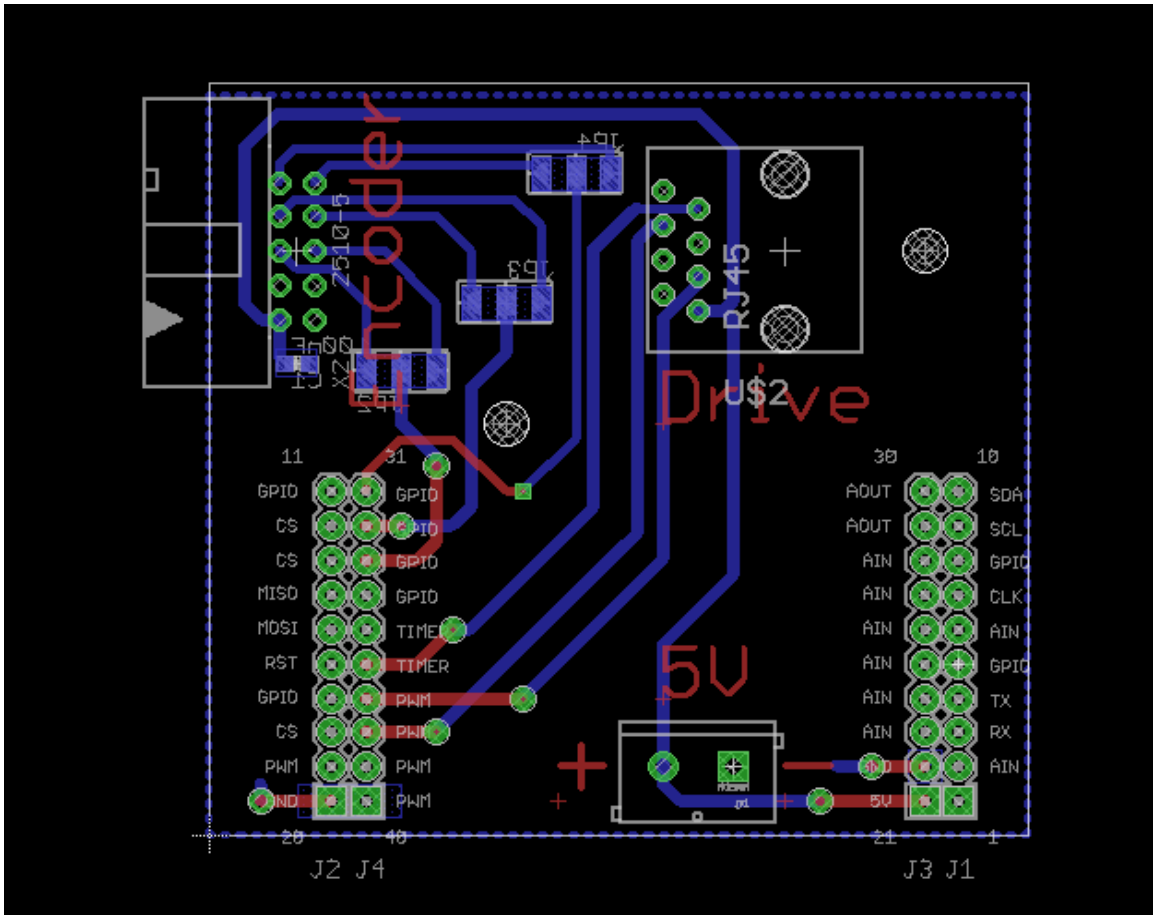


Figure 3-18. Tiva MB final design.

(a) **(b)**

Figure 3-19. Constructed MB. (a) top, (b) bottom layers.

3.3 Tiva

As is already mentioned, the basic design specification shared between all robots in the lab, and the treadmill on which they are tested, is the ability for cross communication and control using one common control center. This process requires two different parts of code. The first is the one that will receive commands, send measurements and handle the communications between the robot, or treadmill, and the control center. In our case, this part shall run on the Tiva microcontroller.

The code utilized activates the PWM and QEI modules to perform the tasks at hand. Also it sets up the UDP communication, which shall be discussed below. The user can easily select what modules he wishes to enable by defining the corresponding parameters. For example, by defining ENABLE_MOTOR and ENABLE_ETHERNET the aforementioned modules are enabled. One can also define ENABLE_IMU to enable inertial measurements and ENABLE_UART for debugging purposes. The whole code is included in Appendix A.

3.4 ROS

This is the second part of the control system, as mentioned before. It consists of five basic nodes, namely the *Hip, Knee* and *IMU interfaces*, *Read setpoint*, *PID control*, *Botasys driver* and *high level controller*. All of them are included in appendix A.

a. Hip_interface node

This node is the one that is charged with communicating with the hip Tiva board. Its function is depicted in Figure 3-20. Tiva is continuously transmitting encoder measurements on a set rate, using the User Datagram Protocol (UDP). This protocol is preferred over the standard Transmission Control Protocol (TCP), mostly because of its speed. In time sensitive applications, like real time control, TCP is rather inappropriate, as it includes error checked delivery of data streams, causing a significant delay. On the other hand, UDP has no such capabilities. For example, whenever a message is sent there is no guarantee it will reach its destination and no retransmission can be requested. Also there is no guarantee for ordered delivery. For example if one sends a message A and a message B right after, the order with which those messages will be delivered cannot be predicted in advance, although it was noticed that in their vast majority the messages arrived in the order they were sent. For these reasons, this protocol is faster and it is preferred when we favor speed over accuracy. In this node, we set up an asynchronous server to receive and read packets, and we define the board's IP and the send and receive ports. Those ports are software structures, numbered between (0 and 65535), and provide application multiplexing. This way the computer can tell which data correspond to which board, because there are three of them sending data to the same IP address.

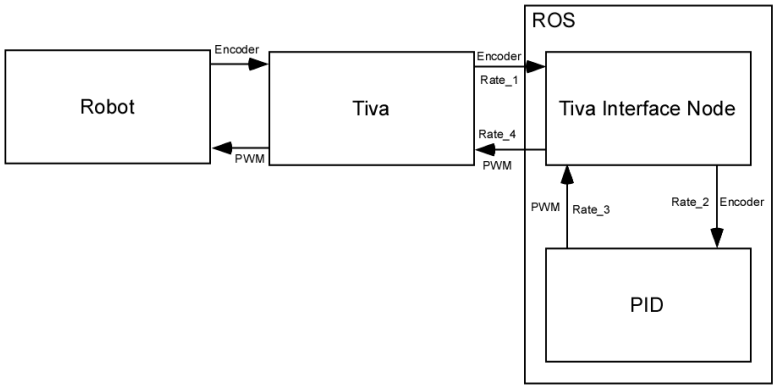


Figure 3-20. Tiva – ROS connection.

The measurements are read using a *sigaction* function and a signal handler. Every time the node receives an angle, this handler is triggered. The execution of the main function is interrupted, data is read and assigned to a global variable. The execution of main is then continued from the point it stopped.

Received data corresponds to hip angle measured in counts, relative to the initial angle where the board was activated and started measuring. We selected the zero point to be at the vertical position, mostly to have better visual control over the real and measured angle. For clockwise rotation the angle would be considered positive and for counterclockwise, negative, in order to avoid measurements with discontinuities. However this is exactly how the QEI library works. When we move towards zero from a small positive angle, the counter will reset and count the largest value defined on memory, which should correspond to 359 degrees. For example, we will count 2, 1, 0 and 359 degrees, when we would actually like to count 2, 1, 0 and -1 degrees. This is represented on Figure 3-21. With black we can see the output of the QEI library, and with red the desired output.

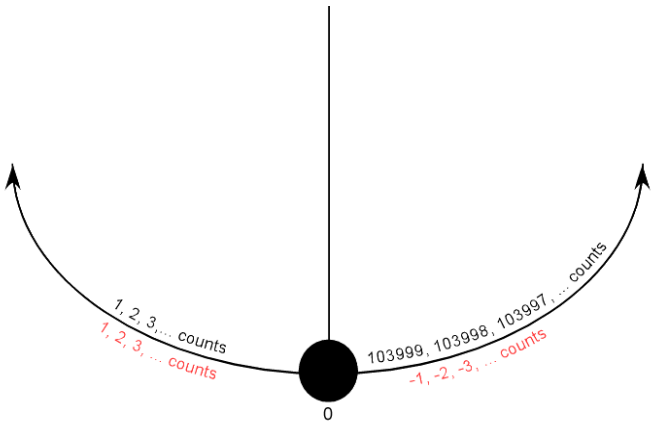


Figure 3-21. Angle succession, black for the QEI output and red for the desired.

Instead of modifying the QEI library, we take care of the issue in this node. If the algorithm detects transition from a small positive to a large positive value, e.g. from 100 to 103500 counts, and given the large sampling rate, it assumes there was a zero crossing and it

subtracts 104000 counts (or 360 degrees) from the measurement, e.g. 359-360=-1 degrees, or the corresponding value in counts. The same subtraction should occur when we previously ended up with a negative value, and we keep receiving descending positive values, which means the leg crossed zero and it keeps moving in the counterclockwise direction. This whole conversion occurs inside the signal handler.

Next, the final angle value is published on the proper topic, usually the one named /*state*, as we send measurements directly to the PID controller. There are two separate ways to handle the publishing. The first way is to publish the measurements inside the signal handler. This way data is published in the same rate it is received, while the main function is practically left empty. E.g. for a Tiva transmission rate of 5 kHz, the publishing rate will be about 4.98 kHz too. This way we have direct control over the publishing frequency. However, this method results in various crushes and program shutdowns, therefore it was not selected.

The second is to publish data inside the main function. This method however, also presents an issue. Since the main function's execution flow is interrupted by the signal handler, its actual loop rate is not the one we define, but it depends on the defined rate and the rate of interrupts, or in other words, the Tiva transmission rate. Consider Table 3-1 with some indicative experimental values. As one can observe, as Tiva's rate increases, the deviation between the defined and the achieved loop rate also increases.

Table 3-1. Tiva transmission rate – control loop rate relation.

Tiva transmission rate (Rate_1 - Hz)	Defined loop rate (Rate_2 - Hz)	Achieved loop rate (Rate_2 - Hz)	Functionality
1000	<1000	Equal to defined	Unstable
1000	1000	~995	Works at first, ends up unstable
10000	1000	~700	Works
15000	1000	~200	Works
15000	10000	~1650	Works
15000	20000	~2700	Works (Best case)
20000	-	-	Cannot publish values
40000	-	-	Cannot publish values

As we can observe, as Tiva sends measurements faster, ROS has increasingly more trouble reaching the defined loop rate. Above 20 kHz, ROS does not even manage to publish the measurements, in any defined loop rate.

It is really important to define what we mean by result. Rate_1 was initially set to 5 kHz. In the case that a step command input was applied, it was observed that the control gains could not exceed some specific values. Generally speaking, a rise in Kd should result in less oscillations and a smoother response. While this was happening at first, for Kd > 0.00012 the oscillations would start to rise again. As a result we could not increase Kp above its own set value (0.0008) because there was no way to reduce the oscillations that would cause. Therefore the system was underperforming.

When Rate_1 was increased, e.g. to 15 kHz, we managed to increase the gain values and reach the motor's limits (Kp = 0.003 and Kd = 0.00024). These are the control gains that correspond to the experiments presented in Table 3-1, and those that decide if the setup is functional or not. So when we refer to instability, it is implied that the setup is unstable for those specific gains, and that it could be stable for lower gains. We will discuss further the control gains and how we can theoretically predict their values in Paragraph 3.6.2. The reason we mention this here is to show those gains predicted in that paragraph could not be reached for a different control loop rate. Additionally, note that what we observe is due to the set transmission rates and not to root locus behavior. As one can observe in Figure 3-22, an increase of Kd, for a fixed Kp, could not possibly cause any increasing oscillations and eventually instability.

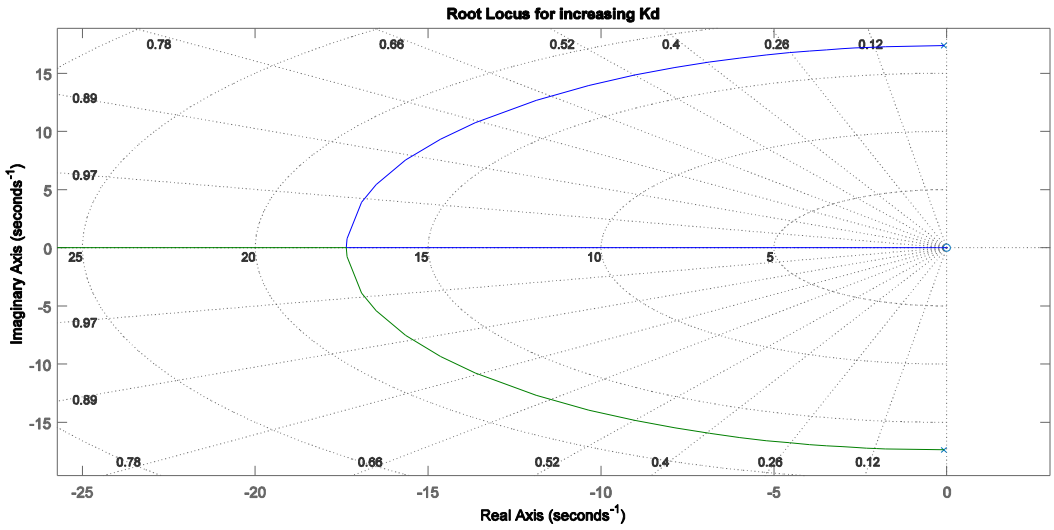


Figure 3-22. Root locus for increasing Kd value.

Finally, this node subscribes on the */control_effort* topic, reads the calculated necessary PWM duty cycle and sends it back to Tiva.

b. Knee_interface node

This node is practically the same as the previous one. The only thing changing is that the knee board does not actuate a motor, therefore there is no need for subscription to any topic. The

quasi-knee angle measurement is received like before only this time it has to be translated into a length in order to be published. The compression values are published to the */compression* topic. The translation occurs as follows.

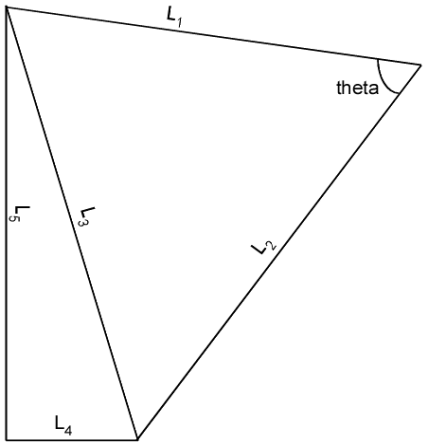


Figure 3-23. Quasi-knee geometry.

We are measuring angle theta, and wish to count L_5.

$$L_3 = \sqrt{L_1^2 + L_2^2 - 2L_1L_2\cos(\text{theta})} \tag{3-1}$$

and

$$L_3 = \sqrt{L_4^2 + L_5^2} \tag{3-2}$$

therefore

$$L_5 = \sqrt{L_1^2 + L_2^2 - 2L_1L_2\cos(\text{theta}) - L_4^2} \tag{3-3}$$

We should finally mention that there is no problem with that node's loop rate as it does not affect the control system's rate in any way. Of course same things hold but, since the measurements can be sent from the board at a lower rate, e.g. 1 kHz, we can use the rates mentioned on the first case of Table 3-1.

c. IMU_interface

As in the two previous cases, this node receives the X, Y and Z angles and respective accelerations, and publishes them to the */IMU_feedback* topic. There was an effort to integrate measurements in order to get the total displacement, but we found out quickly that the accumulating error makes that almost impossible. More complex methods, and of course more than one sensors are required to get an accurate estimation. One method that could be utilized is sensor fusion, but there was no further investigation on the matter [20] since there were no other sensors available.

Again, this node's loop rate can be defined without worrying about affecting the closed loop's rate.

d. Read setpoint

This node reads a desired position in counts using the rqt_reconfigure tool. It was only used on the PID hip position control experiment, as for more complex experiments this angle is usually defined by a high level controller.

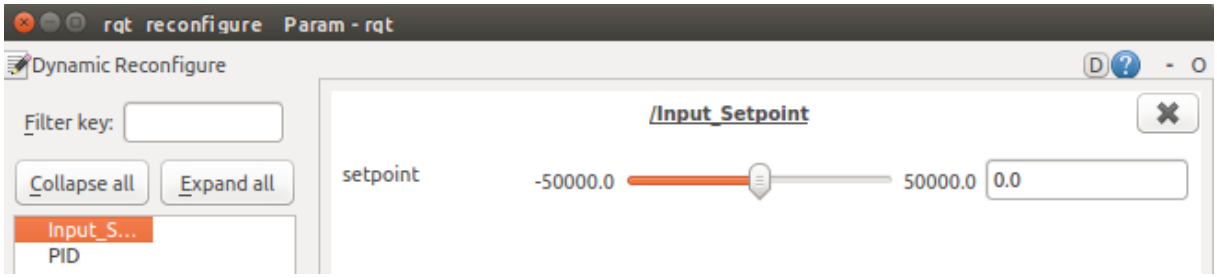


Figure 3-24. Setpoint definition in counts.

e. PID controller

This is the controller provided by the *ros_pid* package. It offers various tools, like dynamic reconfigure for gains, a filter on the estimation of the derivative term, saturation limits and easily configurable published and subscribed topics, in case we wish to rename them, or run more than one controllers at once so name conflicts need to be avoided. This package was selected over the *ros_control* package, as the latter's structure is significantly more complex and for the time being it lacks documentation.

The reason we took special care of the Hip_Interface node's loop rate is that the controller has no loop rate of its own. It operates at the rate it receives messages or, in other words, at the rate that the */state* topic is published. This means that the */state*'s topic publishing rate is in fact the rate of the entire control loop.

The filter was also a point that needed some consideration. This is a low pass filter designed for audio applications, with a default cut off frequency of 3 kHz. It is useful in cases of heavy spiking behavior of the velocity estimation. However, as we shall discuss in Paragraph 3.5, those filters also insert a phase shift. This effect was in fact verified in a Gazebo simulation; we decided not to modify the default value without further investigation.

Finally for the gain definition this node uses the same tool we used to define the setpoint, *rqt_reconfigure*.

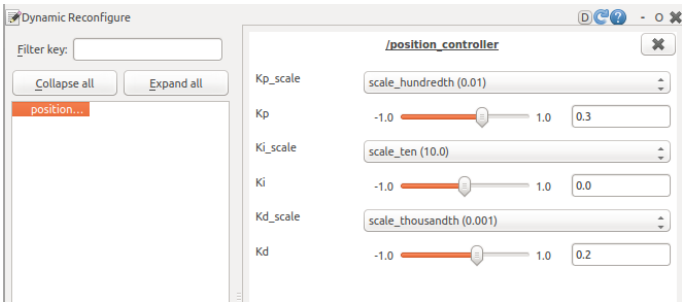


Figure 3-25. PID control node gain reconfigure.

f. Botasys driver

Except for the main nodes mentioned previously, we also use the force sensor's driver to read measurements. This driver comes in a ROS package that only requires building. There are two different launch files. The first, *calibration.launch*, is quite self-explanatory and handles the sensor's calibration. The second, *driver.launch*, is the one that receives and publishes measurements in */botasys* topic. Because the initial measurements come with some noise, this driver provides an integrated filter that publishes filtered data on */filtered_botasys*. Finally there is a capability to visualize the force's direction and magnitude using *Rviz*.

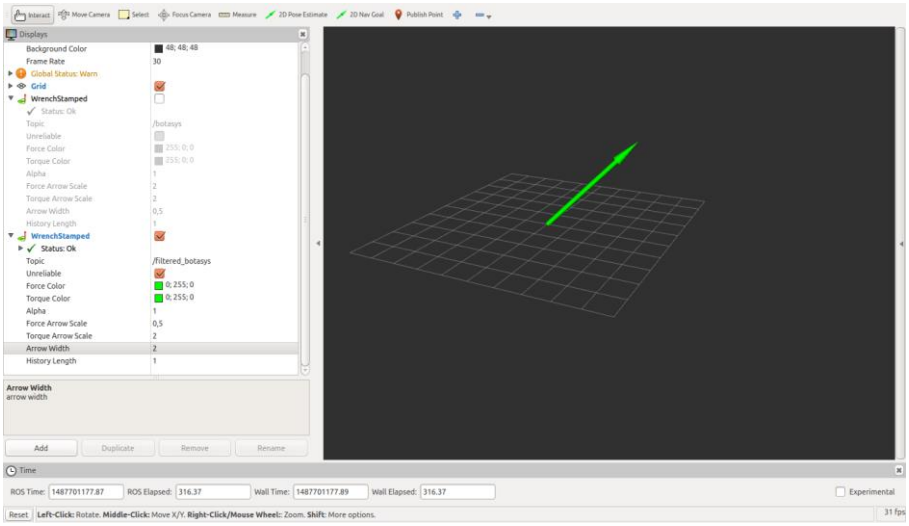


Figure 3-26. Botasys sensor force-torque vector.

g. High level controller

We used another additional node to conduct a simple dynamic hopping experiment. This node reads the compression and angle values and consequently determines if the leg is on a flight or a stance phase. In the first case it activates the PID controller and sends a setpoint, the touch-down angle. During stance phase, it deactivates the controller and sends a PWM duty cycle directly to the */control_effort* topic, which corresponds to a set torque. The controller needs to be deactivated because, since it is receiving data on */state* continuously, it would keep operating and interfere with the continuous torque command we want to apply.

The main parameters to configure in this node are the compression after which we consider the leg is on stance phase, the touch-down angle and the torque applied on stance. The duty cycle is also important as this is the node that drives the controller. However since the measurements are not read here, therefore we do not have the previous delay issue, the user can just define the loop rate in which he desires the control to be closed.

This walking mechanism, if given proper initial conditions (free fall height and translational forward velocity), and if there were no energy losses, its passive nature would allow it to keep moving without actuation, while on stance phase, just utilizing its initial kinetic energy.

However when losses are added, the system becomes quite parameter sensitive and it is hard to find the initial conditions that would lead to stable motion. Therefore it is a matter of great significance to estimate their values, as well as the values of stance phase torque and touch-down angle.

3.5 Simulation in Gazebo

3.5.1 System modeling

The first step towards a complete Gazebo simulation is creating a model with physical properties. Gazebo uses SDF files to describe a model [40]. However when one wants to use it in combination with ROS, a different type of file is used, called URDF [45]. This file is automatically converted into SDF by Gazebo. More info can be found in [46].

This format is quite outdated and lacks several features. One of the most important and easily observable flaws, even after the first attempt to create a model, is the lack of a spring stiffness parameter. An approximation of elastic behavior can be achieved by using the CFM and ERP parameters. The ERP detects joint position errors and tries to correct them by the defined factor, e.g. a value of 0.2 corrects the error by 20%. As it is obvious, increasing the ERP might make the joint more rigid, however it increases the numerical instability. The CFM works in a similar fashion with velocities and allows their modification. In contrast to ERP, increasing the CFM will soften the joint behavior and will improve stability. Those parameters combine stiffness, damping and time step in the following formulas [41]:

$$ERP = \frac{tK}{tK + b} \tag{3-4}$$

and

$$CFM = \frac{1}{tK + b} \tag{3-5}$$

where K stands for the spring stiffness, b for the damping factor and t for the time step. It is suggested in literature that one can achieve an accurate representation of a spring and damper system simulated with implicit first order integration. However every time we modify the time step, these parameters also need to be adjusted properly, therefore their utilization is not suggested, except maybe in the case one wants to model a soft stop.

The aforementioned hold when one wants to insert a model through ROS, according to the corresponding tutorial on the Gazebo page about connection with ROS, model actuation and control using the ros_control package [42]. However we have already selected to control the hip joint position with the ros_pid package. This allows us to use the more advanced SDF

format description, if of course we find a way to connect Gazebo and ROS. To summarize the main objectives are:

- the use, if possible, of SDF files with advanced modeling capabilities
- the connection with ROS, in a similar fashion with the real robot

A first approach is to insert the robot model as part of the simulation environment in the corresponding world file. This way the first objective is achieved. Afterwards, if possible, one can construct a mechanism on top of the environment, like a brace, using a URDF file as ROS demands. This structure, without any properties like stiffness which is absent in the URDF file, should only actuate and essentially move the robot. In our case the final system is presented in Figure 3-27 (white-monopod robot as part of the environment, black-exoskeleton).

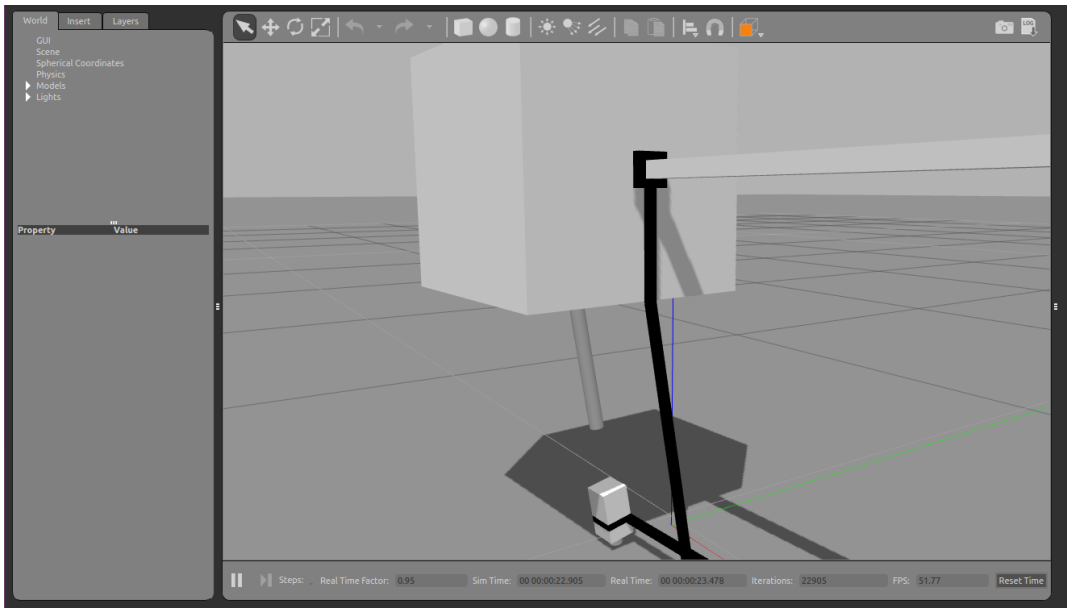


Figure 3-27. Monopod model with external brace for actuation.

This method, although it could work, it is quite hard to implement. It requires relatively small link weights and inertias in order to be realistic, something that Gazebo does not handle well and could lead to the whole model collapsing and simulation breaking down. Also it is hard to receive measurements from the actual joints (position, velocity, torque) without using an additional plugin. Finally, there is no correlation between the simulated and the real system. Therefore the main issue remains.

A solution to our problem is finally offered by custom plugins. These plugins can virtually perform any function a user wants. In our case, the target is to create a topic that accepts as input commands from a controller and as output applies torque to a joint. A topic that contains the joint state (e.g. position) is also required. The second topic is relatively easy to construct, using the standard *joint_state_publisher* plugin that is available on the Gazebo plugins folder.

Note that if, in addition to position, publication of velocity and torque is required, the standard plugin has to be modified.

To create the command topic, another custom plugin was constructed, named *monopodplugin*. This plugin has, again, to be built and moved into the gazebo plugins folder. These two plugins also have to be added in the bottom of the SDF model in order for Gazebo to load and activate them. Afterwards we only have to create a world and a launch file that will run all the necessary nodes. All the codes, plugins and models are included in Appendix A.

Finally we end up with the system shown in Figure 3-28. Its similarity with the one used on the real robot shall become obvious on Paragraph 3.7 (Figure 3-44), where we conduct the experiments. The difference is that instead of the *Tiva_Interface* node we have two new nodes. The *State_Callback_Interface* to receive measurements from Gazebo, and the *Command_Interface*, to send commands.

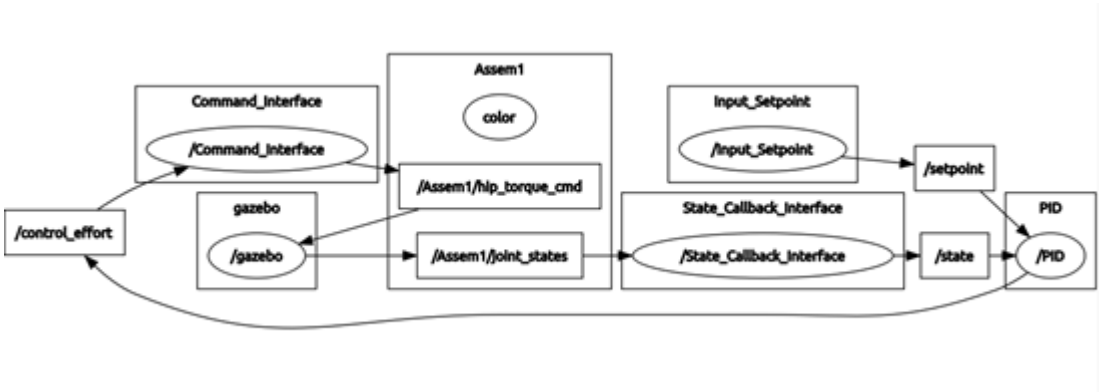


Figure 3-28. ROS control system structure in Gazebo simulation.

3.5.2 Solver parameters configuration - Matlab comparison

As already mentioned, Gazebo offers four different physics engines, ODE, Bullet, Simbody and DART. We selected the ODE solver, as it was the easiest to set up and configure. This solver also allows for easy verification and comparison of the produced results with those from the same simulation in Matlab, which is also using said solver. We shall examine some fundamental parameters that the user has to configure in order to produce accurate results.

Time step

Two values were tested, 1 ms and 0.1 ms. Both responses agreed with those from Matlab simulations. In case that the simulation duration is irrelevant, it is suggested to use the smaller time step. In all other cases, it is preferable to use the bigger, for reasons that will be mentioned subsequently.

PID loop rate

The controller's loop rate (LR) should match the simulation's solving frequency. For example, considering the two mentioned time steps, the LRs should be 1 kHz and 10 kHz respectively. The second case is quite computationally demanding, especially if many different PIDs must run simultaneously (e.g. on a quadruped robot).

This is one of the reasons the smaller time step, if not otherwise required, should be our first choice while defining the simulation's parameters. If the LR is not selected appropriately, the simulation results will not match the expected ones (received e.g. from Matlab), as shown in the following diagrams. All following figures present simulation results of PID control on the hip joint, and specifically the response to a step input of 6000 counts, or about 20.5 degrees, for the same Kp and Kd gains.

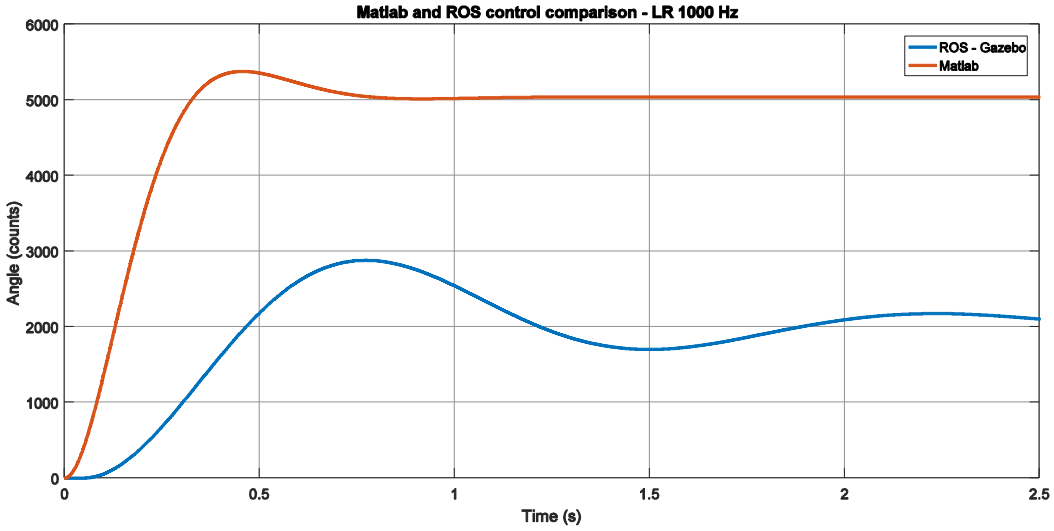


Figure 3-29. Gazebo - Matlab control comparison. Time step = 0.1 ms, LR = 1 kHz, Update rate = 100.

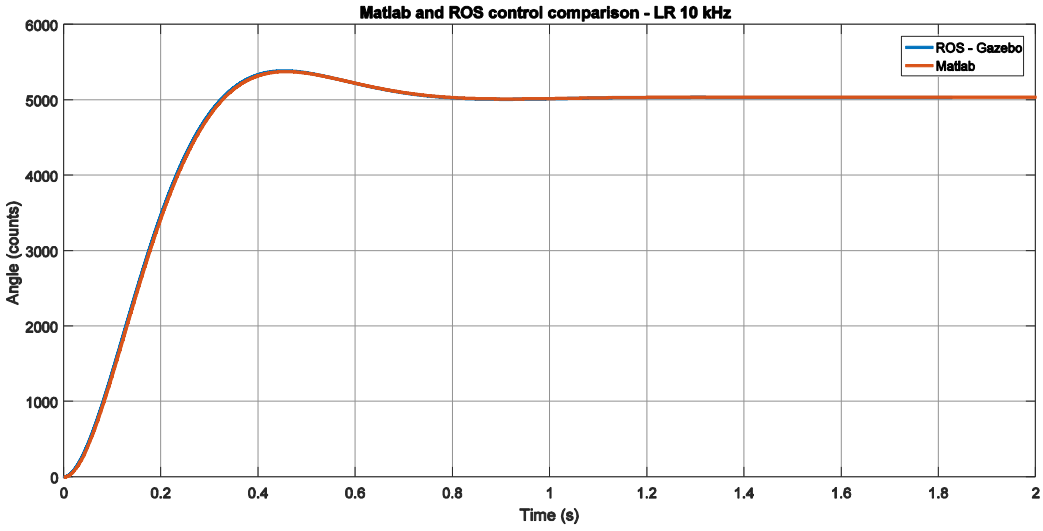


Figure 3-30. Gazebo - Matlab control comparison. Time step = 0.1 ms, LR = 10 kHz, Update rate = 100.

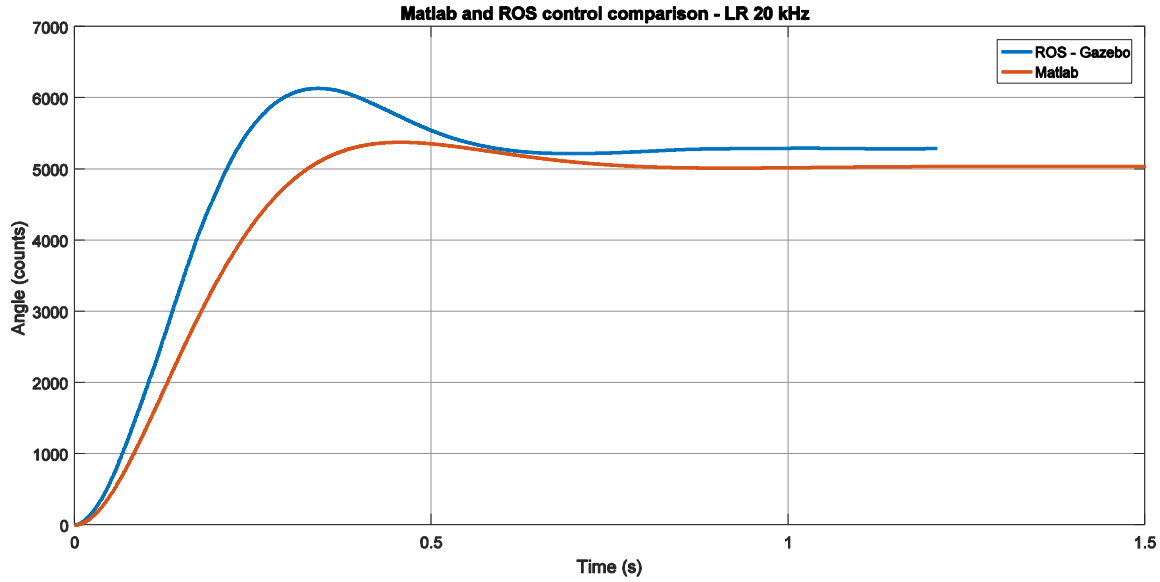


Figure 3-31. Gazebo - Matlab control comparison. Time step = 0.1 ms, LR = 20 kHz, Update rate = 100.

Solver type

Two choices exist, i.e. *quick* and *world*. It is reported that quick, as one could imagine, is faster but offers somewhat lower accuracy. However, it was observed that there were some issues with said solver type. Specifically, a constant torque was exerted to a simple pendulum model. While it was expected that the pendulum would be constantly accelerated to theoretically infinite velocity, that was not the case. Instead the velocity diagram shown in Figure 3-32 was recorded in ROS.

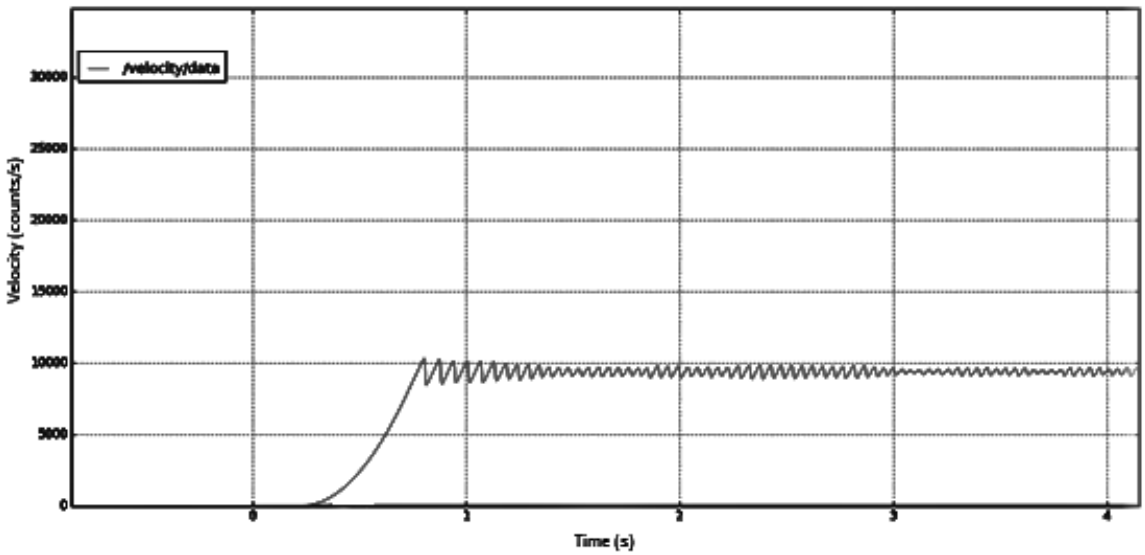


Figure 3-32. Pendulum with constant torque applied (g = 0, b = 0, no static friction).

Using the *world* type solver, that problem was resolved. Therefore its use is recommended.

Real time update rate

The real time update rate parameter specifies in Hz the number of physics updates that will be attempted per second. If this number is set to zero, it will run as fast as it can. Note that the product of real time update rate and max step size represents the target real time factor, or ratio of simulation time to real-time. Therefore:

$$Real\_time\_factor = real\_time\_update\_rate \cdot dt \tag{3-6}$$

The *real time factor* parameter expresses the correlation between the real time and the simulated time. For example, if during two real seconds, Gazebo has solved one second for the simulated system, real time factor is 0.5 (1/2). Consequently, the meaning of the equation above is clear. If the time step is 1 ms and the solver is called 1000 times per second, real time factor will be equal to 1. In one real second, a simulated second will be solved.

This point however needs a lot of caution. The user can select the real time update rate and the real time factor in the simulation launch file, but one needs to keep in mind two things:

1. Defining the real time update rate seems to transcend the definition of the real time factor. To make this clear, consider the following example. We define a 0.1 ms time step, real time factor equal to 1 and real time update rate equal to 100. Those values are incompatible, according to (3-6), and one of those should be modified to satisfy it. The real time update rate could become equal to 10000. This does not happen and we end up with real time factor = 0.01, as the aforementioned equation dictates, which means that this is the parameter that must be changed in the case of conflicts.
2. There is no guarantee that Gazebo will manage to achieve the defined real time update rate, end even if it does, that there will not be any unexpected behavior in the simulation due to increased computational load. In our case there was an unexplained oscillation at steady state. Consider the following cases.

A time step of 1 ms and the appropriate loop rate (1 kHz) are defined. If the update rate is set to 1000, as we discussed previously, the following response is recorded (Figure 3-33):

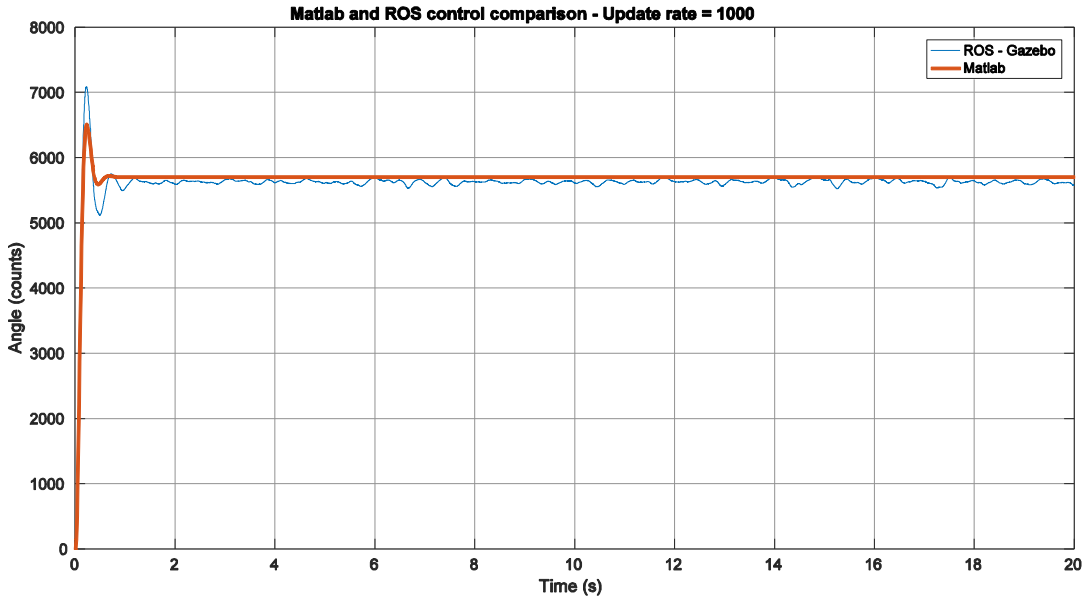


Figure 3-33. Gazebo - Matlab control comparison. Time step = 1 ms, LR = 1 kHz, Update rate = 1000.

The expected real time factor (equal to 1) was not achieved. We end up with a value of 0.93, which also appears to be oscillating between 0.9-0.95. This instability suggests that there might be a system overload. This hypothesis is supported by Figure 3-33, where random oscillations appear at steady state.

This issue is observed better if we alter the update rate to 10000 calls per second (estimated real time factor = 10). Again, the update rate parameter reaches only 4 when it should actually become 10, with even bigger variation (3.8-4.5). Also the oscillations at steady state increased dramatically, see Figure 3-34.

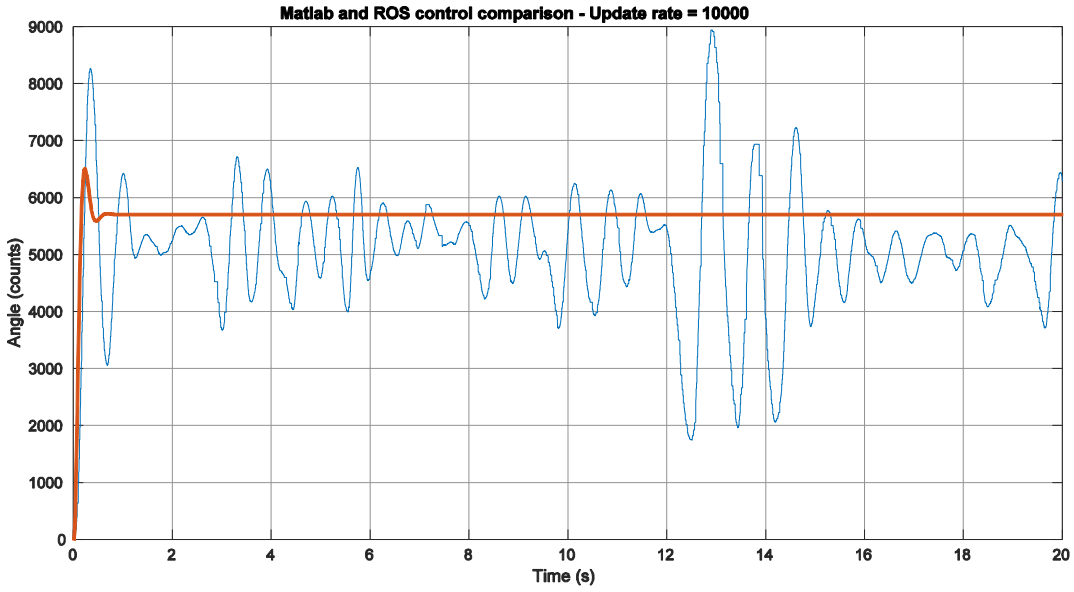


Figure 3-34. Gazebo - Matlab control comparison. Time step = 1 ms, LR = 1 kHz, Update rate = 10000.

We can easily deduce form the aforementioned results that the update rate, in combination with the ROS loop rate that runs simultaneously, is a parameter that loads significantly the computational system. For that reason, we choose a smaller time step, in order to choose the lower loop rate, but also to match the loop rate value on the real robot as well. It is unlikely to exceed values of 1 kHz on the actual hardware.

In conclusion, it is hard to give an update rate value that will suffice for all cases. For the two time step values that were mentioned before, a value that does not cause instabilities and variations at the real time factor value is 100. This value is, of course, just an example and it depends on the computational system. It is suggested to fine tune these parameters by comparing the Gazebo simulation with one from another simulation environment, e.g. Matlab.

3.5.3 Spike examination

If we try to compute the velocity for one of the previous cases, we will record the responses presented in Figure 3-35 and Figure 3-36.

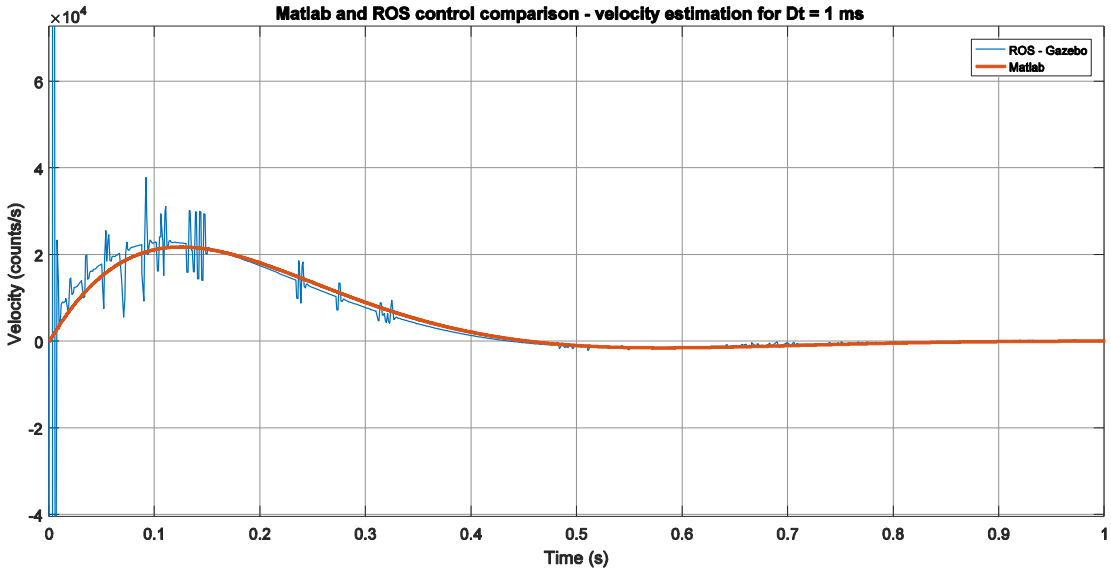


Figure 3-35. Gazebo - Matlab control comparison. Time step = 1 ms, LR = 1 kHz, Update rate = 100.

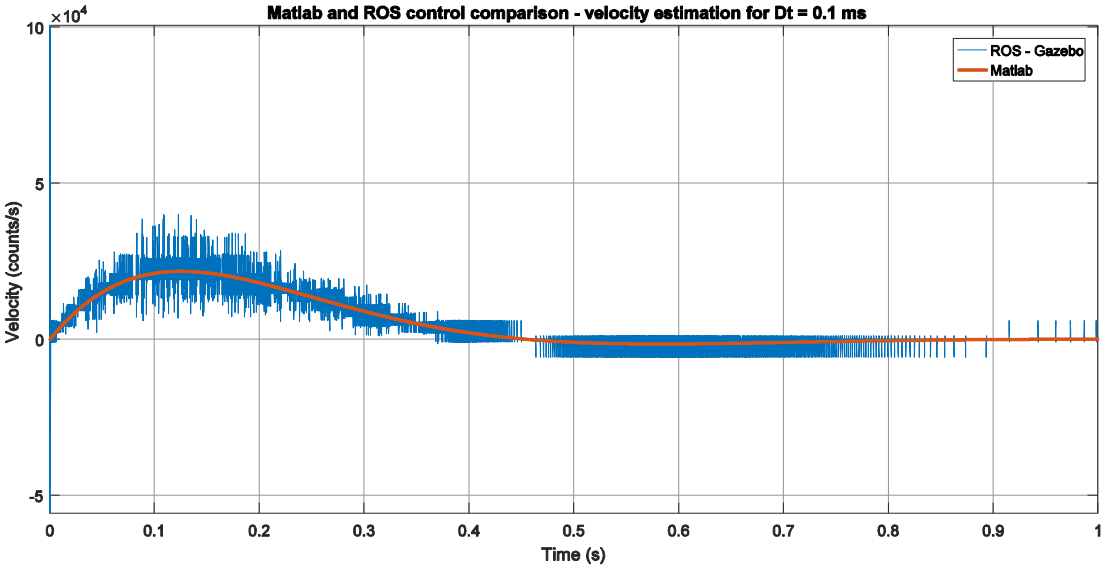


Figure 3-36. Gazebo - Matlab control comparison. Time step = 0.1 ms, LR = 10 kHz, Update rate = 100.

As observed, there are spikes in the velocity estimation as taken from the PID. In fact the problem worsens in the case of a smaller step, where the control loop rate is 10 times higher.

The problem in the smaller time step case can be normalized using the incorporated PID's filter in the ros_pid package. By altering the cut off frequency from 2.5 kHz to 300 Hz, the result shown in Figure 3-37 is recorded:

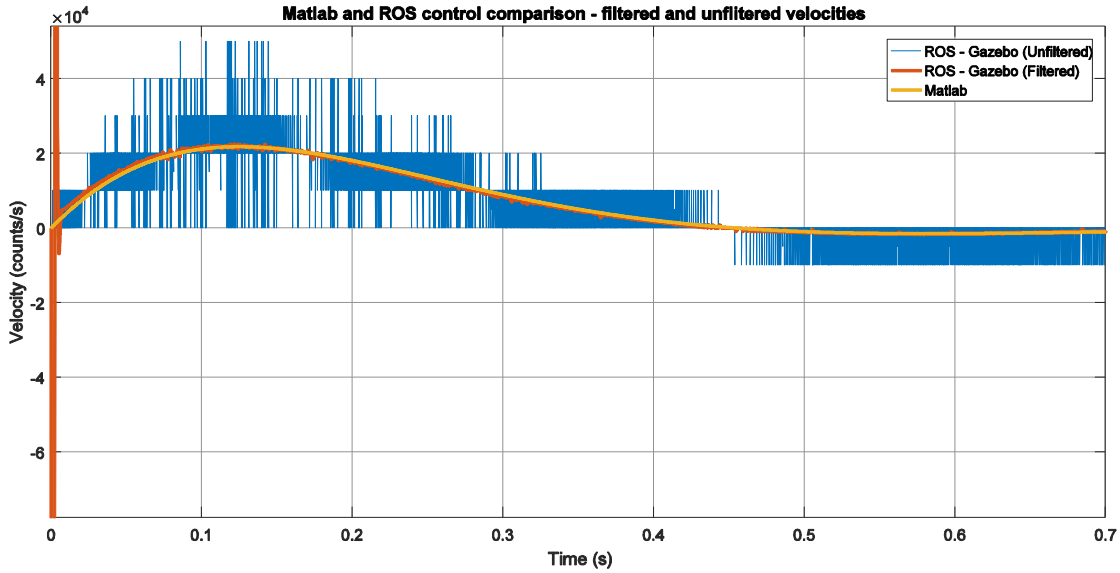


Figure 3-37. Gazebo (filtered and unfiltered velocities) - Matlab control comparison. Time step = 0.1 ms, LR = 10 kHz, Update rate = 100. Cut off frequency adjusted to 300 Hz.

In red colour, we can see the clearly improved velocity form. The oscillations can vanish totally if we lower the cut off frequency even more. However, in such a case, the behavior of the model is altered too, because of the phase delay that the filter introduces. Consider Figure 3-38 for cut off frequency equal to 30 Hz.

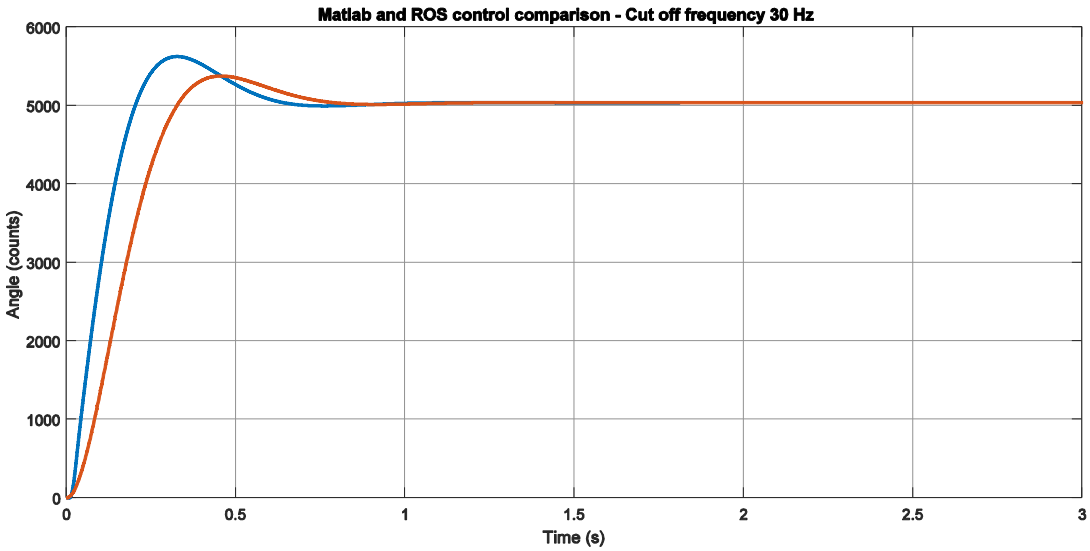


Figure 3-38. Gazebo - Matlab control comparison. Time step = 0.1 ms, Loop Rate = 10 kHz, Update rate = 100. Cut off frequency adjusted to 30 Hz.

In the case of a larger time step (and loop rate), those values do not seem to work in a similar fashion. The model's behavior starts altering earlier. Therefore, without further investigation, the use of this filter is not suggested.

In any case, the filter would not deal with the actual cause but it would only mask the problem. Being suspicious from previous cases, where system overload was leading to unexplained simulation phenomena, we lower the update rate (referring to the case of time step equal to 0.1 ms). The result is shown in Figure 3-39, where the spikes are less, and with smaller amplitude.

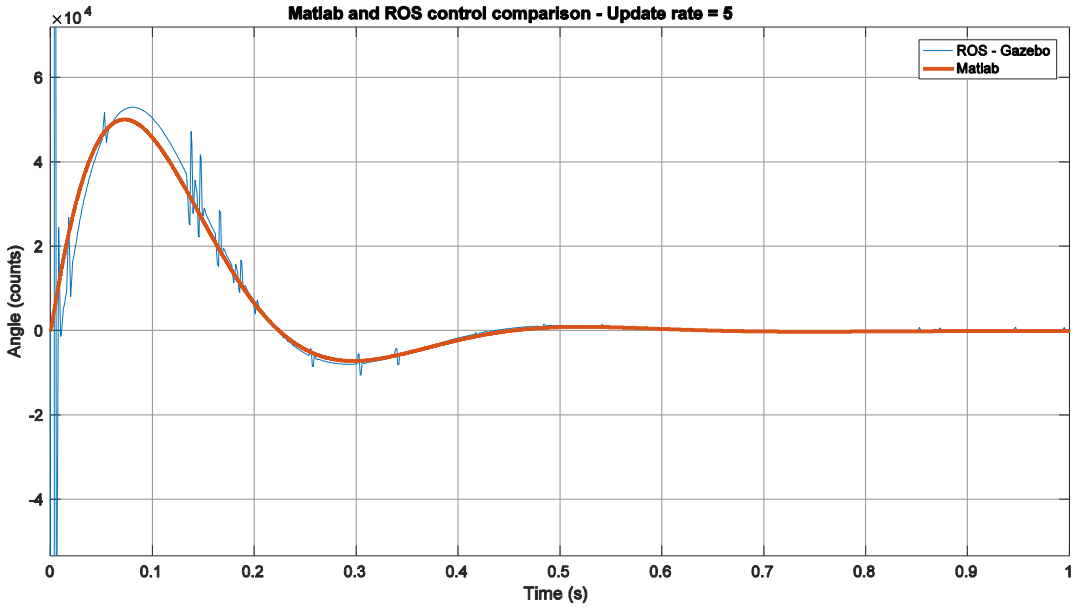


Figure 3-39. Gazebo (filtered and unfiltered velocities) - Matlab control comparison. Time step = 1 ms, Loop Rate = 1 kHz, Update rate = 5.

The disadvantage is that we end up with an extremely small real time factor, which results in long simulation duration. It's up to the user to determine the update rate value, keeping this tradeoff in mind, and considering that those spikes do not affect the simulation's behavior (it is the same for both update rates).

3.6 Experiments and comparison

Three main experiments were conducted to verify the setup's functionality. The first was a passive static hopping, to test measurements on the knee joint and examine how close simulation could approximate reality. The second was a PID control on the hip joint, to test software setup and also examine simulation accuracy. Finally the high level controller that was previously described was used, in order to test all of the sensors at once and verify that there would be no complications.

3.6.1 Passive static hopping

For the first experiment, the robot was left free to fall from a 5 cm height, while the leg was commanded to remain on the vertical position. The responses both from the simulation and the experiment were recorded using the rosbag tool from ROS and inserted in Matlab using the code in Appendix A.

Although the two responses are almost identical in the beginning, after the first second we can observe a small deviation between them. There are two reasons that could lead to this phenomenon. First of all there might be a small miscalculation between the actual parameters and those we got from Solidworks. Next, and more significant, this could probably occur because of unmodeled dynamics. E.g. we notice a small damped oscillation on the real response when the leg reaches and collides with the mechanical stop and absorber. This is not modeled and it functions as an energy drain for our system. Also there is no ground model on our simulation, whereas the treadmill's belt, where we conducted the experiment, functions as an absorber too. Nevertheless, this model managed to simulate all four bounces as well as the actual settling time, steady state compression and general form.

Table 3-2. Passive static hopping simulation parameters.

Parameter	Value
Real time update rate	100
Step time (s)	0.001
Real time factor	0.1

Table 3-3. Passive static hopping experiment parameters.

Parameter	Value
Defined PID control loop rate (Hz)	8000
Achieved PID control loop rate (Hz)	~1000
Tiva transmission rate (Hz)	15000
Kd	0.003
Kp	0.0002

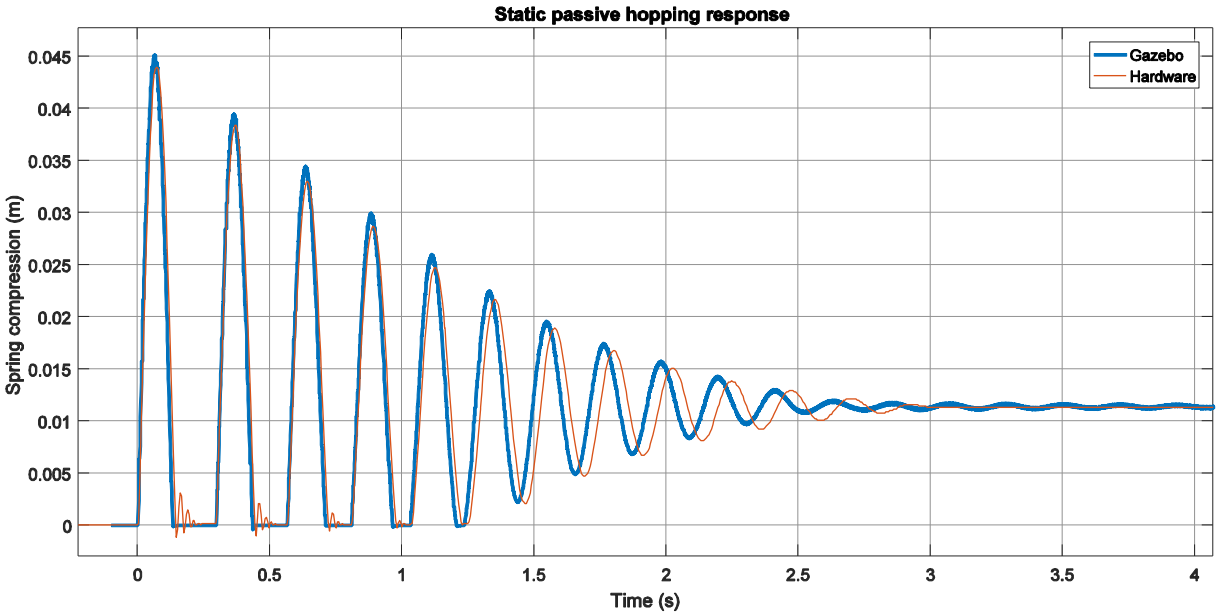


Figure 3-40. Simulation-experiment comparison of static damped hopping.

3.6.2 Hip PID control

In this experiment, the body was held still above the ground and the only thing controlled was the hip joint. We tested for a 7500 counts (26 degrees) step input and the response, real and simulated, is presented in Figure 3-41.

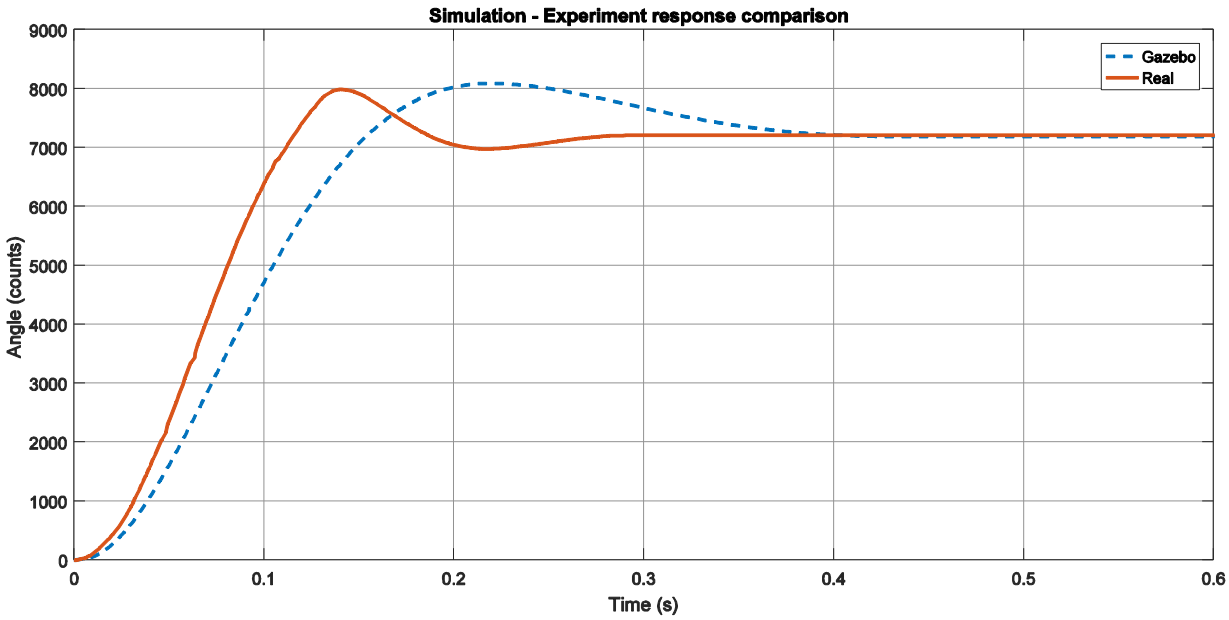


Figure 3-41. Simulated and actual response to a 7500 counts step input.

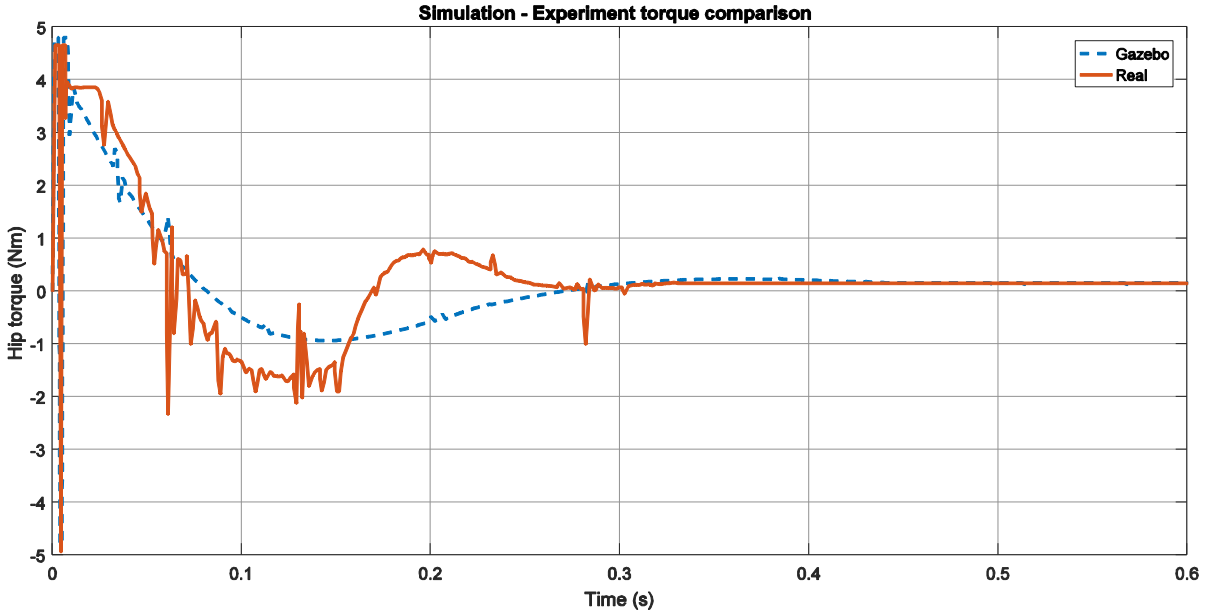


Figure 3-42. Simulated and actual torque requirement.

Considering that such applications in legged robots demand extremely low swing time, i.e. around 0.2 - 0.4 seconds, there is really no point in using an integral term. This is because either it will be of small value and will have no time to load and produce an observable result or it will be of large value and will cause instability and oscillations. So in fact, a simple PD was employed. An estimation of the necessary proportional and derivative gains was taken using a linearized leg model (Figure 3-43), around the vertical position, as follows.

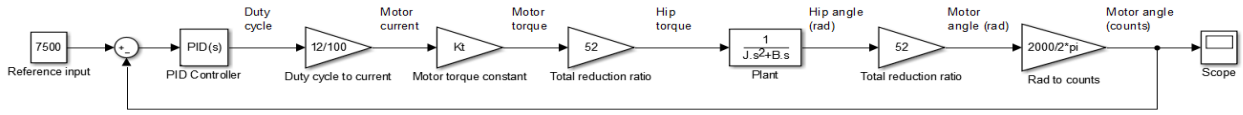


Figure 3-43. Linearized leg model with PID control block diagram.

That model includes the linearized, around the vertical position, pendulum transfer function as plant:

$$G_P = \frac{1}{Js^2 + Bs} \tag{3-7}$$

The following parameters were used:

Table 3-4. System parameters.

Parameter	Value
Reduction ratio	52
J (kg·m^2)	0.0241
B (N·m·s)	0.005

The closed loop transfer function is:

$$G_{CL} = \frac{G_C G_P}{1 + G_C G_P} \rightarrow$$

$$\text{Characteristic equation: } 1 + G_C G_P = 0 \rightarrow$$

$$s^2 + \frac{B + \frac{12}{100} K_T 52 \frac{2000}{2\pi} 52 K_D}{J} s + \frac{\frac{12}{100} K_T 52 \frac{2000}{2\pi} 52 K_P}{J} = 0$$

Therefore:

$$2\zeta\omega_n = \frac{B + \frac{12}{100} K_T 52 \frac{2000}{2\pi} 52 K_D}{J} \tag{3-8}$$

and

$$\omega_n^2 = \frac{\frac{12}{100} K_T 52 \frac{2000}{2\pi} 52 K_P}{J} \tag{3-9}$$

where ω_n is the undamped natural frequency and ζ the damping ratio. When ζ is equal to 1 we have a critically damped system, which is the fastest without overshoot [21]. A value that is frequently used in such applications is 0.7, which offers a good rise-settling time relation. In

our case there is no need to completely eliminate overshoot as it produces a faster response, so we selected ζ equal to 0.6. We also set a settling time value of 0.35 s.

$$t_s = \frac{4}{\zeta \omega_n} = 0.35 \rightarrow \omega_n = 19.05 \tag{3-10}$$

Using (3-8), (3-9) and (3-10), the control gains are found to be:

$$\begin{aligned} K_P &= 0.003368 \\ K_D &= 0.000211 \end{aligned} \tag{3-11}$$

This approximation is really accurate. It was experimentally confirmed that the gains that produced the best results were:

$$\begin{aligned} K_P{}' &= 0.003 - 0.0035 \\ K_D{}' &= 0.0002 - 0.00025 \end{aligned} \tag{3-12}$$

Of course their actual value was affected by many parameters, like those we mentioned earlier about the PID node frequency and the Tiva transmission rate. Also, beyond these values, any further increase of Kp would not produce any visible results as the applied torque was already clipped due to saturation (Figure 3-42). An increase of Kd would again lead to instabilities and oscillations, as we discussed earlier. An overview of the PID control simulation and experiment is given in Table 3-5 and Table 3-6.

Table 3-5. Hip PID control simulation parameters.

Parameter	Value
Real time update rate	100
Step time (s)	0.001
Real time factor	0.1
PID control loop rate	1000
Kd	0.003
Kp	0.0002

Table 3-6. Hip PID control experiment parameters.

Parameter	Value
Defined PID control loop rate (Hz)	8000
Achieved PID control loop rate (Hz)	~1000
Tiva transmission rate (Hz)	15000
Kd	0.003
Kp	0.0002

The gain values of (3-11) could probably increase if path planning methods were employed, because smaller errors would be created during the motion. These increased gains would provide better robustness and rigidity while the leg is still.

3.6.3 High level control

Additionally to those previous simple experiments, and in order to ensure system functionality, we used a high level controller, similar to the one already used in our lab's first monopod robot [8].

This algorithm is a simple one. When the robot is on flight phase it repositions the leg in a predefined angle of attack, or touch-down angle. After it lands and transits to stance phase, a torque is applied to the hip joint in order to move forward and repeat the cycle. The system structure appears in Figure 3-44.

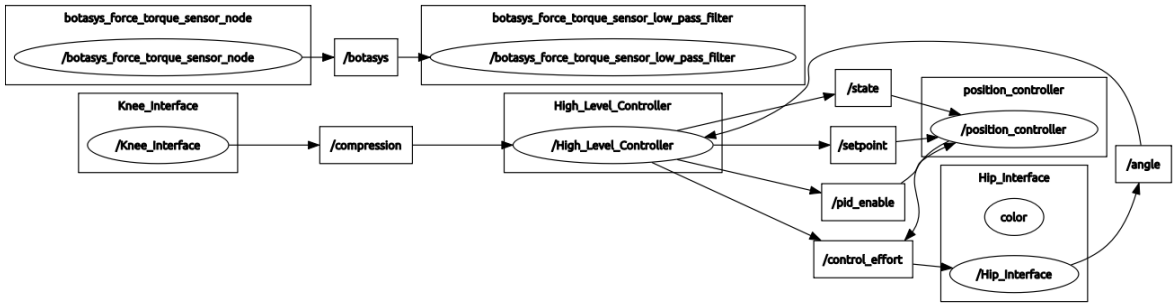


Figure 3-44. Control system structure.

We run the experiment twice, with and without the force sensor. Both cases are presented in Figure 3-45 and Figure 3-47.

Table 3-7. High level controller experiment without force sensor – parameters.

Parameter	Value
Compression for stance (mm)	4
Touchdown desired angle (counts)	870
Stance phase torque (Nm)	1.8
Kp	0.0038
Kd	0.00024

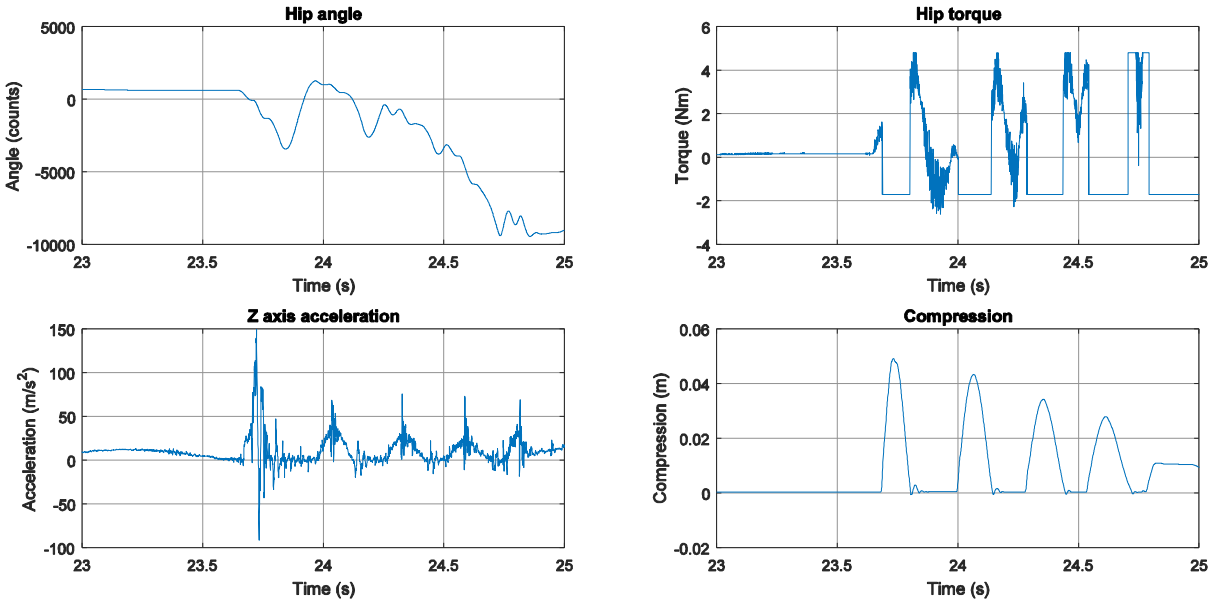
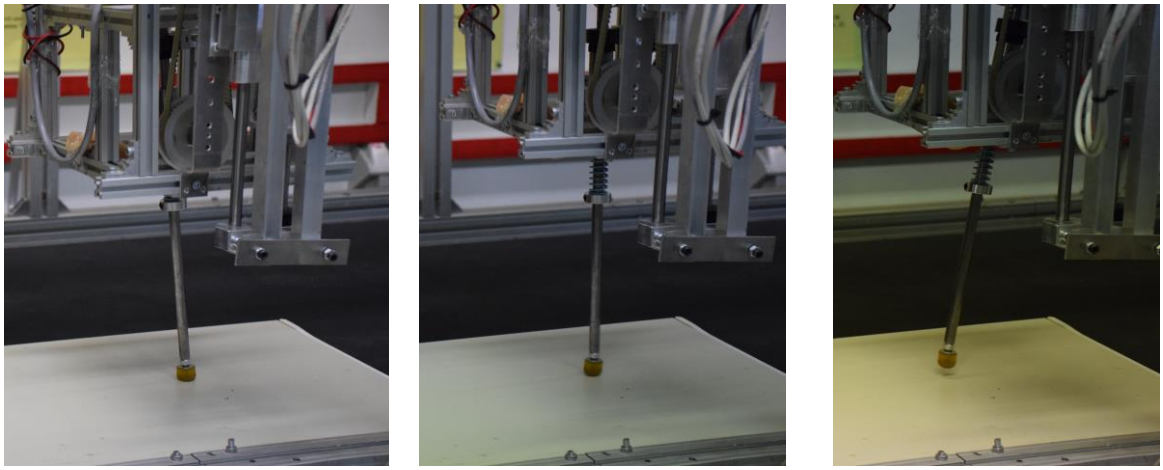


Figure 3-45. High level controller experiment without force sensor.



(a) **(b)** **(c)**

Figure 3-46. Experiment pictures - (a) touchdown, (b) midstance, (c) takeoff.

One can easily see that the leg stops moving after approximately four strides. This issue is quite complex and there might be many reasons for not achieving a stable gait. One of them is the difficulty in controlling the initial conditions, i.e. of the translational velocity and drop height. The latter was 0.08 m (counting from the toe), but there is no easy way to calculate and apply a certain desired velocity, neither repeat it every time. In any case, that velocity and height should be equal or slightly greater than those desired [32]. For example, for desired apex height of 0.32 m and forward velocity of 0.5 m/s, the corresponding initial values should be around 0.33 m and 0.6 m/s respectively.

Moving on to other parameters, after testing several combinations we selected those in Table 3-7. This application, where the leg is moving towards negative values when commanded to reposition in contrast to the PID control experiment we run previously when

the leg was static, allows for utilization of slightly larger gain values than before. The values of stance phase torque and the compression in which the controller considers it has entered the stance phase were also experimentally configured. For larger torque or smaller compression, the leg would simply slip and collapse on itself. That is also what would happen for greater touchdown angles, where the torque provided was not enough to push the body forward; with the motor at its maximum torque. It should be stated however, that this combination is one of the several that might work in a similar manner and could definitely use some refining.

Table 3-8. High level controller experiment with force sensor – parameters.

Parameter	Value
Compression for stance (mm)	4
Touchdown desired angle (counts)	2500
Stance phase torque (Nm)	4
Kp	0.004
Kd	0.00026

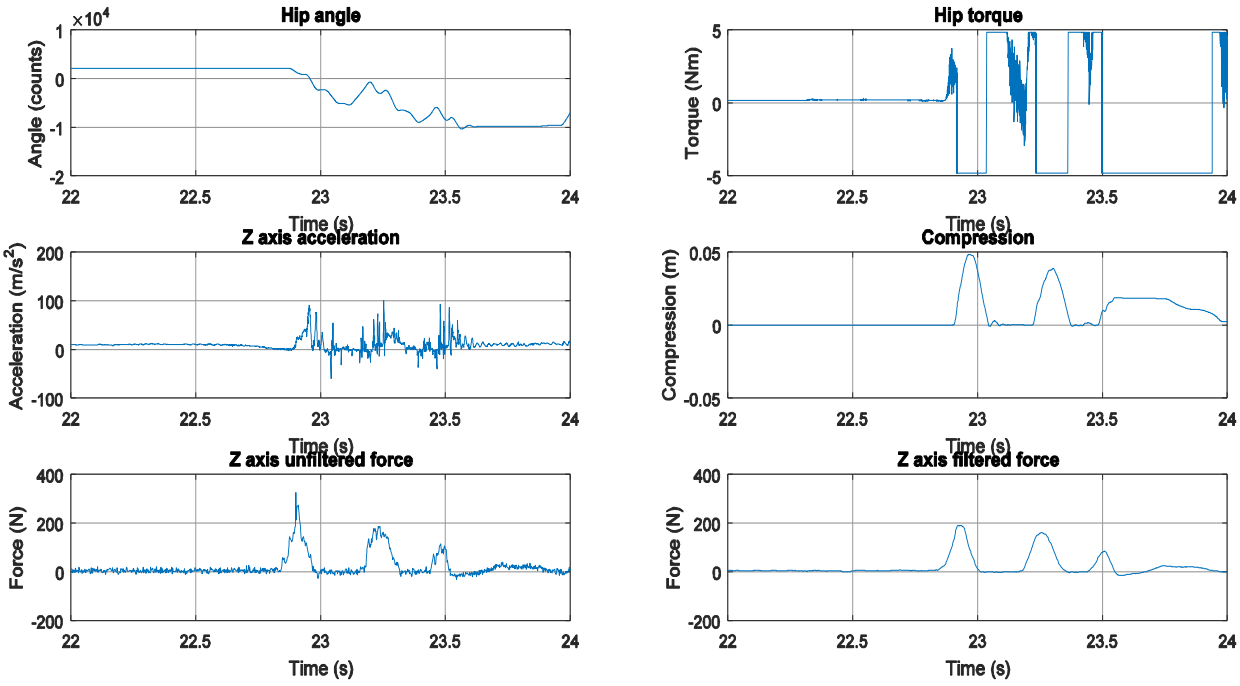


Figure 3-47. High level controller experiment with force sensor.

With the addition of the force sensor, the leg is getting somewhat heavier, which is depicted in Figure 3-47. There, we can observe that there are only have two strides instead of four. That might have something to do with the silicon cover that acts like an absorber. For this reason, the gains as well as the stance torque were increased, with the other parameters remaining the same. We can also see the filtered and unfiltered measurements along the

vertical axis. Peak impact force was measured about 200 N, or about 2.5-3 times the robot's weight.

4 Treadmill

4.1 Setup description

Most of the robots developed in the Control Systems Laboratory are legged and especially quadrupeds or monopods. The salient characteristics and the main design influences come from nature itself (biomimetics). Since we are focusing on resembling nature and its features, e.g. high speed, balance, obstacle detection and avoidance, it only makes sense that those features should be tested in an environment that can simulate nature conditions. Considering that it would be unwise and even dangerous to conduct the first experiments outside the laboratory, the most common solution is to use a treadmill, with inclination capability. That requires a support system, as well as a control system able to set and control velocity and inclination. In addition, a treadmill ensures the security of both the robot and the researcher and facilitates experimentation.

The treadmill currently installed in our laboratory is 6 meters long, and actuated by two 3-phase induction motors. The first motor (model MS 100L 2-4, XIUSHI) drives the belt's main pulley achieving a maximum running velocity of 12.6 m/s. The second motor (model FC80-4, Electro Adda) actuates an endless screw and a rack-pinion system that sets the treadmill's inclination to the desired value (maximum angle of 20 degrees). Both are driven by inverters; an EMERSON M200-022 model for the belt's motor, and a SIEMENS SINAMICS G110 for the inclination motor.

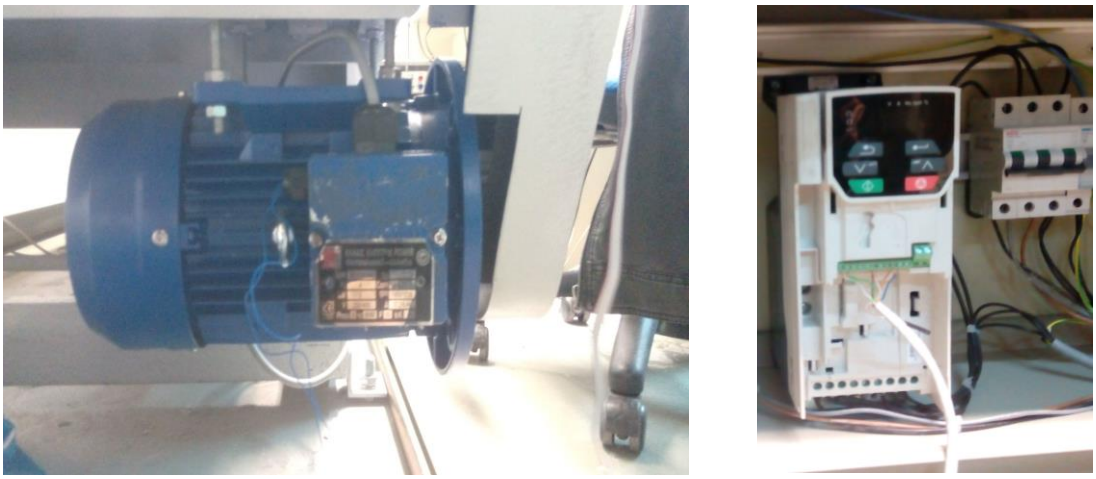


Figure 4-1. MS 100L 2-4 motor and M200-0 inverter.

Figure 4-2. FC80-4 motor and SINAMIS G110 inverter.

4.2 Electronic system

4.2.1 Previous setup

In this setup's previous version, a custom combination of magnets and a Hall effect sensor were used to measure the rotational velocity of the belt's drum. Specifically there were 8 magnets placed circularly, at 45 degrees apart. When a magnet would pass in front of the Hall effect sensor, a voltage pulse would be generated. The pulse was received and processed by an AVR ATmega16A microcontroller. The same microcontroller handled the transmission of the desired velocity to the inverter. It communicated with a central computer, where the loop was closed, with a serial communication protocol, RS-232. However to transform the microcontroller's output to RS-232 signal, readable by the computer, a signal transformer such as the MAX 232 by Texas Instruments had to be used. For the inverter-microcontroller's communication, for sending the command, a digital to analog converter (DAC) was used (AD7302 by Analog Devices). The same circuit was employed for the inclination control system. Finally for the selection of motion direction bipolar transistors were utilized [17].

The control loop of the experimental setup was realized in Simulink (model-based control), which calculated the necessary command and transmitted it to the microcontroller.

This system required three different supply channels, one at 5V and two at 11.42V precisely to supply the operational amplifier. This voltage had to be supplied using a dedicated power supply, something that was not always available or easy to find. Also the Simulink model was not very user friendly and easy to use. For those reasons, and the inability to incorporate ROS, a redesign was imposed.

4.2.2 Redesign and current setup

Let us first examine the system specifications. Both motors, for velocity and inclination, are controlled by an inverter. Those inverters modify the supply current's frequency, and therefore modulate the operating speed as follows:

$$n = \frac{120\,f}{p} \tag{4-1}$$

where f stands for the supply frequency and p for the number of poles.

The method to control the inverters is quite simple. According to Figure 4-3 that follows, the inverter offers several signal inputs and outputs called terminals. Those connections alone can fully control and define the operation of the machine, of course under the condition that it is supplied.

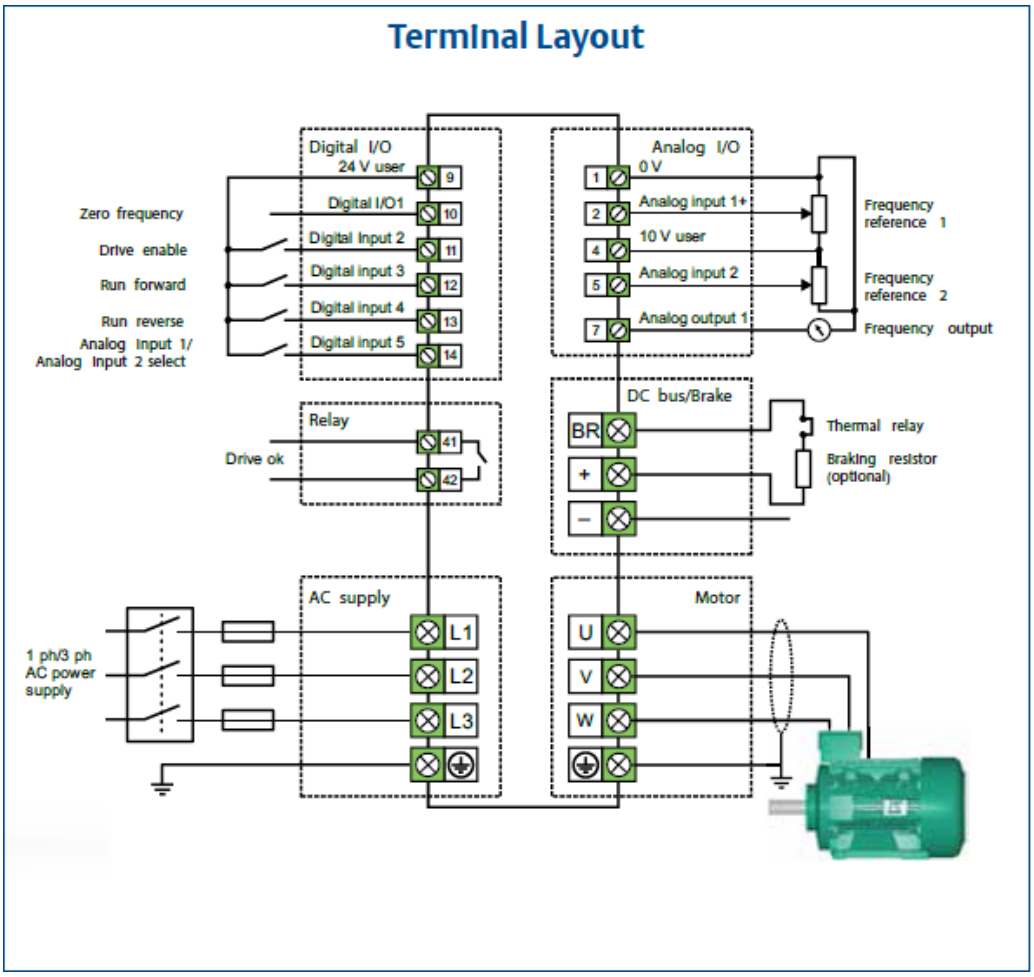


Figure 4-3. Unidrive M200 velocity inverter terminal layout.

When on *terminal control mode*, three connections are required at minimum. System activation using *enable* (terminal 11), *selection of motion direction* (terminal 12 or 13) and *frequency reference* (terminals 1 and 2). The order in which those connections were

mentioned corresponds to the suggested procedure one has to follow while activating the system according to the manufacturer. Specifically, for the motion to be activated, connection of both the enable and the direction terminal is required. When this occurs and there is no frequency reference, the treadmill will start running on a predefined frequency of about 7 Hz, coded into its memory. So in order for the system to be fully functional through a central computer, there must exist a way for all those connections to be activated electronically.

For those terminals to be connected (*enable* and *direction*) a 24V voltage is required. The inverter provides such a voltage (terminal 9), so all one has to do is short-circuit terminals 9-11/13 or 9-12/13 (depending on the desired direction). For the frequency reference a simple analog signal 0-10V (terminal 2) is required, as well as the ground (GND, terminal 1).

Additional specifications are inserted in the design, such as the measurement the angular speed of the belt's drum. Having already decided to use an incremental encoder for this purpose, the need to read its output emerges.

Another point that needs to be taken into consideration during the design of the electronics and the control system is that the treadmill, much like the robots we create, should be structured in a way that it is possible to be run by a common control center, maybe even remotely through the Internet. Since the robot's and the treadmill's function are so closely tied, their control systems should be compatible. This means that there should be a capability for data exchange, and maybe even intervention of the one control system to the other. Finally there are safety limitations and specifications, like in every system. The treadmill's operation should be able to stop both by command and manually, if something goes wrong either in the mechanical subsystem or in the control algorithm.

To summarize, the design specifications are:

- Maximum velocity of 10 m/s
- Incremental encoder feedback
- Control system compatible with the monopod or quadruped
- Function controlled by a computer
- Safety precautions - switches

Again, the basic decision regards the sensory system and specifically how the encoders will be read. We choose to use the Tiva microcontroller, because of the know-how that already exists and the subsequent ease of implementation, and of course because of the characteristics we have already mentioned.

The inverter receives a 0-10V analog signal and translates it into an operating frequency according to the frequency range defined in its memory. The lowest frequency would occur for a 0V input and the maximum for 10V. The maximum frequency currently defined is 50 Hz.

Therefore, we need to ensure that the analog signal can drive the treadmill to velocities up to 10 m/s, according to specifications.

Let us first examine the transmission until the main belt's drum. Right after the motor, which has no gearbox of its own, there is a belt drive (Figure 4-4), which transmits motion to the treadmill's level, but also reduces the velocity, with a reduction ratio of 1.524. Considering the drum's diameter is 245 mm, the velocity relation to the operating frequency is described by (4-2), not considering load and slippage effects:

$$V = \frac{\frac{120\,f}{4}}{1.524} \cdot \frac{\pi}{30} \cdot \frac{D}{2} = 0.2525\,f \tag{4-2}$$

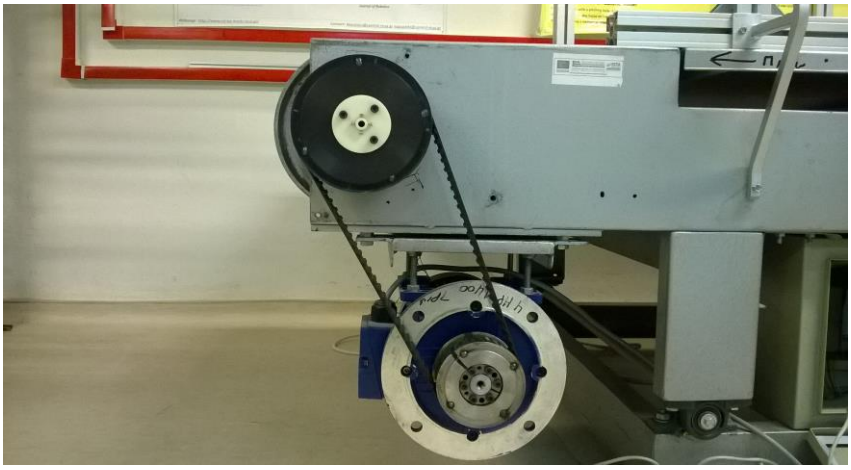


Figure 4-4. Belt drive.

Using (4-2) it is easy to observe that for a 50 Hz operating frequency, a velocity of 12.6 m/s is obtained, while the maximum desired speed is reached with f = 40 Hz. Of course this motor can safely operate up to 60 Hz, according to the manufacturer, so even larger velocity values could be achieved, if it was so desired. Also if for some reason the output analog signal cannot reach the required value, the maximum operating frequency could be defined to 60 Hz in order to change the voltage-frequency relation. This way the same voltage value would be matched to a higher frequency (e.g. 10V would correspond to 60 Hz instead of 50, and same holds for the rest of the values). In any case, the design should result in a properly modified signal, which shall lead to the exploitation of the whole range of the treadmill's capabilities.

This signal can be produced by a PWM pulse. Specifically, in this form of signal, a pulse is modulated so as to be high for a certain width, a percentage of the signal period. That ratio, pulse width to period, is called duty cycle. Because of the extremely high modulation frequency, which is also a parameter, the result is the average voltage. For example, for a signal with 5V amplitude and 50% duty cycle the result would be a 2.5V signal.

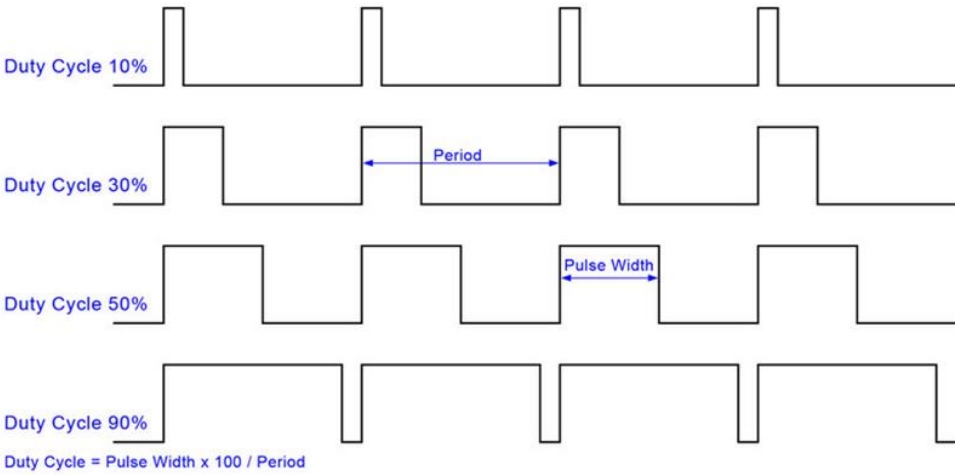


Figure 4-5. Pulse width modulated signal.

Despite the high modulation frequency and the ostensibly steady signal produced, and because an amplification of this signal will follow, it is common to insert a low pass filter and stabilizer between the source and the amplifier. This filter attenuates high noise frequencies and normalizes the signal, with a small repercussion on its amplitude. The most common choice for such a filter is a passive RC circuit (Figure 4-6).

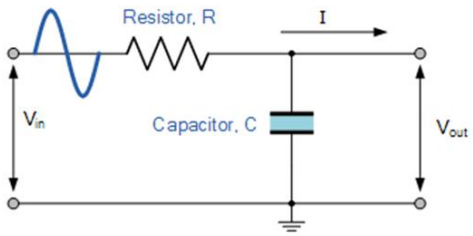


Figure 4-6. Passive RC low pass filter.

The reactance of the capacitor is given as:

$$X_C = \frac{1}{2\pi fC} \tag{4-3}$$

where C stands for the capacity and f for the signal frequency. The circuit's total impedance is calculated as:

$$Z = \sqrt{R^2 + X_C^2} \tag{4-4}$$

Finally, using a resistive potential divider, the output voltage amplitude is given by:

$$\|V_{OUT}\| = \|V_{IN}\| \frac{X_C}{\sqrt{R^2 + X_C^2}} \rightarrow \|V_{OUT}\| = \|V_{IN}\| \frac{X_C}{Z} \tag{4-5}$$

Therefore in our case, where the amplitude of the PWM pulse the Tiva board produces is 0-3.3V, and for filter parameters 4700 Ω και 47 nF for the resistor and the capacitor respectively, the output voltage will be:

$$V_{OUT} = 0.99V_{IN} \rightarrow V_{OUT,\mathrm{MAX}} = 3.268\ V \tag{4-6}$$

Next, for the purpose of signal amplification till the desired value of 10V, an operational amplifier was utilized, and specifically the LM358N by Texas Instruments (Figure 4-7). Such types of setups require a supply larger than the amplified output, in our case 12V, as well as an additional resistor circuit, like the one presented in Figure 4-8.

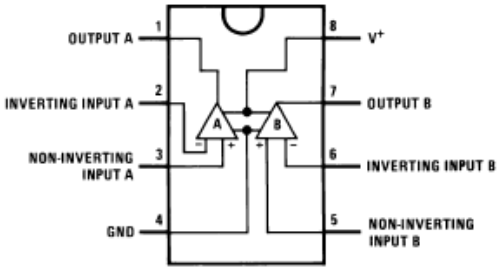


Figure 4-7. Operational amplifier LM358N.

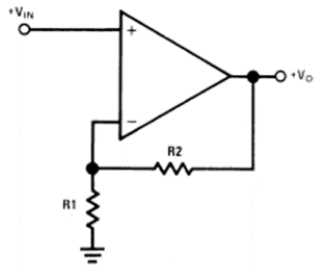


Figure 4-8. Amplifying resistor circuit.

The amplification gain is defined in the equation (4-7):

$$K = 1 + \frac{R_2}{R_1} \tag{4-7}$$

In our case, in which the amplification gain is equal to 3, the resistors are selected as shown in Figure 4-9:

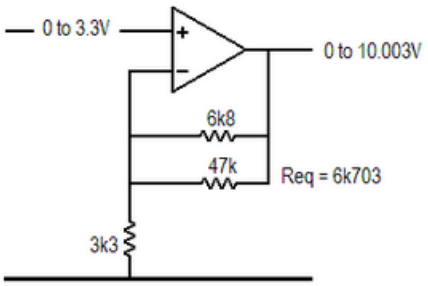


Figure 4-9. Resistor selection.

According to (4-7), we obtain:

$$K = 1 + \frac{6.703}{3.3} = 3.0312 \tag{4-8}$$

The amplified average PWM signal can be fed directly to the inverter. For the whole setup to be fully controllable by a control station, some additional elements have to be incorporated so as to allow the user to initiate and stop the motion. This task was carried out using three relay modules (Figure 4-10). These elements require a 5V supply and a logic level signal (HIGH or LOW). According to this value, the relays open or close a circuit. Using these parts, and a digital output coming from the Tiva board, it is possible to control the short-circuit of the *enable* and *direction* terminals mentioned earlier.

Figure 4-10. Parallax relay module.

Relays like the one in Figure 4-10 include three inputs-outputs. Let us consider Figure 4-11. At the middle contact, we connect the first end of the circuit that has to be controlled. The other two contacts are marked as normally open (NO) and normally closed (NC). Depending on the application the user has to select in which he shall connect the other end of the circuit. For example in Figure 4-11, the circuit is connected to the normally open contact. That means that in order for the circuit to close, one needs to send a HIGH logic level. If the circuit was connected on the normally closed contact, a HIGH logic level would open the circuit instead.

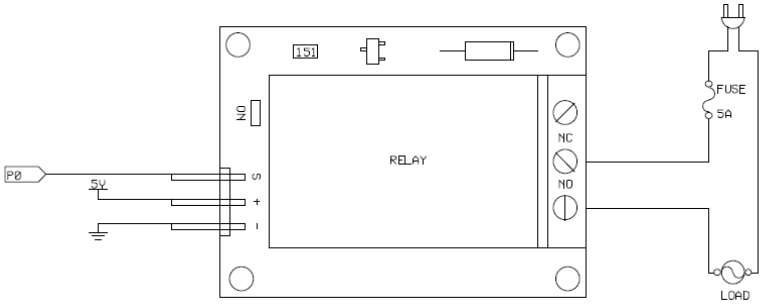


Figure 4-11. Application example with relay module.

The three relays used in our case control the enable, run forward and run reverse terminals. The first was connected exactly as in Figure 4-11. The enable terminal is normally disconnected, until given the proper command, as a safety precaution. It was critical to decouple this certain terminal's connection with any other in order to be able to stop the motion in case of emergency. For the same reason there is a manual switch right after the relay, just

in case there is a malfunction in the control system and we cannot shut down the motion by command. The whole circuit is shown in Figure 4-12.

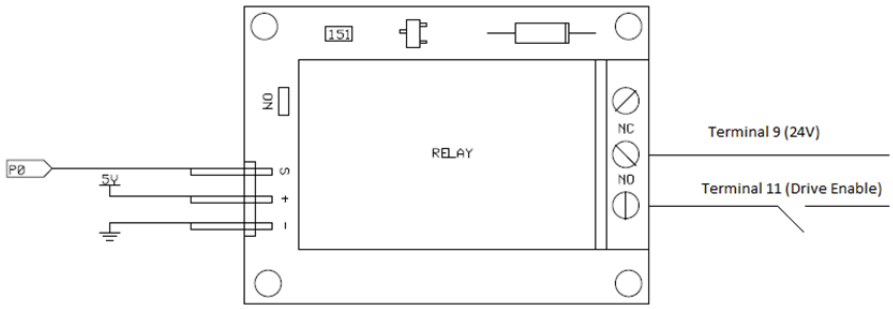


Figure 4-12. Activation system (terminal Drive enable).

The corresponding system for the selection of motion is slightly more complicated, because there was an additional safety precaution. Since it is not clarified in the manufacturer's manual and as reason dictates, the two direction terminals should never be short-circuited simultaneously, because of the unknown and potentially harmful effect this might have. We could short-circuit the terminals using a single relay like before but one of them would be connected on the normally closed contact, so we would have a default direction of motion and that is not desired. Therefore we chose to use a double relay module (Figure 4-13) with a small extra cost (6 euros instead of 3 the single costs).

Figure 4-13. Double relay for motion direction selection.

Once again we need a 5V supply, however this time there are two enabling signals. In order to satisfy the safety precaution that was mentioned before, these relays will be connected in series, as shown in Figure 4-14. The system works as follows. We shall refer to the relays as R1 and R2. The 24V line is connected to the central contact of R1. At start, when no signal is sent to the module, this line is directed through the NC contact of R1 to the NC contact of R2 which is not connected to anything, so there is no motion. When we set R1 to HIGH, the 24 line is redirected through its NO contact to the run forward terminal, regardless of what signal is sent to R2, since its central contact is disconnected. That protects the system from a possible malfunction. To run the treadmill in reverse, we need to send a LOW signal to R1 and HIGH to R2. It is obvious that there is no signal combination that leads to both terminals being short-circuited simultaneously.

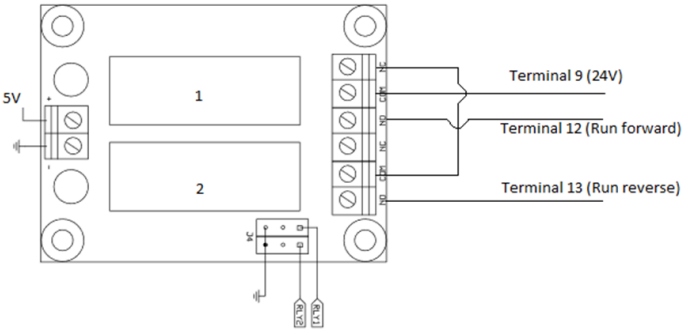


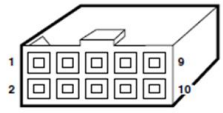
Figure 4-14. Motion direction selection system (terminal run forward and reverse).

The incremental encoder is of type HEDL-5540, with a line driver to reduce the noise. In our case this was not needed, since the signal did not have to travel too far and no noice was observed. The encoder produces 500 counts per revolution, augmented to 2000 when in quadrature mode. It has the same connections as the previous, 5V, GND, A, B channels and optionally the index, used to count the total number of revolutions. The encoder and its connectivity are presented in Figure 4-15.

(a)

10-PIN CONNECTOR

NO.	COLOR	PARAMETER
1	BROWN	NC
2	RED	V_{CC} (+ 5 V)
3	ORANGE	GND
4	YELLOW	NC
5	GREEN	$\overline{A}$
6	BLUE	A
7	VIOLET	$\overline{B}$
8	GREY	B
9	WHITE	$\overline{I}$ (INDEX)
10	BLACK	I (INDEX)



(b)

Figure 4-15. (a) HEDL-5540, (b) HEDL-5540 Pinout.

Finally, all those different pieces of equipment had to be properly supplied. The relays, as well as the Tiva board and the encoder require 5V, while the operational amplifier needs 12V. To use a single power supply, we employed a step down regulator, the same used on the monopod, i.e. Polulu D24V60F5. Generally such parts are selected in high power applications, as signified by their maximum current handling capacity (6A), but its use is simpler than that of a simple voltage regulator; also it was available in our lab. Therefore it was preferred compared to other solutions.

4.2.3 Construction

The final electronic subsystem schematic is presented in Figure 4-16.

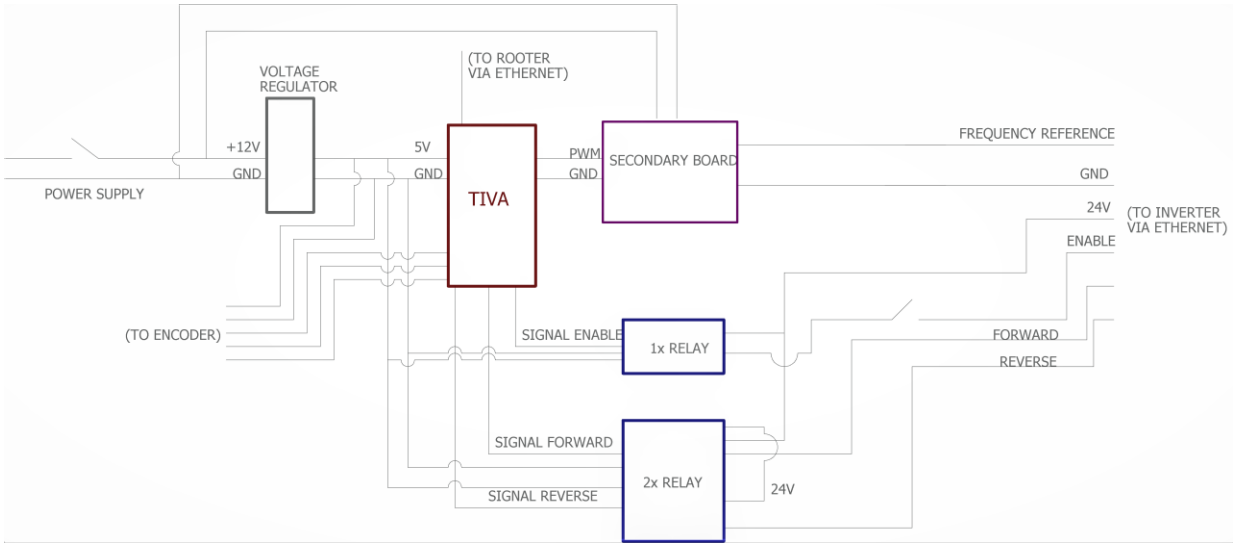


Figure 4-16. Electronic subsystem schematic.

The secondary board includes the PWM filter and amplification circuits, as shown in Figure 4-17.

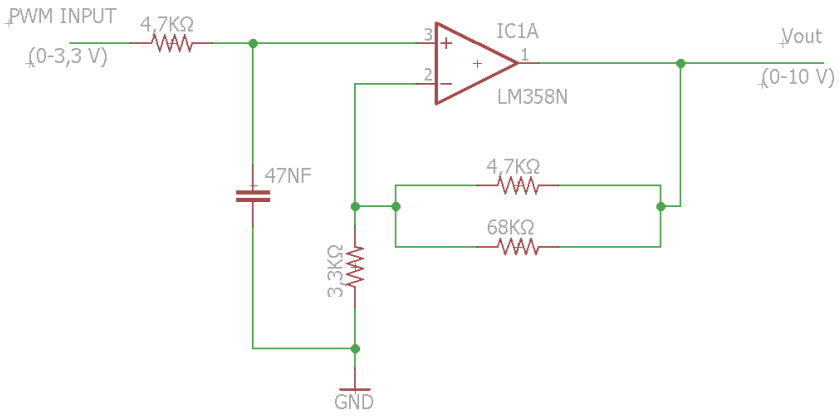


Figure 4-17. Secondary board.

All the aforementioned subsystems were connected and placed between two plexiglass sheets as shown in Figure 4-18.

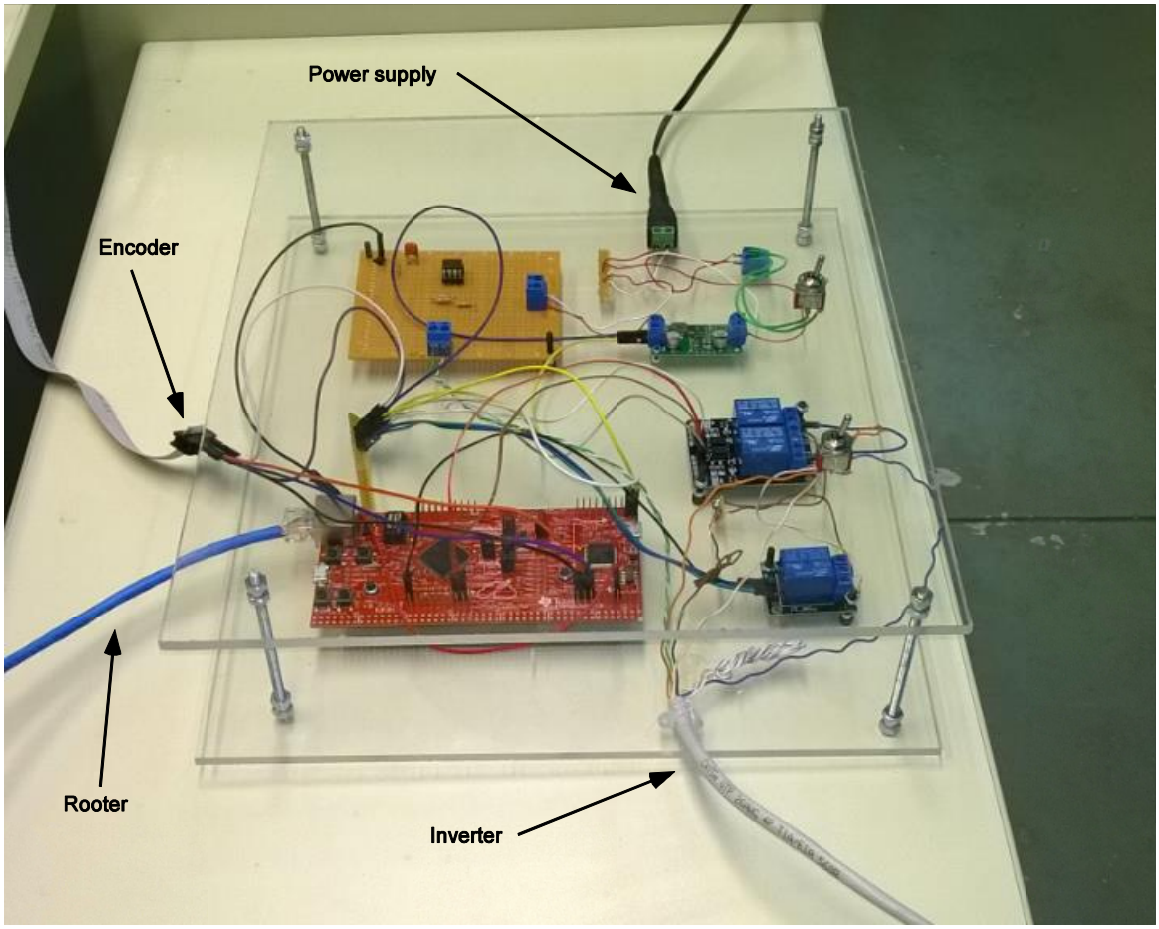


Figure 4-18. Electronic subsystem prototype.

One can observe four inputs-outputs. The blue Ethernet cable connects to the rooter and communicates with the control center, the white cable is the one that connects to the inverter, the flat cable on the upper left leads to the incremental encoder and the black is the power supply. The two switches are also visible above the relay modules, one for the power supply and one for motion activation.

4.3 Control system

The control system in the case of the treadmill is similar to the monopod's. There are again two main pieces of code, one running in Tiva and of course its counterpart running in a control station using ROS.

4.3.1 Tiva

The part of code running in Tiva handles the PWM and QEI module activation, as well as the set up of the UDP communication. The same code is used with some minor modifications in order to send velocity instead of position and handle activation of motion and selection of direction.

Special care has to be taken with the velocity estimation, as it is defined indirectly. Specifically, we define a time lapse in form of a number of clock ticks, over which we want to

sum the measured counts. This number in combination with the set clock frequency defines the duration of velocity estimation. For example if we define the clock frequency to 120 MHz, with a measuring lapse of 40.000.000 ticks, we shall get a new velocity estimation value three times per second. Let us consider the following case. We define the velocity estimation duration equal to 1s. If we request a new measurement precisely at 1s we shall get an X value, the total number of counts in (0,1). If a new value is requested at 1.5s we shall not get the total number of counts measured in (0.5,1.5) but we will receive the same value X as before, until we reach 2s where a new value is calculated. This of course means that there is no point in us asking for new values faster than they are calculated, as we would receive the same value multiple times.

Finally for the code's proper operation, it is again required to define the IP addresses for the Tiva board and the control center as well as the device subnet και device gateway. The send and receive ports also need to be defined.

4.3.2 ROS

Necessary condition for the robot and the treadmill control systems to be able to communicate with ease is to be compatible. Therefore, since all robots use ROS, it was again decided to build the treadmill control system.

The system is similar to the one we already described for the monopod and it is comprised of three nodes. The first, *ros_speed*, is the interface between the Tiva board and the computer. Like before it activates the UDP communication and publishes received measurements on /*state* topic. It also subscribes to the /*control_effort* topic and receives the desired PWM duty cycle command. Obviously one has to define the IP address for Tiva, as well as the send and receive ports.

The second node, *ros_read_vel*, takes as user input the desired velocity and publishes it on the /*desired_speed* topic. Also this node can receive and recognize four key words, *enable*, *cw*, *ccw* and *kill*. To enable clockwise motion, one needs to type the words *enable* and *cw*. Typing *ccw* right after will reverse the motion, and of course *kill* will stop everything.

The third and last node is the PD controller from the *ros_pid* package that was used before. It is the exact same node, except for a small modification. Generally in the PD control, zero error corresponds to zero output. This is something we want to avoid in our case. The velocity would rise to the desired value, then the inverter would receive a zero command and it would start to fall until the error is significant again. That would cause an oscillating behavior around the desired velocity, which of course is not desired. For this reason we modify the control law as follows, using also (4-2):

$$u = K_P e + K_D \dot{e} + \frac{100 f_{des}}{50} = K_P e + K_D \dot{e} + \frac{100 \frac{V_{des}}{0.2525}}{50} \quad (4\text{-}9)$$

i.e. a constant term is added which, when the error (e) is equal to zero, produces the open loop command, the frequency that is required by (4-2). The gains receive and handle any oscillations and differences that might occur.

The cross node communication is presented in Figure 4-19.

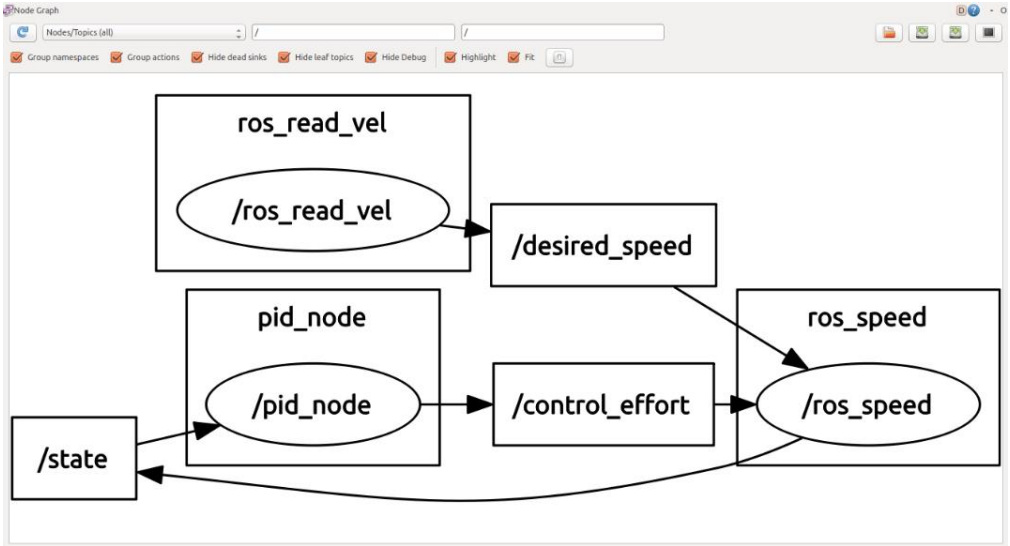


Figure 4-19. Node cross communication as depicted from the rqt_graph tool.

4.4 Experiments

To ensure the setup's proper operation, simple experiments were conducted. The inverter received desired frequency commands. Specifically the first experiment was an open loop definition of the operating frequency. The velocity was gradually increased to 10m/s. In the second experiment we closed the loop and recorded the responses for the simple and augmented PD controller of (4-9). On both occasions the gains were experimentally defined. Specifically, the following parameters were used.

Table 4-1. Treadmill velocity closed loop control experiment parameters – augmented PD.

Parameter	Value
Defined PID control loop rate (Hz)	500
Achieved PID control loop rate (Hz)	499
Tiva transmission rate (Hz)	1000
Tiva sampling frequency (Hz)	15
Kd	100
Kp	10

As one can observe in Figure 4-20 to Figure 4-23, this is a particularly slow system, mainly because of the inverter. It takes about 22 seconds to reach the desired velocity of 10 m/s, even if the command for its augmentation is given momentarily. This happens because the inverter applies a standard fixed rate of acceleration or deceleration (counted in seconds per 100 Hz) for reasons of safety and proper function.

A second point that needs clarification are the oscillations and spikes that appear in the steady state response (e.g. Figure 4-20). These spikes are the result of the sampling frequency as we mentioned earlier. We can observe that with a 15 Hz sampling rate, 15 steps appear per second (Figure 4-21), which confirms the aforementioned on the sampling frequency in Paragraph 4.3.1. As far as the oscillations are concerned, these are probably result from a slight misalignment between the encoder axis and its fixed frame which result in different sums of counts during a second. However, the velocity is in fact steady. This was confirmed both by visual and acoustic observations, while the value was also measured with an optical tachometer and found to be the same with the one appearing on the diagrams.

We can finally observe the difference between the various experiments. The open loop has large differences between the expected and the actual velocity. Using the closed loop we observe the mentioned oscillations in steady state velocity because of the simple PD controller and how the problem was resolved with the augmented control law.

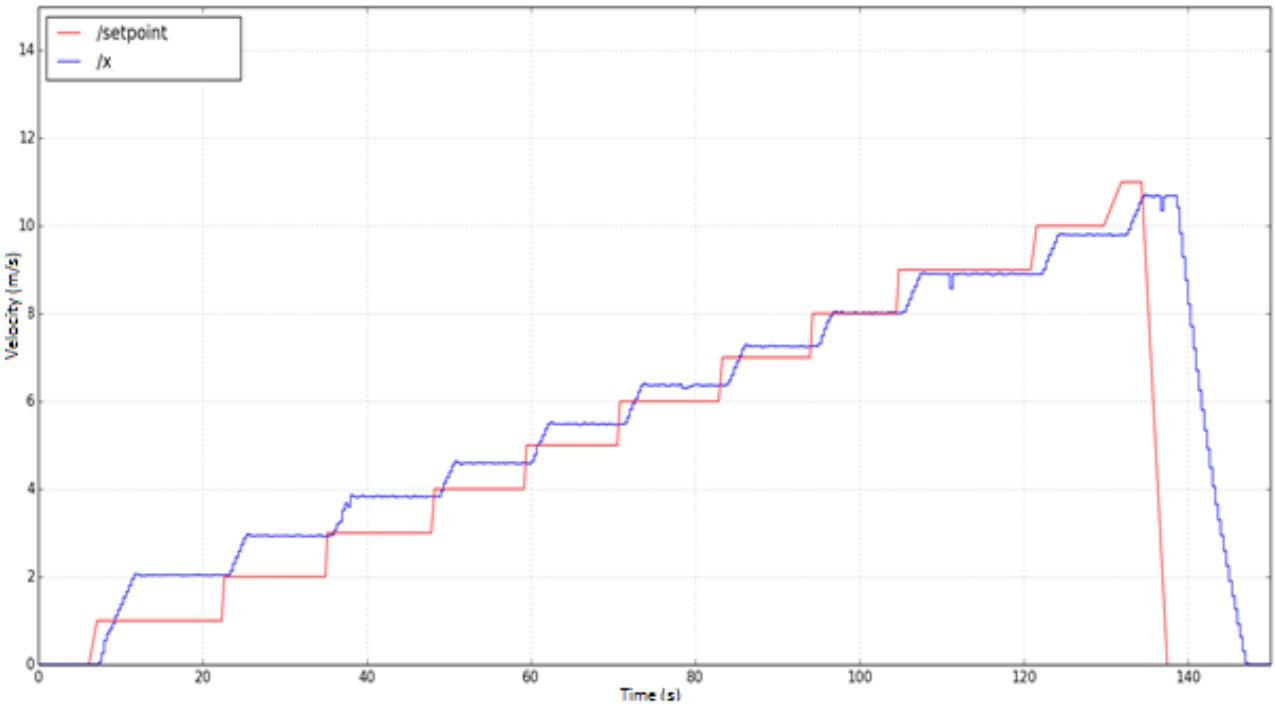


Figure 4-20. Treadmill velocity open loop control experiment.

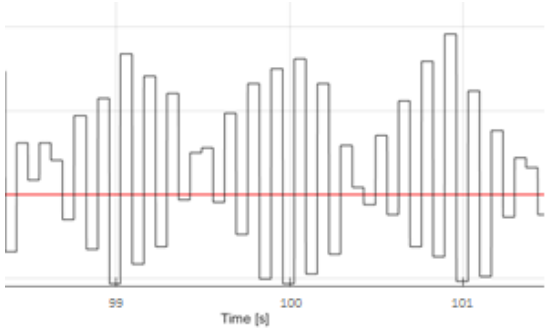


Figure 4-21. Velocity response detail.

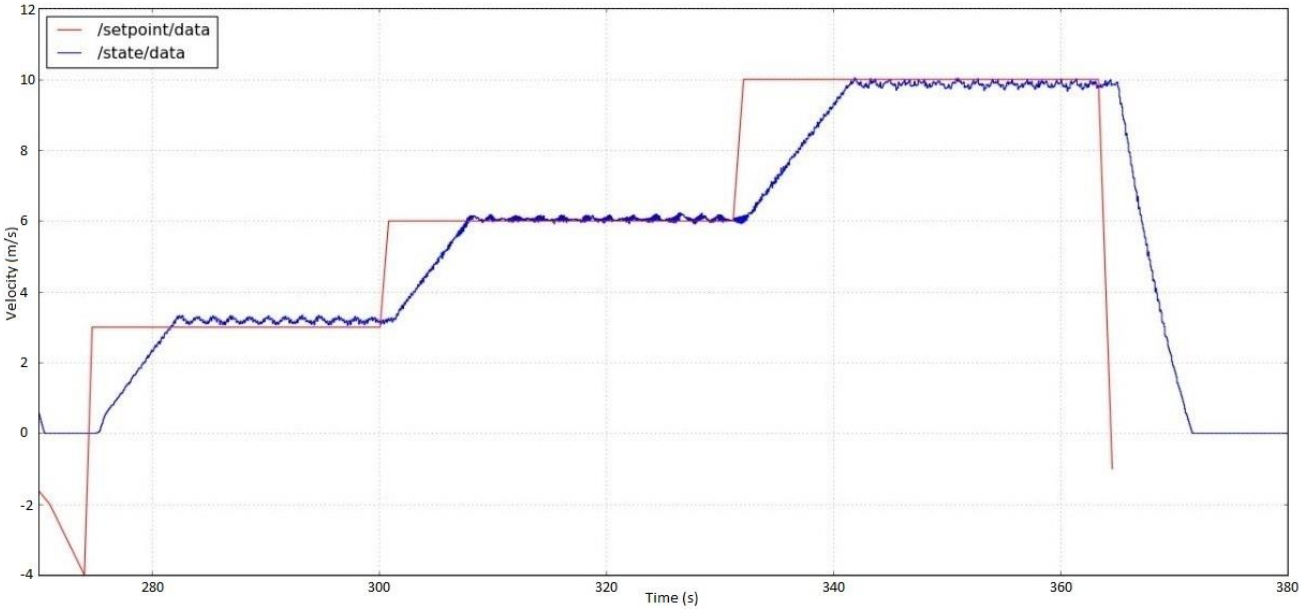


Figure 4-22. Treadmill velocity closed loop control experiment – simple PID.

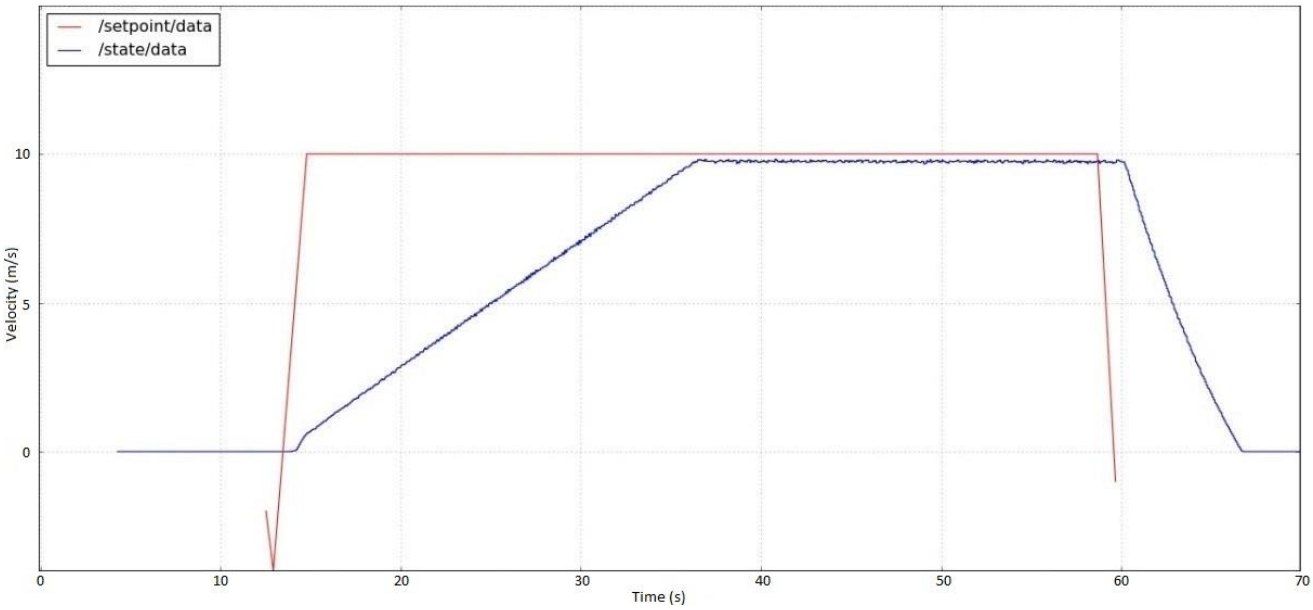


Figure 4-23. Treadmill velocity closed loop control experiment – augmented PID.

5 Articulated monopod design

5.1 Literature review

Ever since the first appearance of legged robots, a large variety of designs, materials and manufacturing methods have been used. In this section we will focus more on these leg aspects and examine some that have been actually used on biped or quadruped robots rather than just monopod experimental testbeds.

One of the first attempts to create a walking robot that could alter its gait is recorded in 1968, with the General Electric quadruped, developed by R. Mosher (Liston and Mosher, 1968; Mosher, 1968). That was a hydraulic actuated robot, with 3 DOF (knee flexion-extension, hip flexion-extension and abduction-adduction). This example is not representative and easily comparable to most modern biomimetic walking robots, as it was 3.3 m tall, 3 m long and it weighted about 1400 kg [16]. Each leg was controlled through a different joystick, requiring 4 different joysticks – and operators – in total. However as we can see in Figure 5-1.

General Electric quadruped robot and compare with following figures, the basic mechanical design principles have not changed dramatically through the years. We notice the similar link design and general structure as, for example, in StarlETH. Of course the manufacturing processes, the materials used and the mathematical analysis behind the design have greatly improved in the last 50 years.

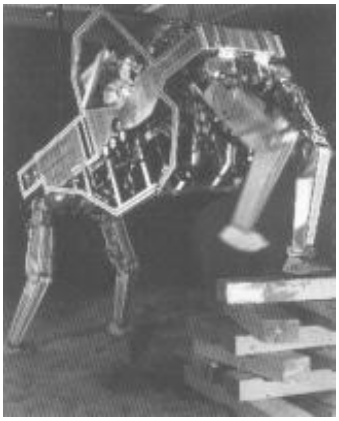


Figure 5-1. General Electric quadruped robot.

MIT's Quadruped (1984-1987) was the robot that probably set the foundations and started the revolution in the legged robotics field that continues today. This robot utilized the control principles and algorithms of Raibert's monopod and it could trot, pace and bound [24]. It was comprised of four 3 DOF legs (hip flexion-extension, abduction-adduction and a prismatic knee joint) and an aluminum frame [23].

Figure 5-2. MIT Leg Lab's Quadruped Robot.

Since those first attempts to create a biomimetic walking robot, countless other quadruped and monopod robots and leg designs have appeared. An indicative example are the Sugoi-Neco legs [28]. Utilization of high tensile aluminum structures is quite common across the literature and up till recently it was almost the only choice available, if one wanted to maintain a low weight with a reasonable safety factor. Examples include ETH's StarlETH [12], Boston Dynamics' Spot and SpotMini, and IIT's hydraulically actuated HyQ [29], [15]. StarlETH and HyQ are actually combining an aluminum frame structure on the femur link and a tube on the shank, an element that we, too, are going to use next. HyQ also specifies that a combination of stainless steel and aluminum alloy (Ergal - 7075) was used.

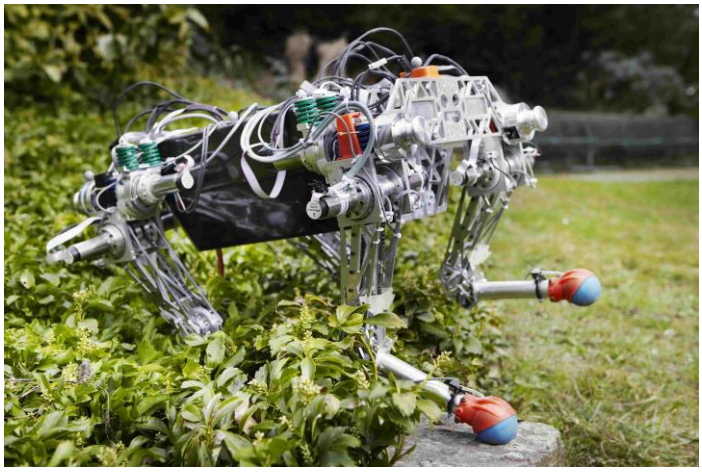


Figure 5-3. StarlETH quadruped robot.

While designing a leg, in terms of reducing the rotating inertia, it is a common practice to place the motors as close to the body as possible (a known exception shall be reviewed below). This of course results in the necessity to transmit motion from the body to the knee, which in most cases is the most distal actuated joint. There seem to be two main approaches in order to resolve the problem. The first is to use a four bar linkage, like in the aforementioned

Sugoi-Neco legs and in the MIT's Cheetah leg [18]. Specifically, the Cheetah robot takes advantage of two concentric custom motors that reduce the inertia and a steel bar to actuate the knee. Tendons are also utilized to reduce structural loads [3].

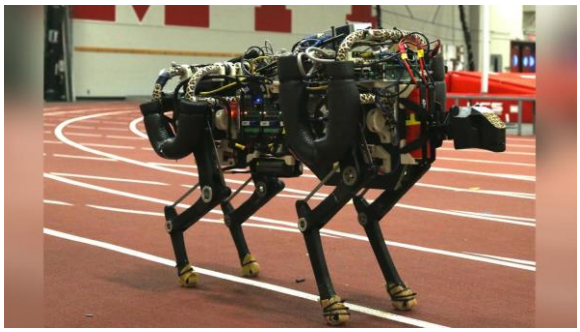


Figure 5-4. MIT's Cheetah robot.

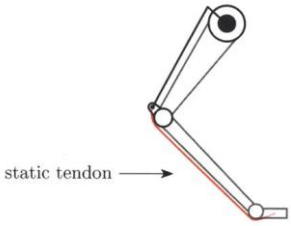


Figure 5-5. Cheetah leg with four bar linkage and tendon.

The second approach is to use a standard torque transmission system, commonly a pulley-belt or a sprocket-chain combination. In most cases, this solution succeeds to keep the weight somewhat lower and if calculated correctly, it could provide added features. For example, in the StarlETH robot, the miniature chain drive that is utilized is designed to fail earlier than the gearbox in order to protect it and provide convenient and cheap maintenance if failure occurs.

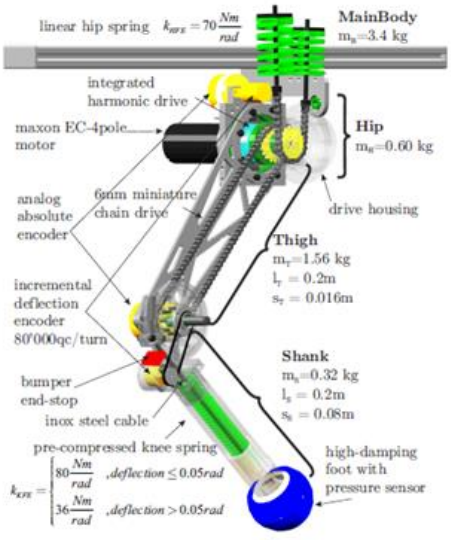


Figure 5-6. StarlETH leg transmission.

Steel cable transmission is also utilized in the Carnegie Mellon's BiMASC's legs. Steel was selected after tests that proved the inability of Vectran and Xylon to sustain the fast repetitive wrapping around a pulley – each for different reasons. Kevlar and other related materials were also considered but because of their abrasive nature (when in contact with itself) it was judged that would wear quickly. This leg also contains an interesting fiberglass plate spring design [11].

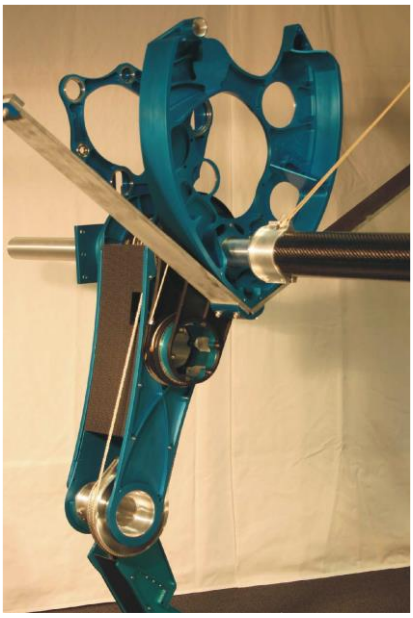


Figure 5-7. BiMASC leg with fiberglass plate springs.

From the aforementioned designs and applications, only the MIT's Cheetah has something new to offer in terms of materials and manufacturing methods. Specifically, as mentioned in [18], the leg is made of a polyester resin shell and polyester foam core. For the manufacturing process two molds are required. The first is undersized (compared to the leg's actual external dimensions) and it is used to cast the foam core. Then, the core is transferred to the second mold, which is dimensioned correctly, the polyester resin is cast and the shell is formed. To fill the mold and make sure that no air bubbles are trapped inside, vacuum resin infusion is utilized. During this process, the part is covered and hermetically sealed, usually with plastic sheets, leaving only two openings. In the first a vacuum pump in connected, and in the second a pot that carries the resin. When the pump is activated the resin is slowly sucked in the part leaving no air bubbles, provided there is a smooth flow, and avoiding accumulation of excess resin in some points.

Carbon fiber is also known to have been used in leg manufacturing. Perhaps the most indicative example is the ATRIAS monopod [10]. Since the robot's design has to be as close to the SLIP model as possible, lightweight carbon fiber tubing has been used, both in the femur and the shank. We can also notice the same fiberglass leaf springs that were used in the BiMASC leg.

Figure 5-8. ATRIAS leg.

Another known innovative quadruped in ANYmal [13]. This robot appeared just recently and it features legs made of carbon fibers and aluminum with 3 DOF on each, enabling full 360 degrees rotation on hip and knee joints. However the most original element is the ANYdrive [34], a custom combination of brushless motor, gearbox, absolute encoder, spring and electronics, all included in a single cylindrical casing. Its extremely reduced weight (only 0.9 Kg) allowed the knee motor to be placed on the knee rather than the body, as usual. This significantly simplifies the design, as there is no need to transmit motion through complex mechanisms. The incorporation of all aforementioned elements in a single module also allows for quick exchangeability of parts in case of failure, instead of time consuming disassembly of motors, belts and pulleys.

Finally, it is worth mentioning some attempts that have been recorded to use 3D printed structural parts, like the MIT's Super Mini Cheetah [2]. However this method, under the current technological limitations, would only apply to small sized robots (8-10 Kg) as it does not provide anywhere near the necessary strength to support a normal sized robot (30-40 Kg) without added reinforcement (e.g. fibers). For even smaller robots, like cockroaches, a new, innovative method that seems to be gaining ground in terms of popularity is SDM (Shape Deposition Manufacturing). It is a method reportedly used by both the MIT [9] and the Florida State University [6] and allows for complex sandwich type structures to be created, like layers of stainless steel and copper. This results in a more biomimetic design, as one can rarely observe limbs or organs entirely made up from a single material layer in nature. Nevertheless, to our knowledge, this method has not been applied yet in large scale projects like ours, and has yet to be developed and expanded before it is.

5.2 Laelaps quadruped robot

Laelaps I is CSL's second quadruped [47]. Designed for velocities up to 10 m/s, it emphasizes on performance rather than energy efficiency. This is the first essential difference compared to the first version.

The other obviously is the transition to an articulated leg and body (spine) design. The first quadruped had a prismatic knee, a revolute actuated hip joint and a rigid torso. In fact the old quadruped's leg is the same leg that we described previously in the case of the monopod robot.

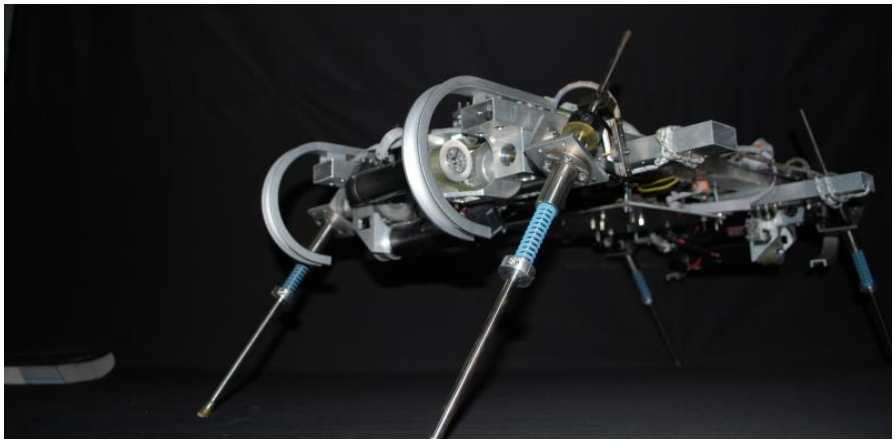


Figure 5-9. CSL's first quadruped robot.

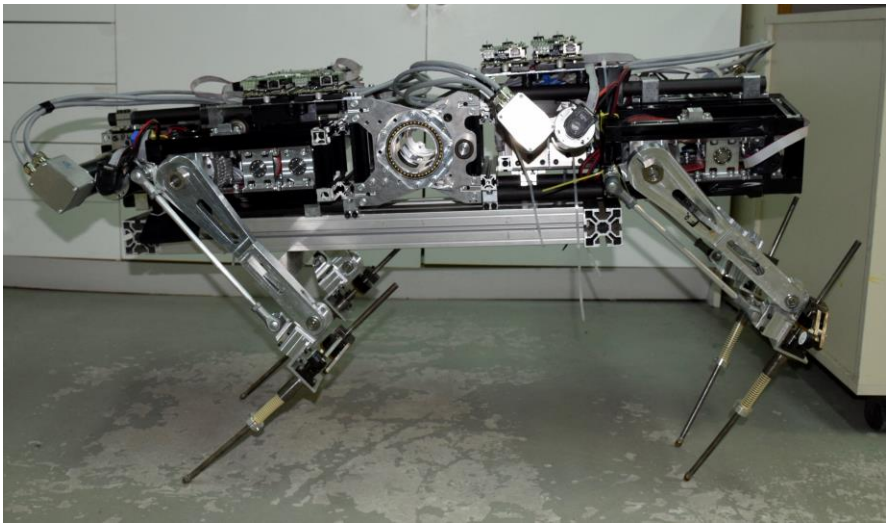


Figure 5-10. Laelaps I quadruped robot.

Current design includes ten Maxon motors for the actuation (two for each leg, one for the spine and one for the tail) as well as an aluminum frame body. The femur is also a new part made of aluminum. Instead of a new shank (lower leg) design, the legs of the old quadruped were utilized, featuring the same springy prismatic joint.

While hip joint is actuated directly, for reasons that we mentioned earlier, that was not an option for the knee joint. A push bar was used instead, forming a four bar linkage. This mechanism is designed to transfer maximum torque (forms a rectangle) when the femur-shank

angle is 40 degrees. This feature shall be transferred to the new design, as well. Finally, the leg has 90^{o} motion width, limited with hard stops on the knee joint.

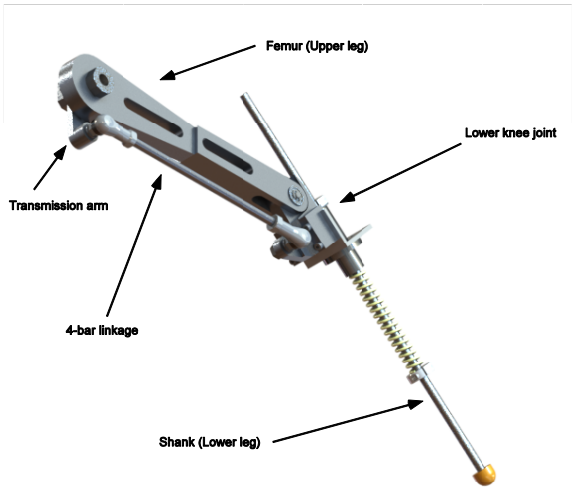


Figure 5-11. Laelaps I leg design.

5.3 Leg Design

Note that in the case of Laelaps I leg, the femur joint was merged with the link design, as it was a compact aluminum part. For the Laelaps II quadruped, the basic design specification and target was to decouple the joint and link design. This way one can easily change each segment's length according to the task at hand. For example, efficient bounding and trotting gaits require different lengths.

5.3.1 Link design

Using carbon fiber tubes as main link parts, like ATRIAS, offers easy exchangeability between pieces of different length while keeping the leg lightweight. In places where one tube cannot bear the load, there is also the possibility to use a second inside the first. Specifically the tubes available in our lab fit perfectly for that purpose as their inside and outside diameters match (30/28 for the outer and 28/26 for the inner tube).

5.3.2 Joint design

The joint parts should be designed so as to safely bind the tubes without damaging or destroying them. As far as the material was concerned a 7075-T6 aluminum was selected, an alloy with yield (430–480 MPa) and ultimate tensile (510–540 MPa) strength comparable to that of many steel alloys.

The knee joint is comprised by two parts, upper and lower. The lower part is a modification of the Laelaps I design. A second protrusion was added to the opposite side of the first, as well as in the transmission arm on the hip, in case we want to use threads to transmit motion to the knee, e.g. Kevlar rope. Since the rope is flexible, in contrast to the pushing rod, it can

only pull so it must be attached on both sides. The tube is bound using a detachable cover and four M4 bolts. Same principles were used for the upper part, only it does not have any protrusions except those to bind it with the lower part. For that connection, the SFL606ZZ ball bearings and 6 mm, 1.1191/C45E steel shaft used on the current design were kept the same. The hard stops at 0-90^{o} were also preserved.

On the hip joint, besides the change that was already mentioned on the transmission arm, the femur link was essentially shrunk and modified in a similar manner with the knee joints in order to adapt the carbon tubes. The only difference is the utilization of M5 instead of M4 bolts because of the increased load that seemed to be exerted in the following simulations in comparison to the knee.

5.3.3 Compliance and transmission

The last things considered were the spring and force sensor-toe placement. A mechanism like the StarlETH leg was examined, with the spring inside the carbon tube, but since it was unclear how well the tube would react on resulting forces, it was decided to place the spring below the carbon tube, using a same mechanism with the old lower leg. A linear bushing is adapted and tied using a custom aluminum clamp. The spring is placed concentrically with a properly shaped shaft. At its lower end, either a rubber toe or a force sensor can be adapted.

Finally, the push rod mechanism remained the same, as there was no reason suggesting a replacement was in order.

5.3.4 Final design

The complete new design is shown in Figure 5-12. It weighs about 300g less than the old (~1000 instead 1300g). The difference is actually increased to 450g if Kevlar rope is utilized instead of the push rod.

The drawings of all parts are included in the CD that accompanies this thesis.

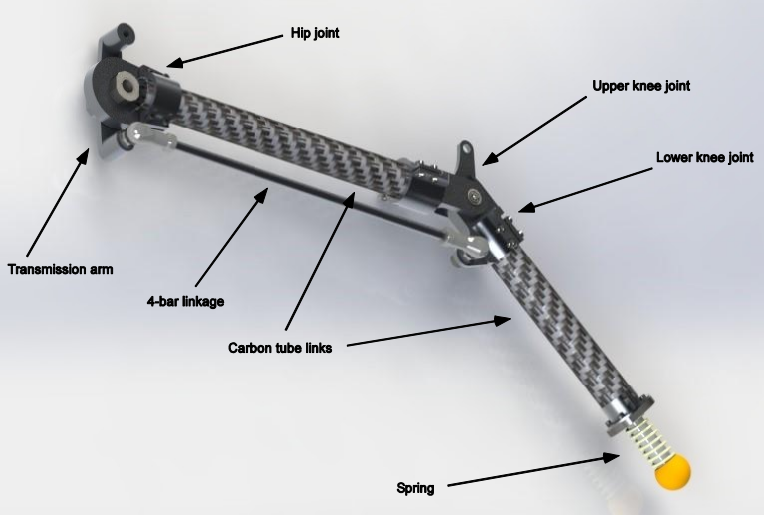


Figure 5-12. Laelaps II new leg design.

5.4 Structural analysis

The structural durability of the new parts in this design -links, joints and toe- were tested and verified in the extreme condition they would have to support the whole body weight. This is also the case of the rotary gallop, where only one leg is in contact with the ground at each moment. Four main simulations were conducted, three drop tests and a static analysis. Drop test is an analysis where the user defines a drop height or an impact velocity and can study the resultant stresses and displacements.

The first was a drop test with the entire body weight, while femur and shank links are aligned, with the knee locked because of the 0º stop, for a 45º touchdown angle. To stress the design even more, the impact velocity was set to 9.7 m/s, which corresponds to the velocity the toe would acquire if the hip motor increased to its no load speed. The second drop test was the same, only for vertical touchdown, in order to check the compressive durability of parts. The third drop test included the knee bent in 90º and again a 45º touchdown. The static analysis was similar to the third simulation only this time the hip motor was exerting its maximum possible torque on the leg, with the knee joint being fixed. These simulations are presented below, alongside a separate analysis for the toe.

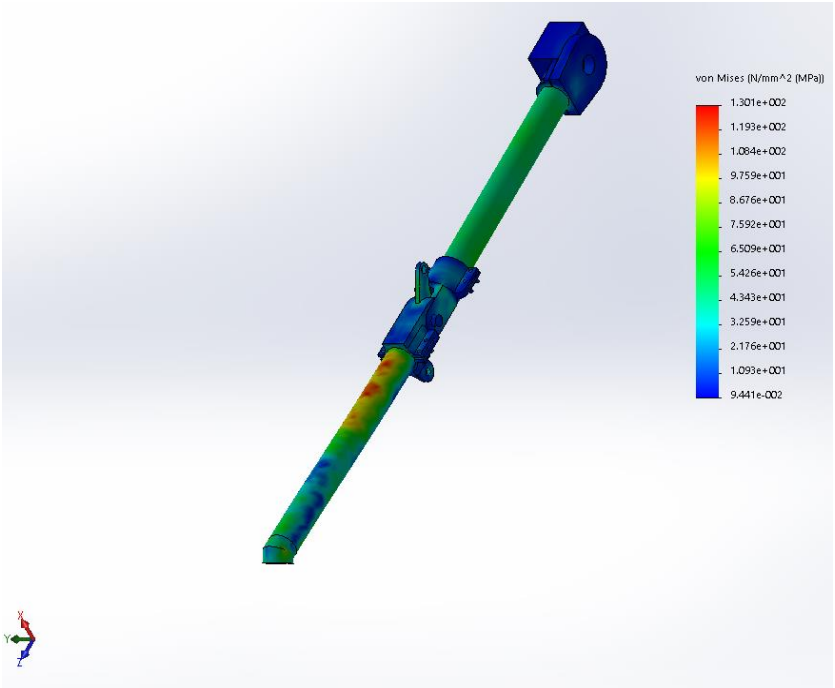


Figure 5-13. First simulation, drop test with a 45º touchdown angle and impact velocity of 9.7 m/s.

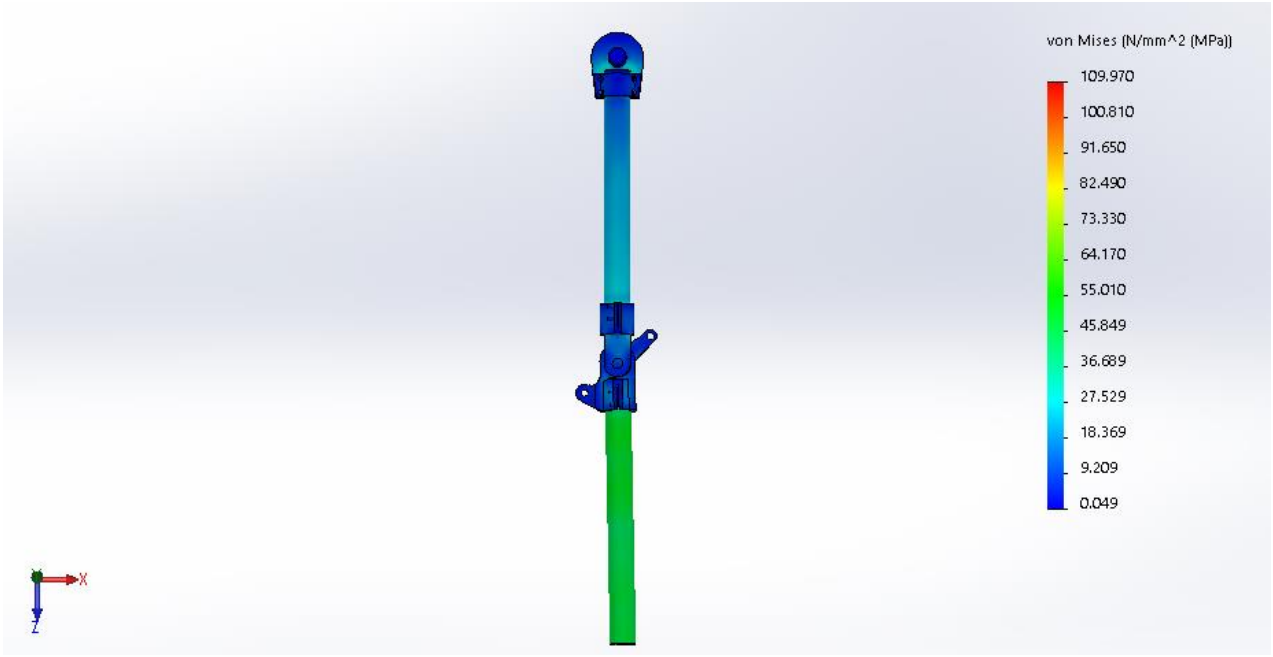


Figure 5-14. Second simulation, drop test for vertical touchdown.

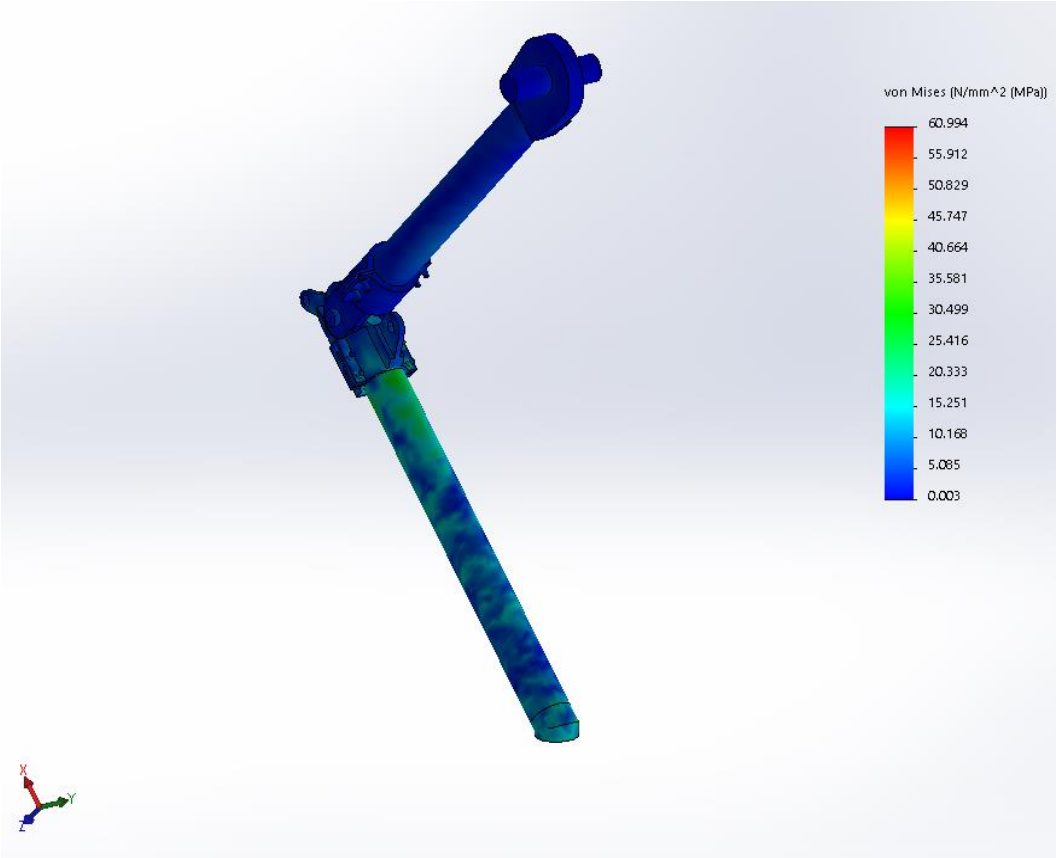


Figure 5-15. Third simulation, drop test with a 45º touchdown angle.

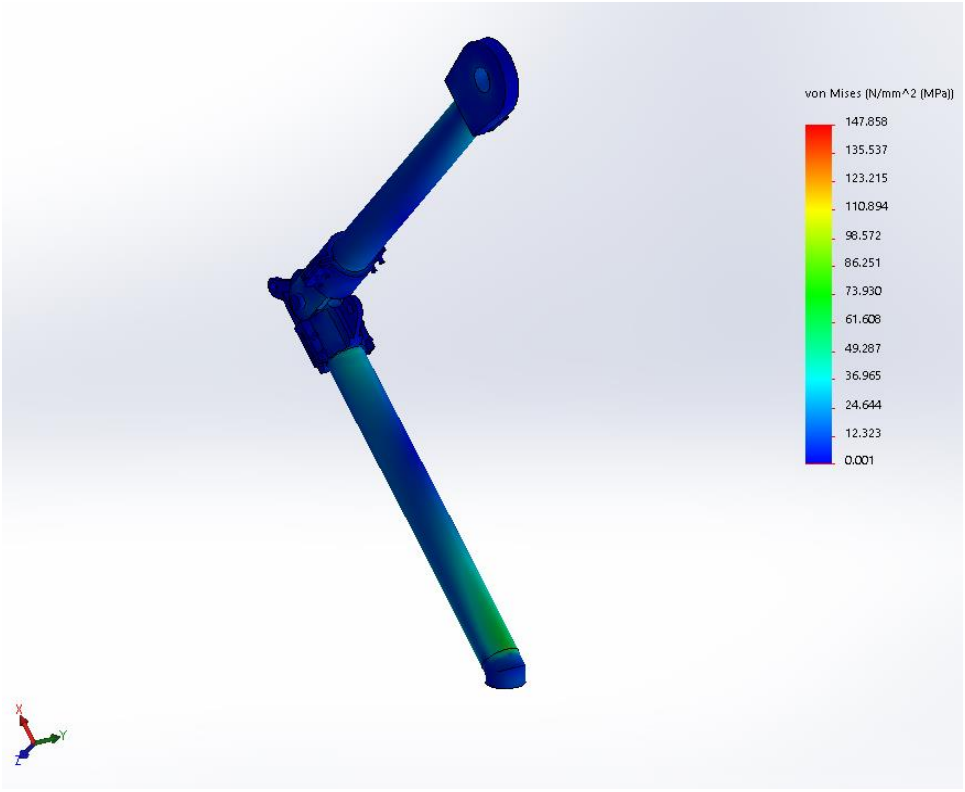


Figure 5-16. Fourth simulation, static analysis stresses.

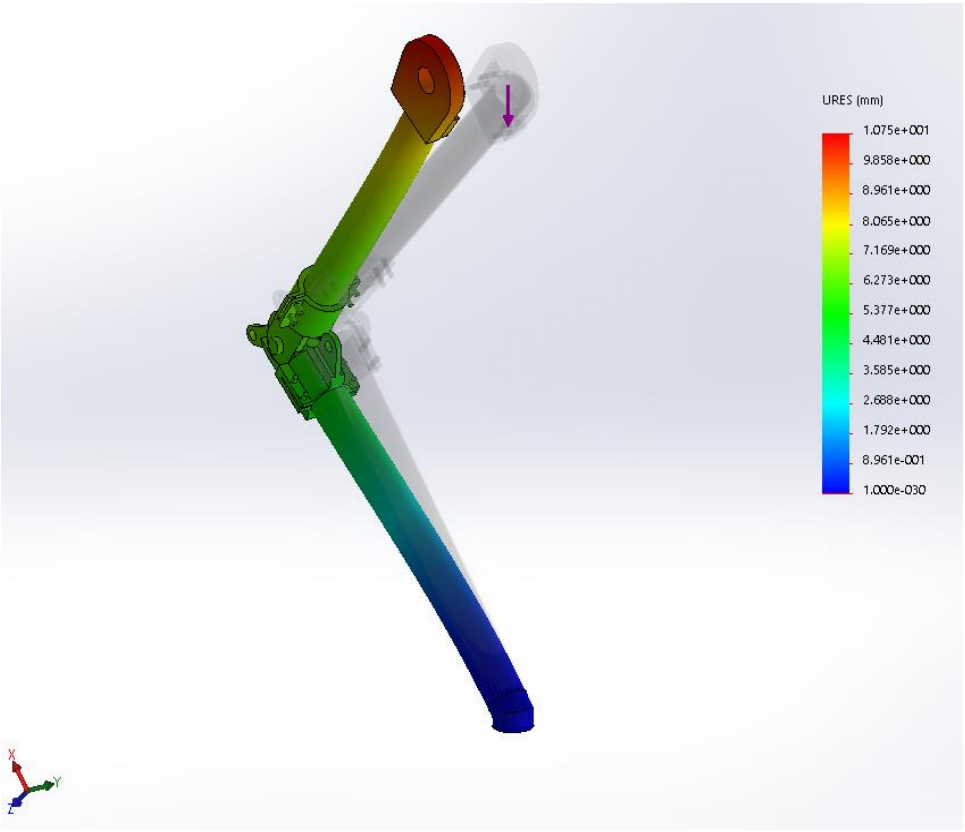


Figure 5-17. Fourth simulation, static analysis displacements.

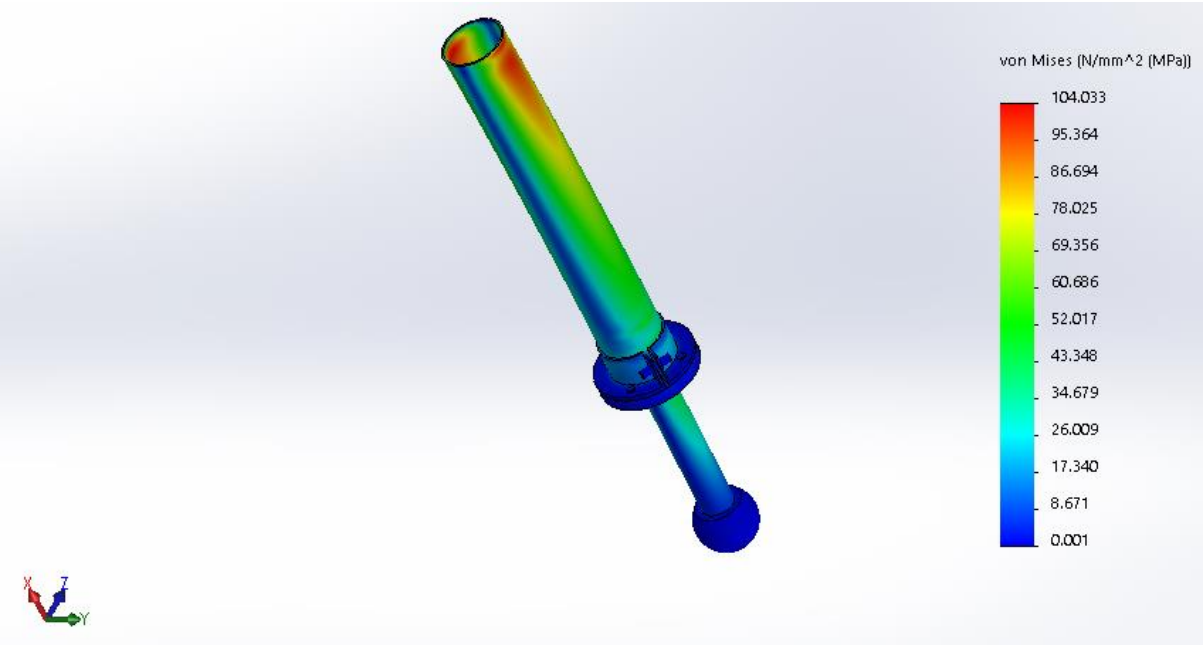


Figure 5-18. Toe setup structural analysis stresses.

The simulations above verify the structural durability of the designed parts, as the stresses do not exceed the yield strength values at any point. The part which was most stressed was the carbon tube of the lower leg, so maybe the second smaller tube should be inserted inside the first, since it would not make the design much heavier; it weighs about 50 g. Finally, we could probably even refine and reduce the dimensions of the joint parts even more, since they do not seem to have any trouble bearing the loads, in order to further reduce the total weight.

6 Conclusion and Future Work

6.1 Conclusion

In the context of this thesis, the ROS platform was successfully employed to receive measurements from and command the actuators of our lab's monopod robot. At first we introduced ROS and we mentioned its basic tools and capabilities. Additionally useful tools and capabilities were mentioned and their operation briefly presented. Gazebo was also introduced and described, as well as the path planning tool MoveIt!.

To incorporate ROS, but also to upgrade system performance and ease of use, the active electronic system was substituted by Tiva microcontrollers and custom supply boards. The motor drive unit was replaced with a new one, more modern and simple to use. The structure of the control system that was built was presented and this system was tested on a model created especially for this purpose in Gazebo. There, besides gains and typical simulation parameters, we also tested and tuned the control loop and data transmission rates. These simulations were compared to those of Matlab and were found to be identical. They also seemed to approximate the behavior of the real robot quite accurately. Finally, an experiment with a high level controller was conducted. We managed to get a limited hopping motion of about five strides. To summarize, we moved from a dysfunctional and outdated experimental setup to a modern one, with handy tools that make the software development process less time consuming and complicated. This system has accurately defined inputs and outputs and therefore it can connect easily to any high level controller.

Moving on to the lab's treadmill, again the entire sensory system and electronics were replaced. The custom velocity Hall sensor was substituted by an incremental encoder. A Tiva board was once more used to read measurements and a similar control system to that of the monopod was developed. In fact we modified the same low level PID controller to control the velocity which was used on the monopod to control the position. The system's functionality and capability of reaching velocities up to 10 m/s were examined and verified by conducting velocity control experiments.

Simultaneously with the aforementioned work, the new leg design method for performance running was also being developed and refined in our lab. After conclusive results were received, a new articulated leg was designed. This leg is lighter than the old one and seems to be able to withstand any stress that might occur while running. It also offers convenient features, like easy change of segment length and exchangeability between push bars and tendons.

6.2 Future Work

Future work could be divided into two separate branches, software and hardware.

As far as software is concerned, probably the most significant update, and worth of the time it requires, is the incorporation of the ros_control package. This is even more important now, that the new leg has two actuated DOF instead of one. This package's complexity was deemed excessive for our project, but as the number of controlled joints increases (e.g. Laelaps has ten, including tail and spine revolute joints), it is more convenient to utilize ros_control instead of running several instances of the ros_pid package. It shall require a significant amount of time at start but then, the addition of new controllers will be greatly simplified.

Also the utilization of MoveIt! would probably prove to be really convenient. Especially if ros_control is previously set up, the amount of time necessary to configure it would greatly decrease, as there exists a noticeable sum of information on their cooperation. Besides saving time from writing custom path planning codes for joints, it could be used to provide direction for the robot's movement (e.g. move forward, backward) through a graphic environment, where one can even add feedback from cameras or other optical sensors and experiment with navigation and obstacle avoidance.

Another suggestion would be to try and close the low level PID control loop on the Tiva boards, which should be more than capable to withstand the computational load. The high level control would remain on a main computer which, being relieved of the load of many different controllers, could be more efficient or even replaced by a device like a Raspberry Pi, something that was not possible so far, exactly because of the computational load.

Finally, since no stable gait was achieved, the experiments should be run again, perhaps with better refinement of the parameters described and the initial conditions. We could even try to add a counterweight to reduce the motor load.

On the hardware side, the most important task at hand is to manufacture and test the leg design of Chapter 5. Also a more analytical exploration of material options and manufacturing methods might has to be carried out, like 3D printed, plastic or resin parts in combination with carbon skinning.

Moving on to the treadmill, a system like the one described needs to be constructed for the inclination counterpart. Since the first prototype seemed to work properly, the new one should be materialized using printed boards instead of flexible cable connections. Another idea would be to try not to use an external supply for the system, but use the incorporated 24V that the inverter provides instead.

Bibliography

[1] Ahmadi, M., and Buehler, M., “The ARL Monopod II Running Robot: Control and Energetics,” *Proceedings of IEEE International Conference on Robotics and Automation*, Detroit, USA (May 1999) pp. 1689–1694.

[2] Ajilo, D., “Mechanical Design of a Quadruped Robot”, *Diploma thesis*, MIT, June 2015.

[3] Ananthanarayanan, A., Azadi, M., and Kim, S., “Towards a bio-inspired leg design for high-speed running”, *Bioinspiration & Biomimetics, Volume 7, Number 4 ,* 8 August 2012.

[4] Assimakopoulos, K., “Design and Development of the Electric and Electronic Subsystems of a Quadruped Robot”, *MSc Dissertation*, NTUA, 2016

[5] Azad, M. & Featherstone, R. Auton Robot (2016) 40: 93. doi:10.1007/s10514-015-9446-z

[6] Balbuena, David, "Dynamic Legged Robot: Shape Deposition Manufacturing" (2015). *Undergraduate Research Symposium 2015*, Book 9.

[7] Cherouveim, N. and Papadopoulos, E., “Single Actuator Control of a 3DOF Hopping Robot”, *Chapter in Robotics: Science and Systems I*, MIT Press, Cambridge, MA, December 2005.

[8] Cherouveim, N., “Dynamics and control of legged robots”, *PhD Dissertation*, NTUA, 2009.

[9] Cutkosky, M. R. and Kim, S., “Design and fabrication of multi-material structures for bioinspired robots”, *Philosophical Transactions of The Royal Society*, The Royal Society Publishing, 30 March 2009.

[10] Grimmes, J. A., and Hurst, J. W., “The Desing of ATRIAS 1.0, A Unique Monopod, Hopping Robot”, *CLAWAR 2012 – Proceedings of the Fifteenth International Conference on Climbing and Walking Robots and the Support Technologies for Mobile Machines*, Baltimore, MD, USA, 23 – 26 July 2012.

[11] Hurst, J. W., “The Role and Implementation of Compliance in Legged Locomotion”, *PhD Dissertation,* Carnegie Mellon University, Pittsburgh, Pennsylvania, 2008.

[12] Hutter, M., “StarlETH & Co. – Design and Control of Legged Robots with Compliant Actuation”, *PhD Dissertation,* ETH*,* Zurich, Switzerland, 2013.

[13] Hutter, M., Gehring, C., Jud, D., Lauber, A., Bellicoso, C. D., Tsounis, V., Hwangbo, J., Bodie, K., Fankhauser, P., Bloesch, M., Diethelm, R., Bachmann, S., Melzer, A. and Hoepflinger, M., “ANYmal - A Highly Mobile and Dynamic Quadrupedal Robot”, *IEEE/RSJ International Conference on Intelligent Robots and Systems*, IROS 2016, (2016) Zürich, ETH Zürich.

[14] Hyon, S. H., and Mita, T., “Development of a biologically inspired hopping robot-"kenken",” *IEEE International Conference on Robotics and Automation*, 2002. Proceedings. ICRA ’02, vol. 4, 2002, pp. 3984–3991 vol.4.

[15] Khan, H., Featherstone, R., Caldwell, D. G., Semini, C., “Bio-inspired Knee Joint Mechanism for a Hydraulic Quadruped Robot”, *Proceedings of the 6th International Conference on Automation, Robotics and Applications*, Queenstown, New Zealand, Feb 17-19, 2015.

[16] Machado, J. A. Tenreiro and Silva, M. F, “An Overview of Legged Robots”, Department of Electrical Engineering, Institute of Engineering of Porto, Porto, Portugal, 2006.

[17] Manolioudakis, P., “Modeling and closed loop control of an experimental treadmill with 3-phase induction motors”, *Diploma Thesis*, NTUA, 2014.

[18] McKenzie, J. E., “Design of Robotic Quadruped Legs”, *MSc Dissertation*, MIT, February 2012.

[19] Ming, A., Sato, K., Sato, R., Kazama, E., Miyamoto, I., Shimojo, M., “Development of Robot Leg Composed of Parallel Linkage and Elastic Spring for Dynamic Locomotion”, *Proceeding of the 2015 IEEE International Conference on Information and Automation*, Lijiang, China, August 2015.

[20] Noussias, S., “Quadrupedal State estimation using Kalman filtering techniques”, *MSc Dissertation*, NTUA, 2015.

[21] Ogata, K., *Modern Control Engineering*, 5th Edition.

[22] Parthenaki, A. M., “Design and Construction of a Passive 6-dof Mechanism for Constraining Legged Robots on a Treadmill”, *MSc Dissertation*, NTUA, 2016

[23] Raibert, M. H, *Legged robots that balance*, MIT Press, Cambridge, MA, 1986.

[24] Raibert, M. H, “Trotting, pacing, and bounding by a quadruped robot”, *Journal of Biomechanics*, 1991.

[25] Rehman, F., “Steering control of a hopping robot model during the flight phase”, *IEEE Proc.-Control Theory Appl.*, Vol. 152, No. 6, November 2005

[26] Rummel, J., Iida, F., Smith, J. A., Seyfarth, A., “Enlarging Regions of Stable Running with Segmented Legs”, *2008 IEEE International Conference on Robotics and Automation*, Pasadena, CA, USA, May 19-23, 2008.

[27] Sang-Ho Hyon and Takashi Emura, “Energy-preserving control of a passive one-legged running robot”, *Advanced Robotics*, Vol. 18, No. 4, pp. 357–381 (2004).

[28] Sato, R., Miyamoto, I., Sato, K., Ming, A., Shimojo, M., “Development of Robot Legs Inspired by Bi-articular Muscle-tendon Complex of Cats” in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Hamburg, Germany, Sept 28 - Oct 2, 2015.

[29] Semini, C., “HyQ - Design and Development of a Hydraulically Actuated Quadruped Robot”, *PhD Dissertation,* University of Genoa, Italy, Italian Institute of Technology (IIT), April 2010.

[30] Serena, I., Padois, V., Nori, F., "Tools for dynamics simulation of robots: a survey based on user feedback.", 2014.

[31] Vasilopoulos, V., Paraskevas, I. S. and Papadopoulos, E., “Control and Energy Considerations for a Hopping Monopod on Rough Compliant Terrains”, *2015 IEEE International Conference on Robotics and Automation (ICRA)*, Washington State Convention Center, Seattle, Washington, May 26-30, 2015.

[32] Vasilopoulos, V., “Dynamics and control of a monopod robot with a single actuator on compliant terrains”, *Diploma Thesis*, NTUA, 2014.

[33] Zeglin, G., “Uniroo: A one legged dynamic hopping robot”, *Diploma Thesis,* MIT, Cambridge, May 1991.

[34] http://www.rsl.ethz.ch/content/dam/ethz/special-interest/mavt/robotics-n-intelligent-systems/rsl-dam/documents/ANYdriveFlyer.pdf

[35] http://www.ros.org/core-components/

[36] https://www.h-ros.com/#

[37] http://gazebosim.org/

[38] http://newatlas.com/darpa-virtual-robotics-challenge-winners/28092/#gallery

[39] http://moveit.ros.org/robots/

[40] http://sdformat.org/

[41] http://www.kode-physics.org/wiki/index.php?title=Manual:_ERP_and_CFM

[42] http://gazebosim.org/tutorials?tut=ros_control&cat=connect_ros

[43] http://www.ai.mit.edu/projects/leglab/robots/monopod/monopod.html

[44] http://www.agilityrobotics.com/

[45] http://wiki.ros.org/urdf

[46] http://gazebosim.org/tutorials?tut=ros_urdf&cat=connect_ros

[47] https://nereus.mech.ntua.gr/legged/

[48] http://csl-ep.mech.ntua.gr/index.php/research/robotics-for-extreme-environments/space-robotics

[49] http://wiki.ros.org/Robots

Appendix A

In this appendix, the various codes utilized during the work described are included.

Hip_interface

```
#include "arpa/inet.h"
#include "netinet/in.h"
#include "stdio.h"
#include "sys/types.h"
#include "sys/socket.h"
#include "unistd.h"
#include "string.h"
#include "stdlib.h"
#include "signal.h"
#include "unistd.h"
#include "fcntl.h"
#include <stdint.h>
#include <inttypes.h>
#include "ros/ros.h"
#include "std_msgs/Float64.h"
#include <sstream>

// UDP buffer length
#define BUFLEN 512

// UDP port to receive from
#define PORT 2015

// UDP port to send data to
#define PORT_BRD 2016

// Asynchronous UDP communication
#define ASYNC

// Tiva Hip board IP
#define BRD_IP "192.168.1.12"

// Global variables
bool gotMsg = false; // Flag set high when message is received from UDP
int sock; // The socket identifier for UDP Rx communication
int32_t encoderPos = 0;  // Place the received encoder value here
int32_t pos = 0;  // Place the actual value here
int32_t prevpos = 0;  // previous actual position
int32_t prevencoderpos = 0;  //previous encoder position
int msgs = 0; // Incoming message counter
int msgss = 0; // Incoming message counter
struct sockaddr_in si_pwm;  // Struct for UDP send data socket
ssize_t SendPWMBytes = 2;  // Number of bytes to send for PWM command
char SendBuffer[6];  // UDP Send Buffer
int broad;  // The socket identifier for UDP Tx communication
int slen=sizeof(si_pwm);  // Size of sockaddr_in strut
float pi = 4.0*atan(1.0);

// Generic error function
void error(char *s)
{
    perror(s);
    exit(1);
}

// Signal handler for asynchronous UDP
void sigio_handler(int sig)
{
        char buffer[BUFLEN]="";
    unsigned char val[4];
    struct sockaddr_in si_other;
    unsigned int slen=sizeof(si_other);
    ssize_t rcvbytes = 0;
```

```
    // Receive available bytes from UDP socket
    if ((rcvbytes = recvfrom(sock, &buffer, BUFLEN, 0, (struct sockaddr *)&si_other, &slen))==-1)
                error("recvfrom()");
    else
    {
            // Parse data , 1 int32 value
            if(buffer[0] == 0x42)
            {
                        //ROS_INFO(" received");
                    val[3] = (unsigned char)buffer[4];
                    val[2] = (unsigned char)buffer[3];
                    val[1] = (unsigned char)buffer[2];
                    val[0] = (unsigned char)buffer[1];
                        prevencoderpos = encoderPos ;
                        memcpy(&encoderPos, &val, 4);
                        if (((prevencoderpos < 50000) && (encoderPos > 50000)) || ((pos < 0) &&
(encoderPos < prevencoderpos)))
                        {
                       prevpos = pos ;
                       pos = encoderPos - 104000 ;
                        }
                    else
                    {
                       prevpos = pos ;
                       pos = encoderPos ;
                    }
                    if (encoderPos = 0)
                        pos = encoderPos ;
                    // Raise flag that we received a message
                    gotMsg = true;
            }
   }
}

// Function to enable asynchronous UDP communication
int enable_asynch(int sock)
{
  int stat = -1;
  int flags;
  struct sigaction sa;

  flags = fcntl(sock, F_GETFL);
  fcntl(sock, F_SETFL, flags | O_ASYNC);

  sa.sa_flags = 0;
  sa.sa_handler = sigio_handler;
  sigemptyset(&sa.sa_mask);

  if (sigaction(SIGIO, &sa, NULL))
    error("Error:");

  if (fcntl(sock, F_SETOWN, getpid()) < 0)
    error("Error:");

  if (fcntl(sock, F_SETSIG, SIGIO) < 0)
    error("Error:");

  return 0;
}

// Callback function for reception of PWM message from topic
void PWMCallback(const std_msgs::Float64::ConstPtr& msg)
{
  // Extract the duty cycle value and send it to the Tiva board via UDP
  SendBuffer[1] = (int8_t) msg->data;
  if (sendto(broad, SendBuffer, SendPWMBytes, 0, (struct sockaddr *)&si_pwm, slen)==-1)
    error("sendto()");
  // Print-out for debugging
  msgs++;
  if(msgs == 10)
  {
    msgs = 0;
    //ROS_INFO("I heard: [%f]", msg->data);
  }
```

```
}

// Main Function
int main(int argc, char **argv)
{
  struct sockaddr_in si_me, si_other;
  int i, slen=sizeof(si_other), msg_count;
  char buf[BUFLEN], strout[28];

  msg_count = 0;
  memset(SendBuffer, 0, 6);

  // Initialize UDP socket for data transmission
  if ((broad=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP))==-1)
       error("socket");

  memset((char *) &si_pwm, 0, sizeof(si_pwm));
  si_pwm.sin_family = AF_INET;
  si_pwm.sin_port = htons(PORT_BRD);

  if (inet_aton(BRD_IP, &si_pwm.sin_addr)==0) {
       error("inet_aton() failed\n");
       exit(1);
  }

  SendBuffer[0] = 0x31;

  // Initialize ROS node
  ros::init(argc, argv, "Hip_interface");
  ros::NodeHandle n;
  // Initialize the publisher for Encoder data post
  ros::Publisher position_interface_pub = n.advertise<std_msgs::Float64>("/state", 1000);

  // Initialize the subscriber for PWM data reception
  ros::Subscriber pwm_sub = n.subscribe("control_effort", 1000, PWMCallback);

  ros::Rate loop_rate(6000);

  // Wait for ROS node to initialize
  while (!ros::ok());

  // Initialize UDP socket for data reception
  if ((sock=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP))==-1)
     error("socket");

  memset((char *) &si_me, 0, sizeof(si_me));
  si_me.sin_family = AF_INET;
  si_me.sin_port = htons(PORT);
  si_me.sin_addr.s_addr = htonl(INADDR_ANY);
  if (bind(sock, (struct sockaddr *)&si_me, sizeof(si_me))==-1)
      error("bind");

  enable_asynch(sock);

  ROS_INFO("Starting communication with Tiva hip board.");
  ROS_INFO("Communication with Tiva hip board established.");

  std_msgs::Float64 state_msg;
  state_msg.data = 0.0;

  while (ros::ok())
  {
    // If we got a new message, publish to topic and print values every 100 messages
    if(gotMsg)
        {
            msg_count++;
            state_msg.data = (float) pos;
                position_interface_pub.publish(state_msg);
            if(msg_count >= 2000)
            {
                    msg_count = 0;
                    ROS_INFO("%i  angle", pos);
            }
            gotMsg = false;
```

```
    }
    ros::spinOnce();
    loop_rate.sleep();
  }
  return 0;
}
```

Knee_interface

```
#include "arpa/inet.h"
#include "netinet/in.h"
#include "stdio.h"
#include "sys/types.h"
#include "sys/socket.h"
#include "unistd.h"
#include "string.h"
#include "stdlib.h"
#include "signal.h"
#include "unistd.h"
#include "fcntl.h"
#include <stdint.h>
#include <inttypes.h>
#include "ros/ros.h"
#include "std_msgs/Float64.h"
#include <sstream>
#include "math.h"
#include <ros/time.h>

// UDP buffer length
#define BUFLEN 512

// UDP port to receive from
#define PORT 2014

// UDP port to send data to
#define PORT_BRD 2013

// Asynchronous UDP communication
#define ASYNC

// Tiva Knee board IP
#define BRD_IP "192.168.1.11"

// Global variables
bool gotMsg = false; // Flag set high when message is received from UDP
int sock; // The socket identifier for UDP Rx communication
int msgs = 0; // Incoming message counter
struct sockaddr_in si_pwm;  // Struct for UDP send data socket
ssize_t SendPWMBytes = 2;  // Number of bytes to send for PWM command
char SendBuffer[6];  // UDP Send Buffer
int broad;      // The socket identifier for UDP Tx communication
int slen=sizeof(si_pwm);  // Size of sockaddr_in strut
float pi = 4.0*atan(1.0);

int32_t encoderPos = 0;  // Place the received encoder value here
float angle, compression ;
float l1 = 0.05 ;    // knee top link length
float l2 = 0.065 ;    // knee bot link length
float n1 = 0.089 ;    // initial length
float linit = sqrt(l1*l1 + l2*l2 - 2.0*l1*l2*cos(360.0*361/1999.0*pi/180.0) - 0.01*0.01) ;

float norm_compression ;

// Generic error function
void error(char *s)
{
    perror(s);
    exit(1);
}
```

```
// Signal handler for asynchronous UDP
void sigio_handler(int sig)
{
  char buffer[BUFLEN]="";
  unsigned char val[4];
  struct sockaddr_in si_other;
  unsigned int slen=sizeof(si_other);
  ssize_t rcvbytes = 0;

  // Receive available bytes from UDP socket
  if ((rcvbytes = recvfrom(sock, &buffer, BUFLEN, 0, (struct sockaddr *)&si_other, &slen))==-1)
    error("recvfrom()");
  else
  {
    // Parse data , 1 int32 value
    if(buffer[0] == 0x42)
    {
      //ROS_INFO(" received");
      val[3] = (unsigned char)buffer[4];
            val[2] = (unsigned char)buffer[3];
            val[1] = (unsigned char)buffer[2];
            val[0] = (unsigned char)buffer[1];
            memcpy(&encoderPos, &val, 4);
      comrpession = (sqrt(l1*l1 + l2*l2 - 2.0*l1*l2*cos(360.0*encoderPos/1999.0*pi/180.0) -
0.01*0.01)- linit) ;
      // Raise flag that we received a message
      gotMsg = true;
    }
  }
}

// Function to enable asynchronous UDP communication
int enable_asynch(int sock)
{
  int stat = -1;
  int flags;
  struct sigaction sa;

  flags = fcntl(sock, F_GETFL);
  fcntl(sock, F_SETFL, flags | O_ASYNC);

  sa.sa_flags = 0;
  sa.sa_handler = sigio_handler;
  sigemptyset(&sa.sa_mask);

  if (sigaction(SIGIO, &sa, NULL))
    error("Error:");

  if (fcntl(sock, F_SETOWN, getpid()) < 0)
    error("Error:");

  if (fcntl(sock, F_SETSIG, SIGIO) < 0)
    error("Error:");

  return 0;
}


// Main Function
int main(int argc, char **argv)
{
  struct sockaddr_in si_me, si_other;
  int i, slen=sizeof(si_other), msg_count;
  char buf[BUFLEN], strout[28];

  msg_count = 0;
  memset(SendBuffer, 0, 6);

  // Initialize UDP socket for data transmission
  if ((broad=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP))==-1)
       error("socket");

  memset((char *) &si_pwm, 0, sizeof(si_pwm));
```

```
  si_pwm.sin_family = AF_INET;
  si_pwm.sin_port = htons(PORT_BRD);

  if (inet_aton(BRD_IP, &si_pwm.sin_addr)==0) {
       error("inet_aton() failed\n");
       exit(1);
  }

  SendBuffer[0] = 0x31;

  // Initialize ROS node
  ros::init(argc, argv, "Knee_interface");
  ros::NodeHandle n;
  // Initialize the publisher for compression data post
  ros::Publisher compression_pub = n.advertise<std_msgs::Float64>("/compression", 1000);

  ros::Rate loop_rate(1000);

  // Wait for ROS node to initialize
  while (!ros::ok());

  // Initialize UDP socket for data reception
  if ((sock=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP))==-1)
     error("socket");

  memset((char *) &si_me, 0, sizeof(si_me));
  si_me.sin_family = AF_INET;
  si_me.sin_port = htons(PORT);
  si_me.sin_addr.s_addr = htonl(INADDR_ANY);
  if (bind(sock, (struct sockaddr *)&si_me, sizeof(si_me))==-1)
    error("bind");

  enable_asynch(sock);

  ROS_INFO("Starting communication with Tiva knee board.");
  ROS_INFO("Communication with TiVa knee board established.");

  std_msgs::Float64 compression_msg;

  while (ros::ok())
  {
    if(gotMsg)
    {
      msg_count++;
      compression_msg.data = compression ;
      compression_pub.publish(compression_msg);
      if(msg_count >= 2000)
      {
                    msg_count = 0;
                    ROS_INFO("%f  compression in m", compression);
      }
      gotMsg = false;
    }
    ros::spinOnce();
    loop_rate.sleep();
  }
  return 0;
}
```

IMU_interface

```
#include "arpa/inet.h"
#include "netinet/in.h"
#include "stdio.h"
#include "sys/types.h"
#include "sys/socket.h"
#include "unistd.h"
#include "string.h"
#include "stdlib.h"
#include "signal.h"
#include "unistd.h"
#include "fcntl.h"
```

```
#include <stdint.h>
#include <inttypes.h>
#include "ros/ros.h"
#include "legged_robot/AccGyro.h"
#include <sstream>
#include <ros/time.h>

// UDP buffer length
#define BUFLEN 512
// UDP port to receive from
#define PORT 2012

// Asynchronous UDP communication
#define ASYNC

// Tiva IMU board IP
#define BRD_IP "192.168.1.10"

// Global variables
bool gotMsg = false;   // Flag set high when message is received from UDP
int sock;   // The socket identifier for UDP communication
int msgs = 0;  // Incoming message counter
int16_t acc[3] = {0,0,0};  // Place raw accelerometer data here
int16_t gyro[3] = {0,0,0};  // Place raw gyroscope data here
int imu_dev = 0; // 1 - ADIS16375

// Generic error function
void error(char *s)
{
    perror(s);
    exit(1);
}

// Signal handler for asynchronous UDP
void sigio_handler(int sig)
{
   char buffer[BUFLEN]="";
   unsigned char val[2];
   struct sockaddr_in si_other;
   unsigned int slen=sizeof(si_other);
   ssize_t rcvbytes = 0;

   // Receive available bytes from UDP socket
   if ((rcvbytes = recvfrom(sock, &buffer, BUFLEN, 0, (struct sockaddr *)&si_other, &slen))==-1)
       error("recvfrom()");
   else
   {
    //ROS_INFO("%d bytes");
          // Parse data , 6 int16 values
    if(buffer[0] == 0x43 && rcvbytes == 13)
    {
      imu_dev = 1;
      val[1] = (unsigned char)buffer[2];
      val[0] = (unsigned char)buffer[1];
      memcpy(&acc[0], &val, 2);
      val[1] = (unsigned char)buffer[4];
      val[0] = (unsigned char)buffer[3];
      memcpy(&acc[1], &val, 2);
      val[1] = (unsigned char)buffer[6];
      val[0] = (unsigned char)buffer[5];
      memcpy(&acc[2], &val, 2);
      val[1] = (unsigned char)buffer[8];
      val[0] = (unsigned char)buffer[7];
      memcpy(&gyro[0], &val, 2);
      val[1] = (unsigned char)buffer[10];
      val[0] = (unsigned char)buffer[9];
      memcpy(&gyro[1], &val, 2);
      val[1] = (unsigned char)buffer[12];
      val[0] = (unsigned char)buffer[11];
      memcpy(&gyro[2], &val, 2);
          // Raise flag that we received a message
      gotMsg = true;
    }
  }
```

```
}

// Function to enable asynchronous UDP communication
int enable_asynch(int sock)
{
  int stat = -1;
  int flags;
  struct sigaction sa;

  flags = fcntl(sock, F_GETFL);
  fcntl(sock, F_SETFL, flags | O_ASYNC);

  sa.sa_flags = 0;
  sa.sa_handler = sigio_handler;
  sigemptyset(&sa.sa_mask);

  if (sigaction(SIGIO, &sa, NULL))
    error("Error:");

  if (fcntl(sock, F_SETOWN, getpid()) < 0)
    error("Error:");

  if (fcntl(sock, F_SETSIG, SIGIO) < 0)
    error("Error:");
  return 0;
}

// Main Function
int main(int argc, char **argv)
{
  struct sockaddr_in si_me, si_other;
  int i, slen=sizeof(si_other), msg_count;
  char buf[BUFLEN], strout[28];
  legged_robot::AccGyro accgyro_msg;
  double realAcc[3], realGyro[3] ;

  ros::Time prev_time;
  ros::Duration delta_t;
  msg_count = 0;

  // Initialize ROS node
  ros::init(argc, argv, "IMU_interface");
  ros::NodeHandle n;
  // Initialize the publisher for Accelerometer and Gyroscope data post
  ros::Publisher imu_interface_pub = n.advertise<legged_robot::AccGyro>("IMU_feedback", 1000);
  ros::Rate loop_rate(1000);

  // Wait for ROS node to initialize
  while (!ros::ok());

  // Initialize UDP communication
  if ((sock=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP))==-1)
     error("socket");

  memset((char *) &si_me, 0, sizeof(si_me));
  si_me.sin_family = AF_INET;
  si_me.sin_port = htons(PORT);
  si_me.sin_addr.s_addr = htonl(INADDR_ANY);
  if (bind(sock, (struct sockaddr *)&si_me, sizeof(si_me))==-1)
      error("bind");

  enable_asynch(sock);

  ROS_INFO("Starting communication with IMU Tiva board.");
  while (ros::ok()){
    // Initialize ROS message values
    accgyro_msg.accX = 0;
    accgyro_msg.accY = 0;
    accgyro_msg.accZ = 0;
    accgyro_msg.gyroX = 0;
    accgyro_msg.gyroY = 0;
    accgyro_msg.gyroZ = 0;
    accgyro_msg.imu_dev = 0;
```

```
        // If we got a new message, publish to topic and print values every 500 messages
    if(gotMsg)
    {
      msg_count++;
      accgyro_msg.accX = acc[0];
      accgyro_msg.accY = acc[1];
      accgyro_msg.accZ = acc[2];
      accgyro_msg.gyroX = gyro[0];
      accgyro_msg.gyroY = gyro[1];
      accgyro_msg.gyroZ = gyro[2];
            accgyro_msg.imu_dev = imu_dev;
      imu_interface_pub.publish(accgyro_msg);
      if(msg_count >= 1)
      {
                    msg_count = 0;
                    if(imu_dev == 1)
        {
                        realAcc[0] = (accgyro_msg.accX*1.0)*0.8192*9.81/1000.0 ;
                        realAcc[1] = (accgyro_msg.accY*1.0)*0.8192*9.81/1000.0 ;
                        realAcc[2] = (accgyro_msg.accZ*1.0)*0.8192*9.81/1000.0 ;
                        realGyro[0] = (accgyro_msg.gyroX*1.0)*0.013108;
                        realGyro[1] = (accgyro_msg.gyroY*1.0)*0.013108;
                        realGyro[2] = (accgyro_msg.gyroZ*1.0)*0.013108;
                    }
      }
      gotMsg = false;
    }
    ros::spinOnce();
    loop_rate.sleep();
  }
  return 0;
}
```

Read_setpoint

```
#include "stdio.h"
#include "string.h"
#include "stdlib.h"
#include <inttypes.h>
#include "ros/ros.h"
#include <iostream>
#include <string>
#include <sstream>
#include "std_msgs/Float64.h"
#include <dynamic_reconfigure/server.h>
#include <legged_robot/ParamConfig.h>

float rpos, prevpos ;
float dt  ;

using namespace std;

void callback(legged_robot::ParamConfig &config, uint32_t level)
{
        rpos =  config.setpoint;
}

// Global variables
std_msgs::Float64 setpoint_msg;
float pi = 4.0*atan(1.0);

int main(int argc, char **argv)
{
  prevpos = 0.0 ;
  // Initialize ROS node
  ros::init(argc, argv, "Read_setpoint");
  ros::NodeHandle nod;
  // Publish for desired position message
  ros::Publisher read_setpoint_pub = nod.advertise<std_msgs::Float64>("/setpoint", 1000);

  dynamic_reconfigure::Server<legged_robot::ParamConfig> Read_setpoint;
  dynamic_reconfigure::Server<legged_robot::ParamConfig>::CallbackType f;
```

```
  f = boost::bind(&callback, _1, _2);
  Read_setpoint.setCallback(f);

  ros::Rate loop_rate(1000);
  setpoint_msg.data = 0.0;
  ROS_INFO("Reading Setpoint.");

  while (ros::ok())
  {
        if (rpos != prevpos)
        {

                prevpos = rpos ;
        }
        setpoint_msg.data =  rpos ;
        read_setpoint_pub.publish(setpoint_msg);
    ros::spinOnce();
    loop_rate.sleep();
  }
  return 0;
}
```

Tiva boards

```
#include <stdint.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "driverlib/debug.h"
#include "driverlib/gpio.h"
#include "drivers/pinout.h"
#include "driverlib/interrupt.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/rom_map.h"
#include "driverlib/sysctl.h"
#include "driverlib/uart.h"
#include "utils/uartstdio.h"
#include "driverlib/flash.h"
#include "driverlib/systick.h"
#include "utils/locator.h"
#include "utils/lwiplib.h"
#include "utils/ustdlib.h"
#include "inc/hw_pwm.h"
#include "driverlib/pwm.h"
#include "inc/hw_qei.h"
#include "driverlib/qei.h"
#include "driverlib/timer.h"

#include "main.h"

#ifdef DEV_ADIS16375
#include "ADIS16375.h"
#endif
#include "spi.h"

//*****************************************************************************
//
// Defines for setting up the system clock.
//
//*****************************************************************************
#define SYSTICKHZ               100
#define SYSTICKMS               (1000 / SYSTICKHZ)

//*****************************************************************************
//
// Interrupt priority definitions.  The top 3 bits of these values are
// significant with lower values indicating higher priority interrupts.
//
```

```
//*****************************************************************************
#define SYSTICK_INT_PRIORITY    0x80
#define ETHERNET_INT_PRIORITY   0xC0

//*****************************************************************************
//
// The current IP address.
//
//*****************************************************************************
uint32_t g_ui32IPAddress;
//*****************************************************************************
//
// The system clock frequency.
//
//*****************************************************************************

uint32_t g_ui32SysClock;

#ifdef DEBUG
void
__error__(char *pcFilename, uint32_t ui32Line)
{
}
#endif

#ifdef ENABLE_ETHERNET
// Initialize the UDP receive pcb
struct udp_pcb * udp_init_r(void);
// Send data over UDP
void udp_send_data(void* sbuf, u16_t len);
// Callback for UDP data reception
void udp_receive_data(void *arg, struct udp_pcb *pcb, struct pbuf *p, struct ip_addr *addr, u16_t
port);
// The variable that hold the UDP receive pcb
struct udp_pcb *Rpcb;
// Variables assgined to the controller pc IP and current board IP
struct ip_addr controller_ip, board_ip;
// Flag that is raised when the IP is assigned
volatile uint8_t gotIP = 0;
// Variables for lwip configuration
unsigned long device_ip,device_subnet,device_gateway;
#endif

// Flags raised when events for encoder send and pwm set are active
bool sendEncoder, setPWMvalue;
// Variable for received PWM command
int8_t pwmValue;

#ifdef ENABLE_IMU

#ifdef DEV_ADIS16375
// Function that configures the interrupt detection of ADIS16375 IMU
void ConfigureADIS16375Int(void);
// ADIS16375 object
ADIS16375 myIMU;
#endif

// Flag for IMU Data ready on interrupt
uint8_t imuDataReady = 0;
// Variables that hold the measurements received from the IMU
int16_t accel_x, accel_y, accel_z, gyro_x, gyro_y, gyro_z, delta_x, delta_y, delta_z, dv_x, dv_y,
dv_z, temp_out;
double dval_x, dval_y, dval_z, temp, deltaAccX, deltaAccY, deltaAccZ;
#endif


#ifdef ENABLE_ETHERNET
// Display the input IP address on UART
void DisplayIPAddress(uint32_t ui32Addr)
{
    char pcBuf[16];

    //
    // Convert the IP Address into a string.
```

```
    //
    usprintf(pcBuf, "%d.%d.%d.%d", ui32Addr & 0xff, (ui32Addr >> 8) & 0xff,
            (ui32Addr >> 16) & 0xff, (ui32Addr >> 24) & 0xff);

    //
    // Display the string.
    //
#ifdef ENABLE_UART
    UARTprintf(pcBuf);
#endif

}
#endif

#ifdef ENABLE_ETHERNET
// Ethernet lwip interrupt handler
void lwIPHostTimerHandler(void)
{
    uint32_t ui32Idx, ui32NewIPAddress;

    //
    // Get the current IP address.
    //
    ui32NewIPAddress = lwIPLocalIPAddrGet();

    //
    // See if the IP address has changed.
    //
    if(ui32NewIPAddress != g_ui32IPAddress)
    {
        //
        // See if there is an IP address assigned.
        //
        if(ui32NewIPAddress == 0xffffffff)
        {
            //
            // Indicate that there is no link.
            //
            //UARTprintf("Waiting for link.\n");
        }
        else if(ui32NewIPAddress == 0)
        {
            //
            // There is no IP address, so indicate that the DHCP process is
            // running.
            //
            //UARTprintf("Waiting for IP address.\n");
        }
        else
        {
            //
            // Display the new IP address.
            //
#ifdef ENABLE_UART
            UARTprintf("IP Address: ");
            DisplayIPAddress(ui32NewIPAddress);
            UARTprintf("\n");
#endif
            // Set the gotIP flag once IP is assigned
            gotIP = 1;
        }

        //
        // Save the new IP address.
        //
        g_ui32IPAddress = ui32NewIPAddress;

        //
        // Turn GPIO off.
        //
        MAP_GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_1, ~GPIO_PIN_1);
    }

    //
```

```
    // If there is not an IP address.
    //
    if((ui32NewIPAddress == 0) || (ui32NewIPAddress == 0xffffffff))
    {
        //
        // Loop through the LED animation.
        //

        for(ui32Idx = 1; ui32Idx < 17; ui32Idx++)
        {

            //
            // Toggle the GPIO
            //
            MAP_GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_1,
                    (MAP_GPIOPinRead(GPIO_PORTN_BASE, GPIO_PIN_1) ^
                     GPIO_PIN_1));

            SysCtlDelay(g_ui32SysClock/(ui32Idx << 1));
        }
    }
}
#endif

//*****************************************************************************
//
// The interrupt handler for the SysTick interrupt.
//
//*****************************************************************************
void
SysTickIntHandler(void)
{
    //
    // Call the lwIP timer handler.
    //
#ifdef ENABLE_ETHERNET
    lwIPTimer(SYSTICKMS);
#endif
}

#ifdef ENABLE_UART
// Configure the UART peripheral
void ConfigureUART(void)
{
    //
    // Enable the GPIO Peripheral used by the UART.
    //
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

    //
    // Enable UART0
    //
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);

    //
    // Configure GPIO Pins for UART mode.
    //
    ROM_GPIOPinConfigure(GPIO_PA0_U0RX);
    ROM_GPIOPinConfigure(GPIO_PA1_U0TX);
    ROM_GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

    //
    // Initialize the UART for console I/O.
    //
    UARTStdioConfig(0, 115200, g_ui32SysClock);

#ifdef UART_BUFFERED
    UARTEchoSet(false);
#endif
}
#endif

// Generic delay function
void cyclesdelay(unsigned long cycles)
```

```
{
        MAP_SysCtlDelay(cycles); // Tiva C series specific
}

#ifdef ENABLE_MOTOR
// Setup the PWM peripheral
void SetupPWM()
{
  SysCtlPWMClockSet(SYSCTL_PWMDIV_1);
  SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM0);
  SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOG);
  GPIOPinConfigure(GPIO_PG0_M0PWM4);
  GPIOPinTypePWM(GPIO_PORTG_BASE, GPIO_PIN_0);

  PWMGenConfigure(PWM0_BASE, PWM_GEN_2, PWM_GEN_MODE_UP_DOWN |
                  PWM_GEN_MODE_NO_SYNC);

  //
  // Set the PWM period to 1000Hz.  To calculate the appropriate parameter
  // use the following equation: N = (1 / f) * SysClk.  Where N is the
  // function parameter, f is the desired frequency, and SysClk is the
  // system clock frequency.
  // In this case you get: (1 / 20000Hz) * 120MHz = 6000 cycles.  Note that
  // the maximum period you can set is 2^16.
  // TODO: modify this calculation to use the clock frequency that you are
  // using.
  //
  PWMGenPeriodSet(PWM0_BASE, PWM_GEN_2, 6000);

  // Configure Direction pin
  SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOL);
  MAP_GPIOPinTypeGPIOOutput(GPIO_PORTL_BASE, GPIO_PIN_4);
  MAP_GPIOPadConfigSet(GPIO_PORTL_BASE, GPIO_PIN_4, GPIO_STRENGTH_8MA,
                                GPIO_PIN_TYPE_STD_WPD);
  MAP_GPIOPinWrite(GPIO_PORTL_BASE, GPIO_PIN_4, 0);

}

// Fucntion to set the PWM output given the duty cycle
// PWM can range from -100 to 100, if the value is negative
// we reverse the motion by setting the DIR pin low for the drive
// positive direction correspond to DIR pin being high
int8_t SetPWMDuty(int8_t duty)
{
  // If duty cycle is 0 , disable the PWM generator and output
  if(!duty)
  {
    PWMOutputState(PWM0_BASE, PWM_OUT_4_BIT, false);
    PWMGenDisable(PWM0_BASE, PWM_GEN_2);
  }
  else
  {
    // Set DIR pin accordingly
    if(duty < 0)
    {
      MAP_GPIOPinWrite(GPIO_PORTL_BASE, GPIO_PIN_4, 0);
      duty = 100 - (100 + duty);
    }
    else
      MAP_GPIOPinWrite(GPIO_PORTL_BASE, GPIO_PIN_4, GPIO_PIN_4);

    if(duty == 100)
      duty = 95;

    // Set the PWM pulse width (duty cycle)
    PWMPulseWidthSet(PWM0_BASE, PWM_OUT_4,
                  (PWMGenPeriodGet(PWM0_BASE, PWM_GEN_2) / 100) * (uint32_t)duty);
    PWMOutputState(PWM0_BASE, PWM_OUT_4_BIT, true);
    PWMGenEnable(PWM0_BASE, PWM_GEN_2);
  }
  // Return the set duty cycle
  return duty;
}
```

```
// 5 KHz timer that sets the flag for encoder value transmission
void
Timer0IntHandler(void)
{
    //
    // Clear the timer interrupt.
    //
    ROM_TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

    // Set the flag
    sendEncoder = true;

}
#endif

#ifdef ENABLE_IMU

#ifdef DEV_ADIS16375
// ADIS16375 Interrupt handler
void IntADIS16375(void)
{
  uint32_t status;

  // Clear the interrupt flag
  status = GPIOIntStatus(IMU_IRQ_PORT_BASE, true);

  // Set the appropriate flag
  imuDataReady = 1;

  // Read the desired data from the IMU
  ADIS16375_readAccData(&myIMU, &accel_x, &accel_y, &accel_z);
  ADIS16375_readGyroData(&myIMU, &gyro_x, &gyro_y, &gyro_z);
  //ADIS16375_readDeltaAngle(&myIMU, &delta_x, &delta_y, &delta_z);
  //ADIS16375_readDeltaVel(&myIMU, &dv_x, &dv_y, &dv_z);

  // Value conversion for delta angle displacement
  // Delta angles need to be accumulated to get proper euler angle values
  /*dval_x = (delta_x*1.0)*0.005493;
  dval_y = (delta_y*1.0)*0.005493;
  dval_z = (delta_z*1.0)*0.005493;

  deltaAccX += dval_x;
  deltaAccY += dval_y;
  deltaAccZ += dval_z;*/

  GPIOIntClear(IMU_IRQ_PORT_BASE, status);
}

// Configure the ADIS16375 interrupt pin
void ConfigureADIS16375Int(void)
{
  SysCtlPeripheralEnable(IMU_IRQ_PERIPH);
  GPIOPinTypeGPIOInput(IMU_IRQ_PORT_BASE, IMU_IRQ_PIN);
  GPIOIntTypeSet(IMU_IRQ_PORT_BASE, IMU_IRQ_PIN, GPIO_RISING_EDGE);
  GPIOIntRegister(IMU_IRQ_PORT_BASE, IntADIS16375);
  //GPIOIntEnable(IMU_IRQ_PORT_BASE, IMU_IRQ_PIN);
  IntEnable(IMU_IRQ_INT);
}
#endif


#endif



// Main application
int main(void)
{
   uint32_t status;
   uint8_t printIMU = 0;
   // UART buffer
   uint8_t charUART[256];
   // The first time the IMU gives an interrupt we can set the BIAS NULL
   // command for auto-bias correction on accelerometer and gyroscope data
```

```
   uint8_t firstIMU = 0;

#ifdef ENABLE_UART
    // Character that is used to receive commands from the UART
    // Used only for debugging
    unsigned char uCom = 0;
#endif

#ifdef ENABLE_ETHERNET
    uint32_t ui32User0, ui32User1;
    uint8_t pui8MACArray[8];
    // UDP Send buffer
    uint8_t sendUDP[128];
    // Hold the number of transmission (used for debugging)
    uint32_t sends = 0;
#endif

    // Variable to hold the read encoder value
    int32_t encoderPos = 0;

    // Initialize the application flags
#ifdef ENABLE_MOTOR
    sendEncoder = false;
    setPWMvalue = false;
    pwmValue = 0;
#endif

#ifdef ENABLE_ETHERNET
    gotIP = 0;

    // Set the proper values for lwip configuration based on board selection
#if defined(BOARD_IMU)
    // 192.168.1.10
    device_ip = 0xC0A8010A;
    IP4_ADDR(&board_ip, 0xC0,0xA8,0x01,0x0A);
#elif defined(BOARD_KNEE)
    // // 192.168.1.11
    device_ip = 0xC0A8010B;
    IP4_ADDR(&board_ip, 0xC0,0xA8,0x01,0x0B);
#elif defined(BOARD_HIP)
    // // 192.168.1.12
    device_ip = 0xC0A8010C;
    IP4_ADDR(&board_ip, 0xC0,0xA8,0x01,0x0C);
#endif

    // 255.255.255.0
    device_subnet = 0xFFFFFF00;

    // 192.168.1.1
    device_gateway = 0xC0A80101;

    // 192.168.1.22
    IP4_ADDR(&controller_ip, 0xC0,0xA8,0x01,0x16);

#endif

    // Start the system clock (120 MHz)
    SysCtlMOSCConfigSet(SYSCTL_MOSC_HIGHFREQ);

    g_ui32SysClock = MAP_SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ |
                                             SYSCTL_OSC_MAIN |
                                             SYSCTL_USE_PLL |
                                             SYSCTL_CFG_VCO_480), 120000000);

    // TODO: Enable pins for relay signals -IN CASE OF TREADMILL BOARD
            SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOK);
            SysCtlDelay(10000);
            GPIOPinTypeGPIOOutput(GPIO_PORTK_BASE, GPIO_PIN_7);
            GPIOPinTypeGPIOOutput(GPIO_PORTK_BASE, GPIO_PIN_4);
            GPIOPinTypeGPIOOutput(GPIO_PORTK_BASE, GPIO_PIN_5);
            SysCtlDelay(10000000);
            GPIOPinWrite(GPIO_PORTK_BASE, GPIO_PIN_7, GPIO_PIN_7);
            GPIOPinWrite(GPIO_PORTK_BASE, GPIO_PIN_4, false);               //all on
            GPIOPinWrite(GPIO_PORTK_BASE, GPIO_PIN_5, false);
```

```
            SysCtlDelay(10000000);
            GPIOPinWrite(GPIO_PORTK_BASE, GPIO_PIN_7, true);
            GPIOPinWrite(GPIO_PORTK_BASE, GPIO_PIN_4, GPIO_PIN_4);                    //all off
                GPIOPinWrite(GPIO_PORTK_BASE, GPIO_PIN_5, GPIO_PIN_5);
            SysCtlDelay(10000000);
            GPIOPinWrite(GPIO_PORTK_BASE, GPIO_PIN_7, GPIO_PIN_7);
            GPIOPinWrite(GPIO_PORTK_BASE, GPIO_PIN_4, false);                         //all on
            GPIOPinWrite(GPIO_PORTK_BASE, GPIO_PIN_5, false);
            SysCtlDelay(10000000);
            GPIOPinWrite(GPIO_PORTK_BASE, GPIO_PIN_7, true);
            GPIOPinWrite(GPIO_PORTK_BASE, GPIO_PIN_4, GPIO_PIN_4);                    //all off
                GPIOPinWrite(GPIO_PORTK_BASE, GPIO_PIN_5, GPIO_PIN_5);

#ifdef ENABLE_ETHERNET
    // Set pins for ethernet functionality
    PinoutSet(true, false);
    MAP_GPIOPinTypeGPIOOutput(GPIO_PORTN_BASE, GPIO_PIN_1);
    MAP_GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_1, ~GPIO_PIN_1);
#else
    PinoutSet(false, false);
#endif

#ifdef ENABLE_UART
    ConfigureUART();
#endif

#ifdef ENABLE_ETHERNET
    // Initialize the SysTick timer
    MAP_SysTickPeriodSet(g_ui32SysClock / SYSTICKHZ);
    MAP_SysTickEnable();
    MAP_SysTickIntEnable();

    MAP_FlashUserGet(&ui32User0, &ui32User1);
    if((ui32User0 == 0xffffffff) || (ui32User1 == 0xffffffff))
    {
#ifdef ENABLE_UART
        UARTprintf("No MAC programmed!\n");
#endif
        while(1)
        {
        }
    }

#ifdef ENABLE_UART
    UARTprintf("Waiting for IP.\n");
#endif

    pui8MACArray[0] = ((ui32User0 >>  0) & 0xff);
    pui8MACArray[1] = ((ui32User0 >>  8) & 0xff);
    pui8MACArray[2] = ((ui32User0 >> 16) & 0xff);
    pui8MACArray[3] = ((ui32User1 >>  0) & 0xff);
    pui8MACArray[4] = ((ui32User1 >>  8) & 0xff);
    pui8MACArray[5] = ((ui32User1 >> 16) & 0xff);

    // lwIP stack initialization
    //lwIPInit(g_ui32SysClock, pui8MACArray, 0, 0, 0, IPADDR_USE_DHCP);
    lwIPInit(g_ui32SysClock, pui8MACArray, device_ip, device_subnet, device_gateway,
IPADDR_USE_STATIC);

    MAP_IntPrioritySet(INT_EMAC0, ETHERNET_INT_PRIORITY);
#endif

    MAP_IntPrioritySet(FAULT_SYSTICK, SYSTICK_INT_PRIORITY);

    // Wait until an IP is assigned
#ifdef ENABLE_ETHERNET
    while(gotIP == 0)
        SysCtlDelay(120);
#endif

#ifdef ENABLE_UART
    UARTprintf("Initializing...\n");
#endif
```

```
    // Configure motor interface modules
#ifdef ENABLE_MOTOR
    // Setup the PWM generator
    SetupPWM();

    // QEI Setup
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOL);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_QEI0);
    GPIOPinConfigure(GPIO_PL3_IDX0);
    GPIOPinConfigure(GPIO_PL1_PHA0);
    GPIOPinConfigure(GPIO_PL2_PHB0);
    GPIOPinTypeQEI(GPIO_PORTL_BASE, GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3);

    QEIConfigure(QEI0_BASE, (QEI_CONFIG_CAPTURE_A_B | QEI_CONFIG_RESET_IDX | QEI_CONFIG_QUADRATURE |
QEI_CONFIG_NO_SWAP), 52*2000-1);        QEIVelocityConfigure(QEI0_BASE, QEI_VELDIV_1, 8000000);
    QEIVelocityConfigure(QEI0_BASE, QEI_VELDIV_1, 8000000); // for the treadmill


    //
    // Enable the quadrature encoder.
    //
    QEIEnable(QEI0_BASE);
    QEIPositionSet(QEI0_BASE,0);
    //
    // Delay for some time...
    //
    SysCtlDelay(12000);
#endif

    // Initialize the UDP receive pcb
#ifdef ENABLE_ETHERNET
    Rpcb = udp_init_r();
#endif

    // IMU Initializaztion
#ifdef ENABLE_IMU

#ifdef DEV_ADIS16375
    ADIS16375_Init(&myIMU, cyclesdelay, IMU_CS, IMU_RST, init_spi16, SpiTransfer16);
#ifdef ENABLE_UART
    //UARTprintf("Prod ID : 0x%X\n",ADIS16375_device_id(&myIMU));
    //ADIS16375_write(&myIMU,ADIS16375_REG_GLOB_CMD,0x8000);
#endif
    //ADIS16375_debug(&myIMU);
    ConfigureADIS16375Int();
    //status = GPIOIntStatus(IMU_IRQ_PORT_BASE, true);

    // Clear interrupt flag just to be safe
    GPIOIntClear(IMU_IRQ_PORT_BASE, IMU_IRQ_PIN);
    GPIOIntEnable(IMU_IRQ_PORT_BASE, IMU_IRQ_PIN);

    //ADIS16375_wake(&myIMU);
    //init_spi16();
    //UARTprintf("Prod ID : 0x%X\n",ADIS16375_device_id(&myIMU));
#endif


#endif

    // Configure 5 KHz tier for encoder count acquisition
#ifdef ENABLE_MOTOR
    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
    TimerConfigure(TIMER0_BASE, TIMER_CFG_PERIODIC);
    TimerLoadSet(TIMER0_BASE, TIMER_A, g_ui32SysClock/5000); //here the transmitting frequency is
defined
    IntEnable(INT_TIMER0A);
    TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
    TimerEnable(TIMER0_BASE, TIMER_A);
#endif

    // Application Main Loop
    while(1)
    {
#ifdef ENABLE_IMU
```

```
      // If data ready from IMU output the data
      // On first interrupt only we check the product ID
      // and are able to set the configuration for proper delta angle calculation
      if(imuDataReady == 1)
      {
        if(!firstIMU)
        {
          firstIMU = 1;
#ifdef DEV_ADIS16375
          UARTprintf("Prod ID : 0x%X\n",ADIS16375_device_id(&myIMU));
#endif


#ifdef DEV_ADIS16375
          // Resore factory calibration on strat-up
          ADIS16375_write(&myIMU, ADIS16375_REG_GLOB_CMD, 0x4000);
          // Configure the ADIS16375 IMU
          //MAP_SysCtlDelay(40000*100);

          // Set the decimation coefficient
          //ADIS16375_write(&myIMU, ADIS16375_REG_DEC_RATE, DECIMATION_COEF);

          // Set configuration for the BIAS estimator
          ADIS16375_write(&myIMU, ADIS16375_REG_NULL_CFG, 0x0A07);

          // Load values for bias correction (BIAS NULL command)
          //ADIS16375_write(&myIMU, ADIS16375_REG_GLOB_CMD, 0x0100);

          //GPIOIntDisable(IMU_IRQ_PORT_BASE, IMU_IRQ_PIN);
#endif
          imuDataReady = 0;
          deltaAccX = deltaAccY = deltaAccZ = 0.0;
        }
        else
        {
          imuDataReady = 0;
#ifdef ENABLE_ETHERNET
#ifdef DEV_ADIS16375
          sendUDP[0] = 0x43;
#endif
          memcpy(&sendUDP[1],(uint8_t*)(&accel_x),2);
          memcpy(&sendUDP[3],(uint8_t*)(&accel_y),2);
          memcpy(&sendUDP[5],(uint8_t*)(&accel_z),2);
          memcpy(&sendUDP[7],(uint8_t*)(&gyro_x),2);
          memcpy(&sendUDP[9],(uint8_t*)(&gyro_y),2);
          memcpy(&sendUDP[11],(uint8_t*)(&gyro_z),2);
          udp_send_data((void*)sendUDP,13);
#endif
          // Handle the recieved IMU measrements
#ifdef ENABLE_UART
          if(printIMU)
          {
            printIMU = 0;
#ifdef DEV_ADIS16375
            UARTprintf("ACC_X_OUT : %d 0x%X\n",accel_x,accel_x);
            UARTprintf("ACC_Y_OUT : %d 0x%X\n",accel_y,accel_y);
            UARTprintf("ACC_Z_OUT : %d 0x%X\n",accel_z,accel_z);

            UARTprintf("GYRO_X_OUT : %d 0x%X\n",gyro_x,gyro_x);
            UARTprintf("GYRO_Y_OUT : %d 0x%X\n",gyro_y,gyro_y);
            UARTprintf("GYRO_Z_OUT : %d 0x%X\n",gyro_z,gyro_z);

            dval_x = (gyro_x*1.0)*0.013108;
            dval_y = (gyro_y*1.0)*0.013108;
            dval_z = (gyro_z*1.0)*0.013108;

            sprintf(charUART, "GYRO X : %lf\n", dval_x);
            UARTprintf("%s",charUART);
            sprintf(charUART, "GYRO Y : %lf\n", dval_y);
            UARTprintf("%s",charUART);
            sprintf(charUART, "GYRO Z : %lf\n", dval_z);
            UARTprintf("%s",charUART);

            dval_x = (accel_x*1.0)*0.8192;
```

```
            dval_y = (accel_y*1.0)*0.8192;
            dval_z = (accel_z*1.0)*0.8192;

            sprintf(charUART, "ACC X : %lf\n", dval_x);
            UARTprintf("%s",charUART);
            sprintf(charUART, "ACC Y : %lf\n", dval_y);
            UARTprintf("%s",charUART);
            sprintf(charUART, "ACC Z : %lf\n", dval_z);
            UARTprintf("%s",charUART);

            dval_x = (delta_x*1.0)*0.005493;
            dval_y = (delta_y*1.0)*0.005493;
            dval_z = (delta_z*1.0)*0.005493;

            sprintf(charUART, "DELTA X : 0x%X %lf\n", delta_x, dval_x);
            UARTprintf("%s",charUART);
            sprintf(charUART, "DELTA Y : 0x%X %lf\n", delta_y, dval_y);
            UARTprintf("%s",charUART);
            sprintf(charUART, "DELTA Z : 0x%X %lf\n", delta_z, dval_z);
            UARTprintf("%s",charUART);

            dval_x = (dv_x*1.0)*3.0518;
            dval_y = (dv_y*1.0)*3.0518;
            dval_z = (dv_z*1.0)*3.0518;

            sprintf(charUART, "DELTA VEL X : %lf\n", dval_x);
            UARTprintf("%s",charUART);
            sprintf(charUART, "DELTA VEL Y : %lf\n", dval_y);
            UARTprintf("%s",charUART);
            sprintf(charUART, "DELTA VEL Z : %lf\n", dval_z);
            UARTprintf("%s",charUART);
#endif

          }
        }
#endif
      }
#endif
      // UART command interface for debugging
#ifdef ENABLE_UART
#ifdef UART_BUFFERED
      if(UARTRxBytesAvail()>0)
      {
        uCom = UARTgetc();
        UARTFlushRx();
      }
#endif
      switch(uCom)
      {
#ifdef ENABLE_IMU
      case '1' :
        // Get IMU product ID
#ifdef DEV_ADIS16375
        UARTprintf("Prod ID : 0x%X\n",ADIS16375_device_id(&myIMU));
#endif

        break;
      case '2':
        // Reset IMU measurement data
        imuDataReady = 0;
        accel_x = accel_y = accel_z = 0;
        deltaAccX = deltaAccY = deltaAccZ = 0.0;
        status = GPIOIntStatus(IMU_IRQ_PORT_BASE, true);
        GPIOIntClear(IMU_IRQ_PORT_BASE, status);
        GPIOIntEnable(IMU_IRQ_PORT_BASE, IMU_IRQ_PIN);
        break;
      case '3':
        // Get data directly from IMU without waiting for interrupt
#ifdef DEV_ADIS16375
        ADIS16375_readAccData(&myIMU, &accel_x, &accel_y, &accel_z);
        ADIS16375_readGyroData(&myIMU, &gyro_x, &gyro_y, &gyro_z);
        ADIS16375_readDeltaAngle(&myIMU, &delta_x, &delta_y, &delta_z);
        ADIS16375_readDeltaVel(&myIMU, &dv_x, &dv_y, &dv_z);
#endif
```

```
        dval_x = (gyro_x*1.0)*0.013108;
        dval_y = (gyro_y*1.0)*0.013108;
        dval_z = (gyro_z*1.0)*0.013108;

        sprintf(charUART, "GYRO X : %lf\n", dval_x);
        UARTprintf("%s",charUART);
        sprintf(charUART, "GYRO Y : %lf\n", dval_y);
        UARTprintf("%s",charUART);
        sprintf(charUART, "GYRO Z : %lf\n", dval_z);
        UARTprintf("%s",charUART);

        dval_x = (accel_x*1.0)*0.8192;
        dval_y = (accel_y*1.0)*0.8192;
        dval_z = (accel_z*1.0)*0.8192;

        sprintf(charUART, "ACC X : %lf\n", dval_x);
        UARTprintf("%s",charUART);
        sprintf(charUART, "ACC Y : %lf\n", dval_y);
        UARTprintf("%s",charUART);
        sprintf(charUART, "ACC Z : %lf\n", dval_z);
        UARTprintf("%s",charUART);

        dval_x = (delta_x*1.0)*0.005493;
        dval_y = (delta_y*1.0)*0.005493;
        dval_z = (delta_z*1.0)*0.005493;

        sprintf(charUART, "DELTA X : %lf\n", dval_x);
        UARTprintf("%s",charUART);
        sprintf(charUART, "DELTA Y : %lf\n", dval_y);
        UARTprintf("%s",charUART);
        sprintf(charUART, "DELTA Z : %lf\n", dval_z);
        UARTprintf("%s",charUART);

        dval_x = (dv_x*1.0)*3.0518;
        dval_y = (dv_y*1.0)*3.0518;
        dval_z = (dv_z*1.0)*3.0518;

        sprintf(charUART, "DELTA VEL X : %lf\n", dval_x);
        UARTprintf("%s",charUART);
        sprintf(charUART, "DELTA VEL Y : %lf\n", dval_y);
        UARTprintf("%s",charUART);
        sprintf(charUART, "DELTA VEL Z : %lf\n", dval_z);
        UARTprintf("%s",charUART);
        break;
      case '4' :
        // Output received IMU data
        printIMU = 1;
        break;
#ifdef DEV_ADIS16375
      case '5' :
        // Read and display IMU internal temperature
        temp_out = ADIS16375_temp(&myIMU);
        temp = (temp_out*1.0)*0.00565 + 25.0;
        sprintf(charUART, "%lf", temp);
        UARTprintf("Temp : 0x%X %s\n",temp_out,charUART);
        break;
      case '6' :
        // Output bias coefficients
        UARTprintf("X_GYRO_OFF_L : 0x%X\n",ADIS16375_read(&myIMU, 16, ADIS16375_REG_X_GYRO_OFF_L));
        UARTprintf("X_GYRO_OFF_H : 0x%X\n",ADIS16375_read(&myIMU, 16, ADIS16375_REG_X_GYRO_OFF_H));
        UARTprintf("Y_GYRO_OFF_L : 0x%X\n",ADIS16375_read(&myIMU, 16, ADIS16375_REG_Y_GYRO_OFF_L));
        UARTprintf("Y_GYRO_OFF_H : 0x%X\n",ADIS16375_read(&myIMU, 16, ADIS16375_REG_Y_GYRO_OFF_H));
        UARTprintf("Z_GYRO_OFF_L : 0x%X\n",ADIS16375_read(&myIMU, 16, ADIS16375_REG_Z_GYRO_OFF_L));
        UARTprintf("Z_GYRO_OFF_H : 0x%X\n",ADIS16375_read(&myIMU, 16, ADIS16375_REG_Z_GYRO_OFF_H));
        UARTprintf("X_ACC_OFF_L : 0x%X\n",ADIS16375_read(&myIMU, 16, ADIS16375_REG_X_ACC_OFF_L));
        UARTprintf("X_ACC_OFF_H : 0x%X\n",ADIS16375_read(&myIMU, 16, ADIS16375_REG_X_ACC_OFF_H));
        UARTprintf("Y_ACC_OFF_L : 0x%X\n",ADIS16375_read(&myIMU, 16, ADIS16375_REG_Y_ACC_OFF_L));
        UARTprintf("Y_ACC_OFF_H : 0x%X\n",ADIS16375_read(&myIMU, 16, ADIS16375_REG_Y_ACC_OFF_H));
        UARTprintf("Z_ACC_OFF_L : 0x%X\n",ADIS16375_read(&myIMU, 16, ADIS16375_REG_Z_ACC_OFF_L));
        UARTprintf("Z_ACC_OFF_H : 0x%X\n",ADIS16375_read(&myIMU, 16, ADIS16375_REG_Z_ACC_OFF_H));
        UARTprintf("X_GYRO_SCALE : 0x%X\n",ADIS16375_read(&myIMU, 16, ADIS16375_REG_X_GYRO_SCALE));
        UARTprintf("Y_GYRO_SCALE : 0x%X\n",ADIS16375_read(&myIMU, 16, ADIS16375_REG_Y_GYRO_SCALE));
        UARTprintf("Z_GYRO_SCALE : 0x%X\n",ADIS16375_read(&myIMU, 16, ADIS16375_REG_Z_GYRO_SCALE));
```

```
        UARTprintf("X_ACC_SCALE : 0x%X\n",ADIS16375_read(&myIMU, 16, ADIS16375_REG_X_ACCEL_SCALE));
        UARTprintf("Y_ACC_SCALE : 0x%X\n",ADIS16375_read(&myIMU, 16, ADIS16375_REG_Y_ACCEL_SCALE));
        UARTprintf("Z_ACC_SCALE : 0x%X\n",ADIS16375_read(&myIMU, 16, ADIS16375_REG_Z_ACCEL_SCALE));
        UARTprintf("GEN_CONFIG : 0x%X\n",(ADIS16375_read(&myIMU, 16, ADIS16375_REG_GEN_CFG) &
0x00FF));
        UARTprintf("NULL_CONFIG : 0x%X\n",(ADIS16375_read(&myIMU, 16, ADIS16375_REG_NULL_CFG) &
0x3FFF));
        UARTprintf("DEC_RATE : 0x%X\n",(ADIS16375_read(&myIMU, 16, ADIS16375_REG_DEC_RATE) & 0x07FF));
        break;
      case '7':
        // Load calibration values
        ADIS16375_write(&myIMU, ADIS16375_REG_GLOB_CMD, 0x0100);
        break;
      case '8':
        // Output delta angle from stored accumulated data
        sprintf(charUART, "DELTA X : %lf\n", deltaAccX);
        UARTprintf("%s",charUART);
        sprintf(charUART, "DELTA Y : %lf\n", deltaAccY);
        UARTprintf("%s",charUART);
        sprintf(charUART, "DELTA Z : %lf\n", deltaAccZ);
        UARTprintf("%s",charUART);
        break;
#endif
      case 'w':
        // IMU wakeup
#ifdef DEV_ADIS16375
        ADIS16375_wake(&myIMU);
#endif

        break;
#endif
      default : break;
      }
      uCom = 0;
#endif

#ifdef ENABLE_MOTOR
      // If we received a PWM command set the corresponding PWM duty cyle and direction
      if(setPWMvalue == true)
      {
          if (pwmValue == 103)
          {
                      UARTprintf("enableccw");
                          GPIOPinWrite(GPIO_PORTK_BASE, GPIO_PIN_4, GPIO_PIN_4);
                          GPIOPinWrite(GPIO_PORTK_BASE, GPIO_PIN_5, false);
                          UARTprintf("enableccw");
          }
          if (pwmValue == 101)
          {
                       UARTprintf("kill");
                              GPIOPinWrite(GPIO_PORTK_BASE, GPIO_PIN_7, true);
                              GPIOPinWrite(GPIO_PORTK_BASE, GPIO_PIN_4, GPIO_PIN_4);
        //all off
                              GPIOPinWrite(GPIO_PORTK_BASE, GPIO_PIN_5, GPIO_PIN_5);
                       UARTprintf("kill");
          }
          if (pwmValue == 102)
          {
                       UARTprintf("enable");
                       GPIOPinWrite(GPIO_PORTK_BASE, GPIO_PIN_7, GPIO_PIN_7);
                       UARTprintf("enable");
          }
          if (pwmValue == 104)
          {
                    UARTprintf("enablecw");
                    GPIOPinWrite(GPIO_PORTK_BASE, GPIO_PIN_4, false);
                    GPIOPinWrite(GPIO_PORTK_BASE, GPIO_PIN_5, GPIO_PIN_5);
                    UARTprintf("enablecw");
          }
          else
                  {
                       UARTprintf("else");
                   SetPWMDuty(pwmValue);
                       UARTprintf("else");
```

```
                }
        setPWMvalue = false;
      }

      // If send encoder value event occurs, send data via UDP
      if(sendEncoder == true)
      {
        //sends++;
        encoderPos = (QEIPositionGet(QEI0_BASE));
        //encoderPos = (QEIVelocityGet(QEI0_BASE)); for the treadmill

        //if(sends == 5000)
        //{
          //UARTprintf("Position %d\n",encoderPos);
          //sends = 0;
        //}
        sendEncoder = false;
#ifdef ENABLE_ETHERNET
        sendUDP[0] = 0x42;
        memcpy(&sendUDP[1],(uint8_t*)(&encoderPos),4);
        udp_send_data((void*)sendUDP,5);
#endif
      }
#endif
    }
}

#ifdef ENABLE_ETHERNET

// Initializze the UDP receive pcb
struct udp_pcb * udp_init_r(void)
{
  //err_t err;
  struct udp_pcb *pcb_r;
  pcb_r = udp_new();

  // Bind to given port , receive from any IP
  udp_bind(pcb_r, IP_ADDR_ANY, PORT_R);

#ifdef ENABLE_UART
  UARTprintf("UDP to receive at port %d...\n", PORT_R);
#endif

  // Set the receive data callback
  udp_recv(pcb_r, udp_receive_data, NULL);

  return pcb_r;
}

void udp_receive_data(void *arg, struct udp_pcb *pcb, struct pbuf *p, struct ip_addr *addr, u16_t
port)
{
    char * pPointer;

    //struct pbuf *p1;

    if (p != NULL)
    {
      //UARTwrite((char*)(p->payload), p->len);
      //UARTprintf("R : %s\n",(char*)(p->payload));

      pPointer = (char*)(p->payload);

     /* p1 = pbuf_alloc(PBUF_TRANSPORT,8,PBUF_RAM);
      memcpy (p1->payload, pData, 8);
      udp_send(pcb, p1);
      pbuf_free(p1);*/

      /*if(pPointer[0] == 0x31)
      {
          udp_send_data((void*)pData, 68);
      }*/
      // If we received PWM commnad (0x31 command byte)
      // extract the transmitted value
```

```
      if(pPointer[0] == 0x31)
      {
        pwmValue = pPointer[1];
        setPWMvalue = true;
      }

      /*if(pPointer[0] == 0x32)
      {
        sendEncoder = true;
      }*/

      pbuf_free(p);
    }
}

// Send data over UDP to the defined port
void udp_send_data(void* sbuf, u16_t len)
{
  struct pbuf *p;
  err_t err;

  p = pbuf_alloc(PBUF_TRANSPORT,len,PBUF_RAM);
  memcpy (p->payload, sbuf, len);
  err = udp_sendto(Rpcb, p, &controller_ip, PORT_S);

  pbuf_free(p);
}

#endif
```

Joint_state_publisher

```
#include <boost/algorithm/string.hpp>
#include <gazebo_plugins/gazebo_ros_joint_state_publisher.h>
#include <tf/transform_broadcaster.h>
#include <tf/transform_listener.h>

using namespace gazebo;

GazeboRosJointStatePublisher::GazeboRosJointStatePublisher() {}

// Destructor
GazeboRosJointStatePublisher::~GazeboRosJointStatePublisher() {
    rosnode_->shutdown();
}

void GazeboRosJointStatePublisher::Load ( physics::ModelPtr _parent, sdf::ElementPtr _sdf ) {
    // Store the pointer to the model
    this->parent_ = _parent;
    this->world_ = _parent->GetWorld();

    this->robot_namespace_ = parent_->GetName ();
    if ( !_sdf->HasElement ( "robotNamespace" ) ) {
        ROS_INFO ( "GazeboRosJointStatePublisher Plugin missing <robotNamespace>, defaults to \"%s\"",
                   this->robot_namespace_.c_str() );
    } else {
        this->robot_namespace_ = _sdf->GetElement ( "robotNamespace" )->Get<std::string>();
        if ( this->robot_namespace_.empty() ) this->robot_namespace_ = parent_->GetName ();
    }
    if ( !robot_namespace_.empty() ) this->robot_namespace_ += "/";
    rosnode_ = boost::shared_ptr<ros::NodeHandle> ( new ros::NodeHandle ( this->robot_namespace_ ) );

    if ( !_sdf->HasElement ( "jointName" ) ) {
        ROS_ASSERT ( "GazeboRosJointStatePublisher Plugin missing jointNames" );
    } else {
        sdf::ElementPtr element = _sdf->GetElement ( "jointName" ) ;
        std::string joint_names = element->Get<std::string>();
        boost::erase_all ( joint_names, " " );
        boost::split ( joint_names_, joint_names, boost::is_any_of ( "," ) );
    }

    this->update_rate_ = 100.0;
```

```
    if ( !_sdf->HasElement ( "updateRate" ) ) {
        ROS_WARN ( "GazeboRosJointStatePublisher Plugin (ns = %s) missing <updateRate>, defaults to
%f",
                   this->robot_namespace_.c_str(), this->update_rate_ );
    } else {
        this->update_rate_ = _sdf->GetElement ( "updateRate" )->Get<double>();
    }

    // Initialize update rate stuff
    if ( this->update_rate_ > 0.0 ) {
        this->update_period_ = 1.0 / this->update_rate_;
    } else {
        this->update_period_ = 0.0;
    }
    last_update_time_ = this->world_->GetSimTime();

    for ( unsigned int i = 0; i< joint_names_.size(); i++ ) {
        joints_.push_back ( this->parent_->GetJoint ( joint_names_[i] ) );
        ROS_INFO ( "GazeboRosJointStatePublisher is going to publish joint: %s",
joint_names_[i].c_str() );
    }

    ROS_INFO ( "Starting GazeboRosJointStatePublisher Plugin (ns = %s)!, parent name: %s", this-
>robot_namespace_.c_str(), parent_->GetName ().c_str() );

    tf_prefix_ = tf::getPrefixParam ( *rosnode_ );
    joint_state_publisher_ = rosnode_->advertise<sensor_msgs::JointState> ( "joint_states",1000 );

    last_update_time_ = this->world_->GetSimTime();
    // Listen to the update event. This event is broadcast every
    // simulation iteration.
    this->updateConnection = event::Events::ConnectWorldUpdateBegin (
                                 boost::bind ( &GazeboRosJointStatePublisher::OnUpdate, this, _1 ) );
}

void GazeboRosJointStatePublisher::OnUpdate ( const common::UpdateInfo & _info ) {
    // Apply a small linear velocity to the model.
    common::Time current_time = this->world_->GetSimTime();
    double seconds_since_last_update = ( current_time - last_update_time_ ).Double();
    if ( seconds_since_last_update > update_period_ ) {

        publishJointStates();

        last_update_time_+= common::Time ( update_period_ );

    }

}

void GazeboRosJointStatePublisher::publishJointStates() {
    ros::Time current_time = ros::Time::now();

    joint_state_.header.stamp = current_time;
    joint_state_.name.resize ( joints_.size() );
    joint_state_.position.resize ( joints_.size() );
    joint_state_.velocity.resize ( joints_.size() );
    joint_state_.effort.resize ( joints_.size() );



    for ( int i = 0; i < joints_.size(); i++ ) {
        physics::JointPtr joint = joints_[i];
        math::Angle angle = joint->GetAngle ( 0 );
        double veloc = joint->GetVelocity ( 0 );
        double eff = joint->GetForce ( 0 );


        joint_state_.name[i] = joint->GetName();
        joint_state_.position[i] = angle.Radian () ;
        joint_state_.velocity[i] = veloc ;
        joint_state_.effort[i] = eff ;
    }
    joint_state_publisher_.publish ( joint_state_ );
}
```

simulation.world

```
<?xml version='1.0'?>
<sdf version='1.6'>

  <world name="default">

    <physics type="ode">
      <gravity>0 0 -9.81</gravity>
      <max_step_size>0.001</max_step_size>
      <real_time_factor>1</real_time_factor>
      <real_time_update_rate>10.0</real_time_update_rate>
      <ode>
        <solver>
          <type>quick</type>
          <iters>100</iters>
          <sor>1.3</sor>
        </solver>
        <constraints>
          <cfm>0</cfm>
          <erp>0.2</erp>
          <contact_max_correcting_vel>100.0</contact_max_correcting_vel>
          <contact_surface_layer>0.001</contact_surface_layer>
        </constraints>
      </ode>
    </physics>

    <include>
      <uri>model://ground_plane</uri>
  </include>

  <include>
      <uri>model://Monopod</uri>
  </include>

    <!-- Global light source -->
    <include>
      <uri>model://sun</uri>
    </include>

    <gui fullscreen='0'>
      <camera name='user_camera'>
        <pose>4.927360 -4.376610 3.740080 0.000000 0.275643 2.356190</pose>
        <view_controller>orbit</view_controller>
      </camera>
    </gui>

  </world>
</sdf>
```

monopodplugin.cc

```
#ifndef _MONOPOD_PLUGIN_HH_
#define _MONOPOD_PLUGIN_HH_

#include <gazebo/gazebo.hh>
#include <gazebo/physics/physics.hh>
#include <gazebo/transport/transport.hh>
#include <gazebo/msgs/msgs.hh>
#include <thread>
#include "ros/ros.h"
#include "ros/callback_queue.h"
#include "ros/subscribe_options.h"
#include "std_msgs/Float64.h"
#include <boost/algorithm/string.hpp>
#include <tf/transform_broadcaster.h>
#include <tf/transform_listener.h>
#include <boost/bind.hpp>
#include <gazebo/gazebo.hh>
#include <gazebo/physics/physics.hh>
#include <gazebo/common/common.hh>
```

```
#include <stdio.h>
#include <math.h>

namespace gazebo
{
  /// \brief A plugin to control a Monopod sensor.
  class MonopodPlugin : public ModelPlugin
  {
    /// \brief Constructor
    public: MonopodPlugin() {}

    public: std::string joint_names ;

    /// \brief The load function is called by Gazebo when the plugin is
    /// inserted into simulation
    /// \param[in] _model A pointer to the model that this plugin is
    /// attached to.
    /// \param[in] _sdf A pointer to the plugin's SDF element.

    /// \brief A node use for ROS transport
    private: std::unique_ptr<ros::NodeHandle> rosNode;

    /// \brief A ROS subscriber
    private: ros::Subscriber rosSub;

    /// \brief A ROS callbackqueue that helps process messages
    private: ros::CallbackQueue rosQueue;

    /// \brief A thread the keeps running the rosQueue
    private: std::thread rosQueueThread;

    public: void SetJointPosition(const std::string &_jointName, double _position);

    public: virtual void Load(physics::ModelPtr _model, sdf::ElementPtr _sdf)
    {

      // Safety check
      if (_model->GetJointCount() == 0)
      {
        std::cerr << "Invalid joint count, Monopod plugin not loaded\n";
        return;
      }

      // Store the model pointer for convenience.
      this->model = _model;


      if ( !_sdf->HasElement ( "jointName" ) ) {
        ROS_ASSERT ( "Plugin missing jointNames to send commands to" );
      } else {
        sdf::ElementPtr element = _sdf->GetElement ( "jointName" ) ;
        joint_names = element->Get<std::string>();
        boost::erase_all ( joint_names, " " );
        boost::split ( joint_names_, joint_names, boost::is_any_of ( "," ) );
      }


      // Initialize ros, if it has not already been initialized.
      if (!ros::isInitialized())
      {
        int argc = 0;
        char **argv = NULL;
        ros::init(argc, argv, "gazebo_client",
        ros::init_options::NoSigintHandler);
      }

      // Create our ROS node. This acts in a similar manner to
      // the Gazebo node
      this->rosNode.reset(new ros::NodeHandle("gazebo_client"));

      // Create a named topic, and subscribe to it.
      ros::SubscribeOptions so =
      ros::SubscribeOptions::create<std_msgs::Float64>(
      "/" + this->model->GetName() + "/hip_torque_cmd",
```

```
      1,
      boost::bind(&MonopodPlugin::OnRosMsg, this, _1),
      ros::VoidPtr(), &this->rosQueue);
      this->rosSub = this->rosNode->subscribe(so);

      // Spin up the queue helper thread.
      this->rosQueueThread =
      std::thread(std::bind(&MonopodPlugin::QueueThread, this));

      model->GetJoint("hip")->SetPosition(0 , 0.0) ;
      model->GetJoint("z")->SetPosition(0 , 0.1) ; //to set the initial height
      model->GetJoint("x")->SetVelocity(0 , 0.0) ; //to set the initial velocity

    }

    /// \brief Set the hip torque
    /// \param[in] _vel New target velocity
    public: void SetTorque(const double &_trq)
    {
      model->GetJoint("hip")->SetForce( 0 ,_trq);
    }

    /// \brief Handle an incoming message from ROS
    /// \param[in] _msg A float value that is used to set the velocity
    /// of the Monopod
    public: void OnRosMsg(const std_msgs::Float64ConstPtr &_msg)
    {
        this->SetTorque(_msg->data);

    }

    /// \brief ROS helper function that processes messages
    private: void QueueThread()
    {
        static const double timeout = 0.01;
        while (this->rosNode->ok())
      {
        this->rosQueue.callAvailable(ros::WallDuration(timeout));
      }
    }

        /// \brief Pointer to the model.
    private: physics::ModelPtr model;

    private: std::vector<std::string> joint_names_;

  };

  // Tell Gazebo about this plugin, so that Gazebo can call Load on this plugin.
  GZ_REGISTER_MODEL_PLUGIN(MonopodPlugin)
}
#endif
```

model.sdf

```
<?xml version='1.0'?>
<sdf version='1.6'>
  <model name="Assem1">
    <static>false</static>

  <link name="Lower_leg">
      <pose>0.3 0 0.2 0 0 0</pose>
      <must_be_base_link>0</must_be_base_link>
      <inertial>
        <pose>0 0 0.00903 0 0 0</pose>
        <mass>0.338</mass>
        <inertia>
          <ixx>0.00409816</ixx>
          <ixy>0</ixy>
          <ixz>0</ixz>
          <iyy>0.00409816</iyy>
          <iyz>0</iyz>
```

```
          <izz>0.00409816</izz>
        </inertia>
      </inertial>
      <collision name="Lower_leg_collision">
        <geometry>
           <cylinder>
              <radius>.004</radius>
              <length>.4</length>
           </cylinder>
        </geometry>
        <surface>
          <friction>
            <ode>
              <mu>10000.8</mu>
              <mu2>10000.8</mu2>
            </ode>
          </friction>
        </surface>
      </collision>
      <visual name="Lower_leg_visual">
        <material>
          <ambient>10 0 0 10</ambient>
          <diffuse>10 0 0 10</diffuse>
        </material>
        <geometry>
           <cylinder>
              <radius>.004</radius>
              <length>.4</length>
           </cylinder>
        </geometry>
      </visual>
    </link>

    <link name="Upper_leg">
      <pose>0.3 0 0.35 0 0 0</pose>
      <must_be_base_link>0</must_be_base_link>
      <inertial>
        <pose>0 0 0.00193 0 0 0</pose>
        <mass>0.513</mass>
        <inertia>
<!--            <ixx>0.0240557107408</ixx>
 -->
          <ixx>0.0240557107408</ixx>
          <ixy>0</ixy>
          <ixz>0</ixz>
          <iyy>0.0240557107408</iyy>
          <iyz>0</iyz>
          <izz>0.0240557107408</izz>
        </inertia>
      </inertial>
      <collision name="Upper_leg_collision">
        <geometry>
           <cylinder>
              <radius>.01</radius>
              <length>.1</length>
           </cylinder>
        </geometry>
      </collision>
      <visual name="Upper_leg_visual">
        <material>
          <ambient>10 0 0 10</ambient>
          <diffuse>10 0 0 10</diffuse>
        </material>
        <geometry>
           <cylinder>
              <radius>.01</radius>
              <length>.1</length>
           </cylinder>
        </geometry>
      </visual>
    </link>


    <joint name="spring" type="prismatic">
```

```
  <parent>Upper_leg</parent>
  <child>Lower_leg</child>
  <pose>0 0 0 0 0 0</pose>
  <axis>
    <xyz>0 0 1</xyz>
  <dynamics>
    <damping>0.05</damping>
      <friction>0.2</friction>
    <spring_reference>0.0</spring_reference>
    <spring_stiffness>6279.0</spring_stiffness>
  </dynamics>
    <limit>
      <upper>0.2</upper>
      <lower>0.0</lower>
      <effort>1</effort>
    </limit>
  </axis>
</joint>


<link name="Body">
  <pose>0.3 0 0.5 0 0 0</pose>
  <inertial>
    <mass>6.7</mass>
    <inertia>
      <ixx>0.002048</ixx>
      <ixy>0</ixy>
      <ixz>0</ixz>
      <iyy>0.00182272</iyy>
      <iyz>0</iyz>
      <izz>0.00124928</izz>
    </inertia>
  </inertial>
  <collision name="Body_collision">
    <geometry>
       <box>
        <size>0.3 0.3 0.3</size>
      </box>
    </geometry>
  </collision>
  <visual name="Body_visual">
    <material>
      <ambient>1 0 0 1</ambient>
      <diffuse>1 0 0 1</diffuse>
    </material>
    <geometry>
       <box>
        <size>0.3 0.3 0.3</size>
      </box>
    </geometry>
  </visual>
</link>

<joint name="hip" type="revolute">
  <parent>Body</parent>
  <child>Upper_leg</child>
  <pose>0 0 0 0 0 0</pose>
  <axis>
    <xyz>1 0 0</xyz>
    <dynamics>
      <damping>0.005</damping>
      <friction>0.1</friction>
    </dynamics>
    <limit>
      <upper>0.5</upper>
      <lower>-0.5</lower>
      <effort>4.812</effort>
    </limit>
  </axis>
</joint>

<link name="Base4">
  <pose>0.125 0 0.575 0 0 0</pose>
  <must_be_base_link>0</must_be_base_link>
```

```
  <inertial>
    <mass>0.001</mass>
    <inertia>
      <ixx>0.00177083333333334</ixx>
      <ixy>0</ixy>
      <ixz>0</ixz>
      <iyy>0.00208333333333333</iyy>
      <iyz>0</iyz>
      <izz>0.00177083333333334</izz>
    </inertia>
  </inertial>
  <collision name="Base4_collision">
    <geometry>
      <box>
        <size>0.05 0.05 0.2</size>
      </box>
    </geometry>
    <surface>
      <friction>
        <ode>
          <mu>0.0</mu>
          <mu2>0.0</mu2>
        </ode>
      </friction>
    </surface>
  </collision>
  <visual name="Base4_visual">
    <material>
      <ambient>1 0 0 1</ambient>
      <diffuse>1 0 0 1</diffuse>
    </material>
    <geometry>
       <box>
        <size>0.05 0.05 0.2</size>
      </box>
    </geometry>
  </visual>
</link>

<joint name="Base4-Body" type="fixed">
  <parent>Base4</parent>
  <child>Body</child>
  <pose>-0.15 0 0 0 0 0</pose>
</joint>

 <link name="Base3">
  <pose>0 0 0.575 0 0 0</pose>
  <must_be_base_link>0</must_be_base_link>
  <inertial>
    <mass>0.001</mass>
    <inertia>
      <ixx>0.00260416666666667</ixx>
      <ixy>0</ixy>
      <ixz>0</ixz>
      <iyy>0.0060416666666667</iyy>
      <iyz>0</iyz>
      <izz>0.004166666666667</izz>
    </inertia>
  </inertial>
  <collision name="Base3_collision">
    <geometry>
      <box>
        <size>0.2 0.05 0.05</size>
      </box>
    </geometry>
    <surface>
      <friction>
        <ode>
          <mu>0.0</mu>
          <mu2>0.0</mu2>
        </ode>
      </friction>
    </surface>
  </collision>
```

```
  <visual name="Base3_visual">
    <material>
      <ambient>1 0 0 1</ambient>
      <diffuse>1 0 0 1</diffuse>
    </material>
    <geometry>
       <box>
        <size>0.2 0.05 0.05</size>
      </box>
    </geometry>
  </visual>
</link>

<joint name="z" type="prismatic">
  <parent>Base3</parent>
  <child>Base4</child>
  <pose>-0.025 0 0 0 0 0</pose>
  <axis>
    <xyz>0 0 1</xyz>
    <dynamics>
      <damping>0.01</damping>
      <friction>0.01</friction>
    </dynamics>
    <limit>
      <upper>0.1</upper>
      <lower>0.1</lower>
      <effort>1</effort>
    </limit>
  </axis>
</joint>

<link name="Base2">
  <pose>0 0 0.525 0 0 0</pose>
  <inertial>
    <mass>12.5</mass>
    <inertia>
      <ixx>26.00520833333333334</ixx>
      <ixy>0</ixy>
      <ixz>0</ixz>
      <iyy>26.0442708333334</iyy>
      <iyz>0</iyz>
      <izz>26.0442708333334</izz>
    </inertia>
  </inertial>
  <collision name="Base2_collision">
    <geometry>
      <box>
        <size>.05 50 .05</size>
      </box>
    </geometry>
    <surface>
      <friction>
        <ode>
          <mu>0</mu>
          <mu2>0</mu2>
        </ode>
      </friction>
    </surface>
  </collision>
  <visual name="Base2_visual">
    <material>
      <ambient>1 0 0 1</ambient>
      <diffuse>1 0 0 1</diffuse>
    </material>
    <geometry>
      <box>
        <size>.05 50 .05</size>
      </box>
    </geometry>
  </visual>
</link>

<joint name="x" type="prismatic">
  <parent>Base2</parent>
```

```
      <child>Base3</child>
      <pose>0 0 -0.025 0 0 0</pose>
      <axis>
        <xyz>0 1 0</xyz>
        <dynamics>
          <damping>0.0</damping>
          <friction>0.0</friction>
        </dynamics>
        <limit>
          <upper>0</upper>
          <lower>0</lower>
          <effort>1</effort>
        </limit>
      </axis>
    </joint>

    <link name="Base1">
        <pose>0 0 0.25 0 0 0</pose>
      <inertial>
        <mass>1.25</mass>
        <inertia>
          <ixx>0.0263020833333333</ixx>
          <ixy>0</ixy>
          <ixz>0</ixz>
          <iyy>0.0520833333333333</iyy>
          <iyz>0</iyz>
          <izz>0.0263020833333333</izz>
        </inertia>
      </inertial>
      <collision name="Base1_collision">
        <geometry>
          <box>
            <size>.05 .05 .5</size>
          </box>
        </geometry>
      </collision>
      <visual name="Base1_visual">
        <material>
          <ambient>1 0 0 1</ambient>
          <diffuse>1 0 0 1</diffuse>
        </material>
        <geometry>
          <box>
            <size>.05 .05 .5</size>
          </box>
        </geometry>
      </visual>
    </link>

    <joint name="Base1-Base2" type="fixed">
      <parent>Base1</parent>
      <child>Base2</child>
      <pose>0 0 -0.025 0 0 0</pose>
    </joint>

    <joint type="fixed" name="baseworld">
      <parent>world</parent>
      <child>Base1</child>
      <pose>0 0 -0.25 0 0 0</pose>
    </joint>

    <plugin name="joint_state_publisher" filename="libgazebo_ros_joint_state_publisher.so">
      <jointName>hip, spring</jointName>
      <updateRate>1000.0</updateRate>
      <alwaysOn>true</alwaysOn>
    </plugin>

    <plugin name="commands" filename="libtreadmill_plugin.so"/>


  </model>
</sdf>
```

gazebo_sim.launch

```
<launch>

    <!-- these are the arguments you can pass this launch file, for example paused:=true -->
  <arg name="paused" default="true"/>
  <arg name="use_sim_time" default="true"/>
  <arg name="gui" default="true"/>
  <arg name="headless" default="false"/>
  <arg name="debug" default="false"/>

  <!-- We resume the logic in empty_world.launch, changing only the name of the world to be launched -
->
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="$(find monopod_exp_final)/worlds/simulation.world"/>
    <arg name="debug" value="$(arg debug)" />
    <arg name="gui" value="$(arg gui)" />
    <arg name="paused" value="$(arg paused)"/>
    <arg name="use_sim_time" value="$(arg use_sim_time)"/>
    <arg name="headless" value="$(arg headless)"/>
  </include>


    <node name="position_controller" pkg="pid" type="controller" output="screen" >
      <param name="node_name" value="position_controller" />
      <param name="Kp" value="0.3" />
      <param name="Ki" value="0.0" />
      <param name="Kd" value="0.2" />
      <param name="upper_limit" value="28" />
      <param name="lower_limit" value="-28" />
      <param name="windup_limit" value="10" />
      <param name="diagnostic_period" value="0.25" />
      <param name="max_loop_frequency" value="100.0" />
      <param name="min_loop_frequency" value="100.0" />
    </node>

    <node name="Master" pkg="legged_robot" type="High_level_controller" output="screen" />

    <node name="Command_Interface" pkg="legged_robot" type="Gazebo_Actuation_Interface"
output="screen" />

    <node name="State_Callback_Interface" pkg="legged_robot" type="Gazebo_Sensors_Interface"
output="screen" />

    <node name="rqt_plot" pkg="rqt_plot" type="rqt_plot"
    args="/state/data /setpoint/data" />

</launch>
```

rosbag.m

```
    clear all ;
    bagfile = 'mybagfile.bag';
    bag = rosbag(bagfile);
    % Dislpay Available Topics
    bag.AvailableTopics
    % Select each Topic
    bagselect1 = select(bag, 'Time',  [bag.StartTime bag.EndTime], 'Topic', '/setpoint');
    bagselect2 = select(bag, 'Time',  [bag.StartTime bag.EndTime], 'Topic', '/state');
    bagselect3 = select(bag, 'Time',  [bag.StartTime bag.EndTime], 'Topic', '/control_effort');
%   bagselect4 = select(bag, 'Time',  [bag.StartTime bag.EndTime], 'Topic', '/botasys');
%   bagselect5 = select(bag, 'Time',  [bag.StartTime bag.EndTime], 'Topic', '/filtered_botasys');
    bagselect6 = select(bag, 'Time',  [bag.StartTime bag.EndTime], 'Topic', '/velocity_estimation');
%   bagselect7 = select(bag, 'Time',  [bag.StartTime bag.EndTime], 'Topic', '/IMU_feedback');

    % Store total number of messages for each topic
    all_msgs1 = bagselect1.NumMessages;
    all_msgs2 = bagselect2.NumMessages;
    all_msgs3 = bagselect3.NumMessages;
%   all_msgs4 = bagselect4.NumMessages;
%   all_msgs5 = bagselect5.NumMessages;
    all_msgs6 = bagselect6.NumMessages;
%   all_msgs7 = bagselect7.NumMessages;
```

```
    % Read messages from bag
    msgs1(1:all_msgs1) = readMessages(bagselect1, 1:all_msgs1);
    msgs2(1:all_msgs2) = readMessages(bagselect2, 1:all_msgs2);
    msgs3(1:all_msgs3) = readMessages(bagselect3, 1:all_msgs3);
%   msgs4(1:all_msgs4) = readMessages(bagselect4, 1:all_msgs4);
%   msgs5(1:all_msgs5) = readMessages(bagselect5, 1:all_msgs5);
    msgs6(1:all_msgs6) = readMessages(bagselect6, 1:all_msgs6);
%   msgs7(1:all_msgs7) = readMessages(bagselect7, 1:all_msgs7);


    % Convert messages to data setpoint
    for i = 1:all_msgs1; setpoint(i)= msgs1{i}.Data; end
    % Convert messages to data state
    for i = 1:all_msgs2; angle(i)= msgs2{i}.Data; end
    % Convert messages to data control_effort
    for i = 1:all_msgs3; control_effort(i)= msgs3{i}.Data; end
    % Convert messages to data force sensor
%   for i = 1:all_msgs4; Force_x(i)= msgs4{i}.wrench.force.x; end
%   for i = 1:all_msgs4; Force_y(i)= msgs4{i}.wrench.force.y; end
%   for i = 1:all_msgs4; Force_z(i)= msgs4{i}.wrench.force.z; end
%   for i = 1:all_msgs4; Torque_x(i)= msgs4{i}.wrench.torque.x; end
%   for i = 1:all_msgs4; Torque_y(i)= msgs4{i}.wrench.torque.x; end
%   for i = 1:all_msgs4; Torque_z(i)= msgs4{i}.wrench.torque.x; end
%   Convert messages to data filtered force sensor
%   for i = 1:all_msgs5; Filtered_Force_x(i)= msgs5{i}.wrench.force.x; end
%   for i = 1:all_msgs5; Filtered_Force_y(i)= msgs5{i}.wrench.force.y; end
%   for i = 1:all_msgs5; Filtered_Force_z(i)= msgs5{i}.wrench.force.z; end
%   for i = 1:all_msgs5; Filtered_Torque_x(i)= msgs5{i}.wrench.torque.x; end
%   for i = 1:all_msgs5; Filtered_Torque_y(i)= msgs5{i}.wrench.torque.x; end
%   for i = 1:all_msgs5; Filtered_Torque_z(i)= msgs5{i}.wrench.torque.x; end
%   Convert messages to data compression
    for i = 1:all_msgs6; velocity_est(i)= msgs6{i}.Data; end
%   Convert messages to data IMU
%   for i = 1:all_msgs7; acc_x(i)= msgs7{i}.accX; end
%   for i = 1:all_msgs7; acc_y(i)= msgs7{i}.accY; end
%   for i = 1:all_msgs7; acc_z(i)= msgs7{i}.accZ; end
%   for i = 1:all_msgs7; gyro_x(i)= msgs7{i}.gyroX; end
%   for i = 1:all_msgs7; gyro_y(i)= msgs7{i}.gyroY; end
%   for i = 1:all_msgs7; gyro_z(i)= msgs7{i}.gyroZ; end


    Time_setpoint = bagselect1.MessageList(:,1).Time;
    Time_state = bagselect2.MessageList(:,1).Time;
    Time_control_effort = bagselect3.MessageList(:,1).Time;
%   Time_force = bagselect4.MessageList(:,1).Time;
%   Time_fil_force = bagselect5.MessageList(:,1).Time;
    Time_vel = bagselect6.MessageList(:,1).Time;
%   Time_imu = bagselect7.MessageList(:,1).Time;

    clear all_msgs1 all_msgs2 all_msgs3 all_msgs4 all_msgs5 all_msgs6 all_msgs7 ans bag bagfile i
msgs1 msgs2 msgs3 msgs4 msgs5 msgs6 msgs7   bagselect1 bagselect2 bagselect3 bagselect4 bagselect5
bagselect6 bagselect7  ;
    save('mybagfile');

    clear all
% end
```

Gazebo_sensors_interface.cpp

```
#include "arpa/inet.h"
#include "netinet/in.h"
#include "sys/types.h"
#include "sys/socket.h"
#include "unistd.h"
#include "string.h"
#include "stdlib.h"
#include "signal.h"
#include "unistd.h"
#include <math.h>
```

```
#include "fcntl.h"
#include <stdint.h>
#include <inttypes.h>
#include "stdio.h"
#include <iostream>
#include <string>
#include <sstream>
#include "ros/ros.h"
#include "legged_robot/JointState.h"
#include "std_msgs/Float64.h"

using namespace std;
// Global variables
float pi = 4.0*atan(1.0);
double gposition[90], gvelocity[90] ;
float position, compression, hip_velocity, spring_velocity ;

// Generic error function
void error(char *s)
{
    perror(s);
    exit(1);
}

// Callback function for reception of position anf spring compression value from gazebo joint_states
topic
void GazeboCallback(const legged_robot::JointState::ConstPtr& array1)
{
        int i = 0;
        for(std::vector<double>::const_iterator it = array1->position.begin(); it != array1-
>position.end(); ++it)
        {
                gposition[i] = *it;
                i++;
        }
  i = 0;
  for(std::vector<double>::const_iterator it = array1->velocity.begin(); it != array1->velocity.end();
++it)
  {
    gvelocity[i] = *it;
    i++;
  }
        compression = (float) gposition[2];
        position = (float) gposition[0];
    spring_velocity = (float) gvelocity[2];
    hip_velocity =  (float) gvelocity[0];
return;
}

// Main Function
int main(int argc, char **argv)
{

  // Initialize ROS node
  ros::init(argc, argv, "Gazebo_Actuation_Interface");
  ros::NodeHandle n;

  ros::Subscriber Gazebo_sub = n.subscribe("/Assem1/joint_states", 1000, GazeboCallback);

  ros::Publisher angle2_pub = n.advertise<std_msgs::Float64>("/angle", 1000);
  ros::Publisher spring_vel_pub = n.advertise<std_msgs::Float64>("/spring_velocity", 1000);
  ros::Publisher hip_vel_pub = n.advertise<std_msgs::Float64>("/hip_velocity", 1000);
  ros::Publisher compression_pub = n.advertise<std_msgs::Float64>("/compression", 1000);
  ros::Rate loop_rate(2000); // Control rate in Hz

  // Wait for ROS node to initialize
  while (!ros::ok());

  std_msgs::Float64 pos;
  std_msgs::Float64 comp;
  std_msgs::Float64 svel, hvel;


  while (ros::ok())
```

```
  {

        pos.data = position ;
    angle2_pub.publish(pos);

    svel.data = spring_velocity ;
    spring_vel_pub.publish(svel);

    hvel.data = hip_velocity ;
    hip_vel_pub.publish(hvel);

        comp.data = compression ;
        compression_pub.publish(comp)  ;

    ros::spinOnce();
    loop_rate.sleep();
  }
  return 0;
}
```

Gazebo_actuation_interface.cpp

```
#include "arpa/inet.h"
#include "netinet/in.h"
#include "sys/types.h"
#include "sys/socket.h"
#include "unistd.h"
#include "string.h"
#include "stdlib.h"
#include "signal.h"
#include "unistd.h"
#include <math.h>
#include "fcntl.h"
#include <stdint.h>
#include <inttypes.h>
#include "stdio.h"
#include <iostream>
#include <string>
#include <sstream>
#include "ros/ros.h"
#include "std_msgs/Float64.h"
#include "std_msgs/Float32.h"


using namespace std;
// Global variables
float pi = 4.0*atan(1.0);
float command ;

// Generic error function
void error(char *s)
{
    perror(s);
    exit(1);
}

// Callback function for reception of position anf spring compression value from gazebo joint_states
topic
void CommandCallback(const std_msgs::Float64& com)
{
        command = com.data*4.8/28;
return;
}

// Main Function
int main(int argc, char **argv)
{

  // Initialize ROS node
  ros::init(argc, argv, "Gazebo_Actuation_Interface");
  ros::NodeHandle n;

  ros::Subscriber command_sub = n.subscribe("/control_effort", 1000, CommandCallback);
```

```
  ros::Publisher command_pub = n.advertise<std_msgs::Float64>("/Assem1/hip_torque_cmd", 1000);

  ros::Rate loop_rate(2000); // Control rate in Hz

  // Wait for ROS node to initialize
  while (!ros::ok());

  std_msgs::Float64 new_command;

  while (ros::ok())
  {
        new_command.data = command;
        command_pub.publish(new_command);

    ros::spinOnce();
    loop_rate.sleep();
  }
  return 0;
}
```

ros_speed.cpp

```
#include "arpa/inet.h"
#include "netinet/in.h"
#include "stdio.h"
#include "sys/types.h"
#include "sys/socket.h"
#include "unistd.h"
#include "string.h"
#include "stdlib.h"
#include "signal.h"
#include "unistd.h"
#include "fcntl.h"
#include <stdint.h>
#include <inttypes.h>
#include "stdio.h"
#include "string.h"
#include "stdlib.h"
#include <inttypes.h>
#include "ros/ros.h"
#include <sstream>
#include <iostream>
#include <string>
#include "std_msgs/Float64.h"

// UDP buffer length
#define BUFLEN 512
// UDP port to receive from
#define PORT 2012
// Asynchronous UDP communication
#define ASYNC
// UDP port to send data to
#define PORT_BRD 2011
// Tiva Back Left Leg board IP
#define BRD_IP "192.168.1.82"

using namespace std;
// Global variables
bool gotMsg = false; // Flag set high when message is received from UDP
int sock; // The socket identifier for UDP Rx communication
uint32_t encoderPos = 0;  // Place the received encoder value here
int msgs = 0; // Incoming message counter
struct sockaddr_in si_pwm;  // Struct for UDP send data socket
ssize_t SendPWMBytes = 2;  // Number of bytes to send for PWM command
char SendBuffer[6];  // UDP Send Buffer
int broad;       // The socket identifier for UDP Tx communication
int slen=sizeof(si_pwm);  // Size of sockaddr_in strut

// Generic error function
void error(char *s)
{
    perror(s);
```

```
    exit(1);
}

// Signal handler for asynchronous UDP
void sigio_handler(int sig)
{
  char buffer[BUFLEN]="";
  unsigned char val[4];
  struct sockaddr_in si_other;
  unsigned int slen=sizeof(si_other);
  ssize_t rcvbytes = 0;
  // Receive available bytes from UDP socket
  if ((rcvbytes = recvfrom(sock, &buffer, BUFLEN, 0, (struct sockaddr *)&si_other, &slen))==-1)
    error("recvfrom()");
  else
  {
    // Parse data , 1 int32 value
    if(buffer[0] == 0x42)
    {
      //ROS_INFO(" received");
      val[3] = (unsigned char)buffer[4];
      val[2] = (unsigned char)buffer[3];
      val[1] = (unsigned char)buffer[2];
      val[0] = (unsigned char)buffer[1];
      memcpy(&encoderPos, &val, 4);
      // Raise flag that we received a message
      gotMsg = true;
    }
  }
}

// Function to enable asynchronous UDP communication
int enable_asynch(int sock)
{
  int stat = -1;
  int flags;
  struct sigaction sa;
  flags = fcntl(sock, F_GETFL);
  fcntl(sock, F_SETFL, flags | O_ASYNC);
  sa.sa_flags = 0;
  sa.sa_handler = sigio_handler;
  sigemptyset(&sa.sa_mask);

  if (sigaction(SIGIO, &sa, NULL))
    error("Error:");

  if (fcntl(sock, F_SETOWN, getpid()) < 0)
    error("Error:");

  if (fcntl(sock, F_SETSIG, SIGIO) < 0)
    error("Error:");
  return 0;
}

// Callback function for reception of PWM message from topic
void freqCallback(const std_msgs::Float64::ConstPtr& msg)
{
        // Extract the duty cycle value and send it to the Tiva board via UDP
        SendBuffer[1] = (int8_t) msg->data;
        if (sendto(broad, SendBuffer, SendPWMBytes, 0, (struct sockaddr *)&si_pwm, slen)==-1)
                error("sendto()");
        // Print-out for debugging
}

// Main Function
int main(int argc, char **argv)
{
  struct sockaddr_in si_me, si_other;
  int i, slen=sizeof(si_other), msg_count;
  char buf[BUFLEN], strout[28];
  string inputS;

  msg_count = 0;
  memset(SendBuffer, 0, 6);
```

```
  // Initialize UDP socket for data transmission
  if ((broad=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP))==-1)
    error("socket");

  memset((char *) &si_pwm, 0, sizeof(si_pwm));
  si_pwm.sin_family = AF_INET;
  si_pwm.sin_port = htons(PORT_BRD);

  if (inet_aton(BRD_IP, &si_pwm.sin_addr)==0) {
    error("inet_aton() failed\n");
    exit(1);
  }

  SendBuffer[0] = 0x31;

  // Initialize ROS node
  ros::init(argc, argv, "ros_speed");
  ros::NodeHandle n;


  // Initialize the publisher for Encoder data post
  ros::Publisher motor_interface_pub = n.advertise<std_msgs::Float64>("state", 1000);
  // Initialize the subscriber for PWM data reception
  ros::Subscriber motor_pid_sub = n.subscribe("control_effort", 1000, freqCallback);

  ros::Rate loop_rate(10000);

  // Wait for ROS node to initialize
  while (!ros::ok());

  // Initialize UDP socket for data reception
  if ((sock=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP))==-1)
    error("socket");

  memset((char *) &si_me, 0, sizeof(si_me));
  si_me.sin_family = AF_INET;
  si_me.sin_port = htons(PORT);
  si_me.sin_addr.s_addr = htonl(INADDR_ANY);
  if (bind(sock, (struct sockaddr *)&si_me, sizeof(si_me))==-1)
    error("bind");

  enable_asynch(sock);

  ROS_INFO("Starting communication with TiVa board.");
  ROS_INFO("Communication with TiVa board established.");

  std_msgs::Float64 encoder_msg;
  encoder_msg.data = 0.0;

  while (ros::ok())
  {
    // If we got a new message, publish to topic and print values every 100 messages
    if(gotMsg)
    {
      encoder_msg.data = encoderPos*60.0*120000000.0/(8000000.0*2000.0)*(3.14/30.0)*0.125; // no need
to add reduction, as we receive mesurements on the belt revolution axis
      motor_interface_pub.publish(encoder_msg);
      //ROS_INFO("I heard: [%f]", encoder_msg.data);
      gotMsg = false;
    }
    ros::spinOnce();
    loop_rate.sleep();
  }
  return 0;
}
```

ros_read_vel.cpp

```
#include "stdio.h"
#include "string.h"
#include "stdlib.h"
#include <inttypes.h>
```

```
#include "ros/ros.h"
#include <iostream>
#include <string>
#include <sstream>
#include "std_msgs/Float64.h"

using namespace std;

// Global variables
std_msgs::Float64 position_msg;

int main(int argc, char **argv)
{
  float rpos = 0.0;
  string inputS;

  // Initialize ROS node
  ros::init(argc, argv, "ros_read_vel");
  ros::NodeHandle n;
  // Publish for desired position message
  ros::Publisher read_position_pub = n.advertise<std_msgs::Float64>("setpoint", 1000);
  ros::Rate loop_rate(1);
  position_msg.data = 0.0;
  ROS_INFO("Reading Desired Velocity.");
  while (ros::ok())
  {
        // Read a line from standard input and parse the desired position
        getline (cin,inputS);
        if (inputS == "q")
        {
                rpos = 0.0;
        position_msg.data = rpos;
                read_position_pub.publish(position_msg);
        break;
    }
    else if (inputS == "kill")
    {
                position_msg.data = -1;
                read_position_pub.publish(position_msg);
    }
        else if (inputS == "enable")
        {
                position_msg.data = -2;
                read_position_pub.publish(position_msg);
        }
    else if (inputS == "ccw")
        {
        position_msg.data = -3;
                read_position_pub.publish(position_msg);
        }
        else if (inputS == "cw")
        {
                position_msg.data = -4;
                read_position_pub.publish(position_msg);
        }
        else
        {
                stringstream ss;
                ss<<inputS;
                ss>>rpos; //convert string into int and store it in "asInt"
                ss.str(""); //clear the stringstream
                ss.clear();

                if ((rpos != 0.0) || (rpos == 0.0 && inputS == "0"))
                {
                        //cout << "Read : " << rpos << endl;
                        // Publish the received desired position
                        position_msg.data = rpos;
                        read_position_pub.publish(position_msg);
                }
        }
        ros::spinOnce();
    loop_rate.sleep();
  }
```

```
  return 0;
}
```

Appendix B

This appendix contains information and datasheets of the various of-the-shelf components that were utilized for the purposes of this thesis.

SONGLE RELAY

松乐继电器 ® SONGLE RELAY	RELAY ISO9002	SRD

1. MAIN FEATURES

- Switching capacity available by 10A in spite of small size design for highdensity P.C. board mounting technique.
- UL,CUL,TUV recognized.
- Selection of plastic material for high temperature and better chemical solution performance.
- Sealed types available.
- Simple relay magnetic circuit to meet low cost of mass production.

2. APPLICATIONS

- Domestic appliance, office machine, audio, equipment, automobile, etc.

  (Remote control TV receiver, monitor display, audio equipment high rushing current use application.)

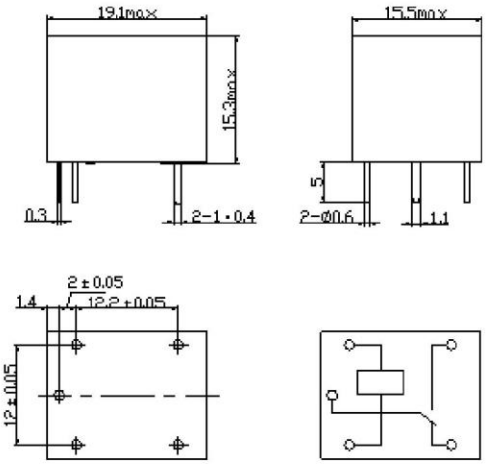
3. ORDERING INFORMATION

SRD	XX VDC	S	L	C
Model of relay	Nominal coil voltage	Structure	Coil sensitivity	Contact form
SRD	03、05、06、09、12、24、48VDC	S:Sealed type	L:0.36W	A:1 form A
				B:1 form B
		F:Flux free type	D:0.45W	C:1 form C

4. RATING

CCC	FILE NUMBER:CH0052885-2000	7A/240VDC
CCC	FILE NUMBER:CH0036746-99	10A/250VDC
UL /CUL	FILE NUMBER: E167996	10A/125VAC 28VDC
TUV	FILE NUMBER: R9933789	10A/240VAC 28VDC

5. DIMENSION(unit:mm) DRILLING(unit:mm) WIRING DIAGRAM



6. COIL DATA CHART (AT20°C)

Coil Sensitivity	Coil Voltage Code	Nominal Voltage (VDC)	Nominal Current (mA)	Coil Resistance (Ω) ±10%	Power Consumption (W)	Pull-In Voltage (VDC)	Drop-Out Voltage (VDC)	Max-Allowable Voltage (VDC)
SRD (High Sensitivity)	03	03	120	25	abt. 0.36W	75%Max.	10% Min.	120%
	05	05	71.4	70				
	06	06	60	100				
	09	09	40	225				
	12	12	30	400				
	24	24	15	1600				
	48	48	7.5	6400				
SRD (Standard)	03	03	150	20	abt. 0.45W	75% Max.	10% Min.	110%
	05	05	89.3	55				
	06	06	75	80				
	09	09	50	180				
	12	12	37.5	320				
	24	24	18.7	1280				
	48	48	10	4500	abt. 0.51W			

7. CONTACT RATING

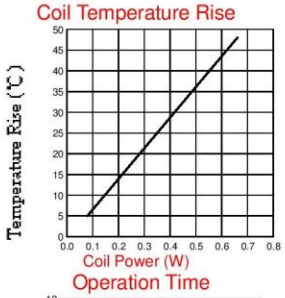
Item \ Type	SRD	
	FORM C	FORM A
Contact Capacity Resistive Load (cosΦ=1)	7A 28VDC 10A 125VAC 7A 240VAC	10A 28VDC 10A 240VAC
Inductive Load (cosΦ=0.4 L/R=7msec)	3A 120VAC 3A 28VDC	5A 120VAC 5A 28VDC
Max. Allowable Voltage	250VAC/110VDC	250VAC/110VDC
Max. Allowable Power Force	800VAC/240W	1200VA/300W
Contact Material	AgCdO	AgCdO

8. PERFORMANCE (at initial value)

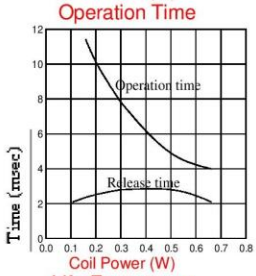
Item \ Type	SRD
Contact Resistance	100mΩ Max.
Operation Time	10msec Max.
Release Time	5msec Max.
Dielectric Strength Between coil & contact Between contacts	 1500VAC 50/60HZ (1 minute) 1000VAC 50/60HZ (1 minute)
Insulation Resistance	100 MΩ Min. (500VDC)
Max. ON/OFF Switching Mechanically Electrically	 300 operation/min 30 operation/min
Ambient Temperature	-25°C to +70°C
Operating Humidity	45 to 85% RH
Vibration Endurance Error Operation	 10 to 55Hz Double Amplitude 1.5mm 10 to 55Hz Double Amplitude 1.5mm
Shock Endurance Error Operation	 100G Min. 10G Min.
Life Expectancy Mechanically Electrically	 10^7 operations. Min. (no load) 10^5 operations. Min. (at rated coil voltage)
Weight	abt. 10grs.

9.REFERENCE DATA

Coil Temperature Rise

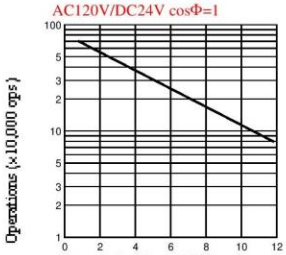


Operation Time

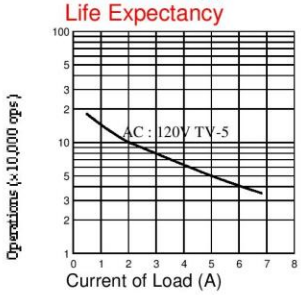


Life Expectancy

AC120V/DC24V cosΦ=1



Life Expectancy



Pololu 5V, 6A Step-Down Voltage Regulator D24V60F5

Pololu item #: 2865

Dimensions

Size:	1.6″ × 0.8″ × 0.3″[1]
Weight:	4.8 g[1]

This synchronous switching step-down (or buck) regulator takes an input voltage of up to 38 V and efficiently reduces it to 5 V with an available output current of around **6 A**. Typical efficiencies of 80% to 95% make this regulator well suited for higher-power applications like powering motors or servos, while high efficiencies are maintained at light loads by dynamically changing the switching frequency, and an optional shutdown pin enables a low-power state with a current draw of a few hundred microamps. The regulator's output voltage setting can also be lowered by adding an external resistor.

General specifications

Minimum operating voltage:	5 V
Maximum operating voltage:	38 V
Continuous output current:	6 A[2]
Output voltage:	5 V
Reverse voltage protection?:	Y
Maximum quiescent current:	15 mA[3]

Notes:

1. Without included hardware.
2. Typical. Actual continuous output current limited by thermal dissipation.
3. Typical worst case (i.e. with VIN close to 5V). Quiescent current depends on the input and output voltages and is much lower for most of the input voltage range (typically below 1 mA). The ENABLE pin can be used to reduce the quiescent current to a few hundred microamps.

Order this document by LM358/D

MOTOROLA

LM358, LM258, LM2904, LM2904V

Dual Low Power Operational Amplifiers

DUAL DIFFERENTIAL INPUT OPERATIONAL AMPLIFIERS

SEMICONDUCTOR TECHNICAL DATA

Utilizing the circuit designs perfected for recently introduced Quad Operational Amplifiers, these dual operational amplifiers feature 1) low power drain, 2) a common mode input voltage range extending to ground/V_{EE}, 3) single supply or split supply operation and 4) pinouts compatible with the popular MC1558 dual operational amplifier. The LM158 series is equivalent to one–half of an LM124.

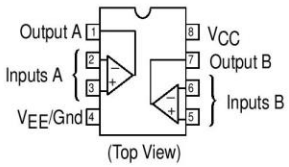
These amplifiers have several distinct advantages over standard operational amplifier types in single supply applications. They can operate at supply voltages as low as 3.0 V or as high as 32 V, with quiescent currents about one–fifth of those associated with the MC1741 (on a per amplifier basis). The common mode input range includes the negative supply, thereby eliminating the necessity for external biasing components in many applications. The output voltage range also includes the negative power supply voltage.

- Short Circuit Protected Outputs
- True Differential Input Stage
- Single Supply Operation: 3.0 V to 32 V
- Low Input Bias Currents
- Internally Compensated
- Common Mode Range Extends to Negative Supply
- Single and Split Supply Operation
- Similar Performance to the Popular MC1558
- ESD Clamps on the Inputs Increase Ruggedness of the Device without Affecting Operation

N SUFFIX
PLASTIC PACKAGE
CASE 626

D SUFFIX
PLASTIC PACKAGE
CASE 751
(SO–8)

PIN CONNECTIONS



(Top View)

MAXIMUM RATINGS (T_A = +25°C, unless otherwise noted.)

Rating	Symbol	LM258 LM358	LM2904 LM2904V	Unit
Power Supply Voltages Single Supply Split Supplies	 V_{CC} V_{CC}, V_{EE}	 32 ±16	 26 ±13	Vdc
Input Differential Voltage Range (Note 1)	V_{IDR}	±32	±26	Vdc
Input Common Mode Voltage Range (Note 2)	V_{ICR}	–0.3 to 32	–0.3 to 26	Vdc
Output Short Circuit Duration	t_{SC}	Continuous		
Junction Temperature	T_J	150		°C
Storage Temperature Range	T_{stg}	–55 to +125		°C
Operating Ambient Temperature Range LM258 LM358 LM2904 LM2904V	T_A	 –25 to +85 0 to +70 – –	 – – –40 to +105 –40 to +125	°C

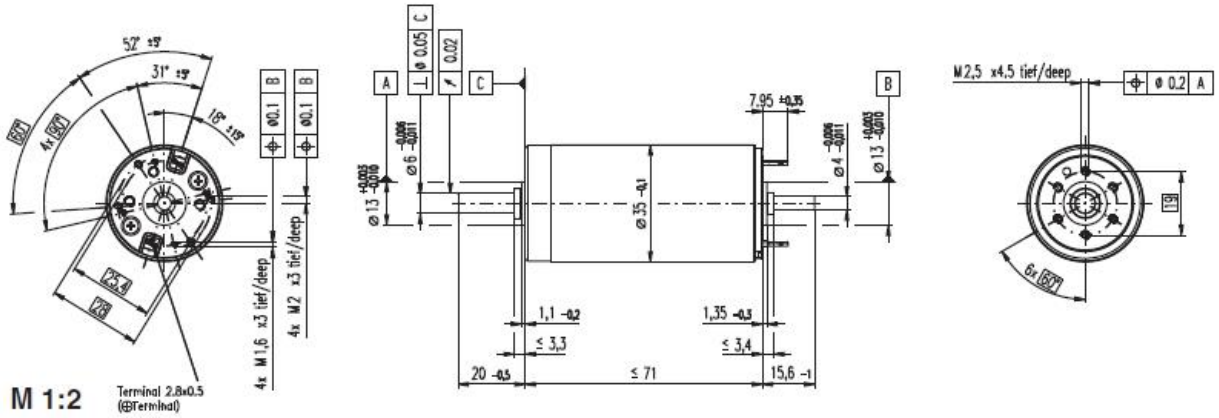
NOTES: 1. Split Power Supplies.
2. For Supply Voltages less than 32 V for the LM258/358 and 26 V for the LM2904, the absolute maximum input voltage is equal to the supply voltage.

ORDERING INFORMATION

Device	Operating Temperature Range	Package
LM2904D	T_A = –40° to +105°C	SO–8
LM2904N		Plastic DIP
LM2904VD	T_A = –40° to +125°C	SO–8
LM2904VN		Plastic DIP
LM258D	T_A = –25° to +85°C	SO–8
LM258N		Plastic DIP
LM358D	T_A = 0° to +70°C	SO–8
LM358N		Plastic DIP

 Rev 2

RE 35 ∅35 mm, Graphite Brushes, 90 Watt



Stock program
Standard program
Special program (on request)

Order Number														
according to dimensional drawing	273752	323890	273753	273754	273755	273756	273757	273758	273759	273760	273761	273762	273763	
shaft length 15.6 shortened to 4 mm	285785	323891	285786	285787	285788	285789	285790	285791	285792	285793	285794	285795	285796	
Motor Data														
	Values at nominal voltage													
1	Nominal voltage	V	15.0	24.0	30.0	42.0	48.0	48.0	48.0	48.0	48.0	48.0	48.0	48.0
2	No load speed	rpm	7070	7670	7220	7530	7270	6650	5960	4740	3810	3140	2570	2100	1620
3	No load current	mA	245	168	123	92.7	77.3	68.7	59.7	44.7	34.2	27.1	21.6	17.2	12.9
4	Nominal speed	rpm	6270	6910	6420	6770	6490	5860	5150	3920	2970	2280	1710	1220	732
5	Nominal torque (max. continuous torque)	mNm	73.2	93.3	92.4	97.7	96.5	98.2	98.8	102	105	105	105	104	104
6	Nominal current (max. continuous current)	A	4.00	3.36	2.50	1.95	1.63	1.51	1.36	1.12	0.915	0.752	0.621	0.503	0.391
7	Stall torque	mNm	874	1160	949	1070	967	878	766	613	493	394	320	253	194
8	Starting current	A	45.0	39.7	24.4	20.3	15.5	12.9	10.1	6.43	4.16	2.74	1.83	1.18	0.704
9	Max. efficiency	%	81	84	84	86	85	85	84	83	82	80	79	77	74
	Characteristics														
10	Terminal resistance	Ω	0.334	0.605	1.23	2.07	3.09	3.72	4.75	7.46	11.5	17.5	26.2	40.5	68.2
11	Terminal inductance	mH	0.085	0.191	0.340	0.620	0.870	1.04	1.29	2.04	3.16	4.65	6.89	10.3	17.1
12	Torque constant	mNm / A	19.4	29.2	38.9	52.5	62.2	68	75.8	95.2	119	144	175	214	276
13	Speed constant	rpm / V	491	328	246	182	154	140	126	100	80.5	66.4	54.6	44.7	34.6
14	Speed / torque gradient	rpm / mNm	8.43	6.79	7.76	7.16	7.62	7.67	7.89	7.85	7.84	8.08	8.19	8.46	8.55
15	Mechanical time constant	ms	5.97	5.60	5.50	5.40	5.38	5.38	5.39	5.38	5.37	5.38	5.39	5.39	5.41
16	Rotor inertia	gcm²	67.6	78.7	67.6	72.0	67.4	67.0	65.2	65.4	65.5	63.6	62.8	60.8	60.4

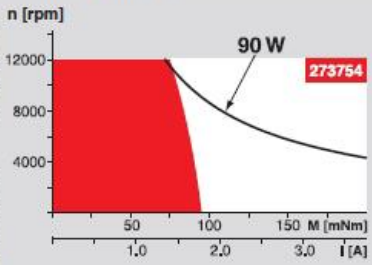
Specifications

	Thermal data	
17	Thermal resistance housing-ambient	6.2 K / W
18	Thermal resistance winding-housing	2.0 K / W
19	Thermal time constant winding	30 s
20	Thermal time constant motor	1050 s
21	Ambient temperature	-30 ... +100°C
22	Max. permissible winding temperature	+155°C
	Mechanical data (ball bearings)	
23	Max. permissible speed	12000 rpm
24	Axial play	0.05 - 0.15 mm
25	Radial play	0.025 mm
26	Max. axial load (dynamic)	5.6 N
27	Max. force for press fits (static)	110 N
	(static, shaft supported)	1200 N
28	Max. radial loading, 5 mm from flange	28 N
	Other specifications	
29	Number of pole pairs	1
30	Number of commutator segments	13
31	Weight of motor	340 g

Values listed in the table are nominal.
Explanation of the figures on page 49.

Option
Hollow shaft as special design
Preloaded ball bearings

Operating Range



Comments

Continuous operation
In observation of above listed thermal resistance (lines 17 and 18) the maximum permissible winding temperature will be reached during continuous operation at 25°C ambient.
= Thermal limit.

Short term operation
The motor may be briefly overloaded (recurring).

Assigned power rating

maxon Modular System

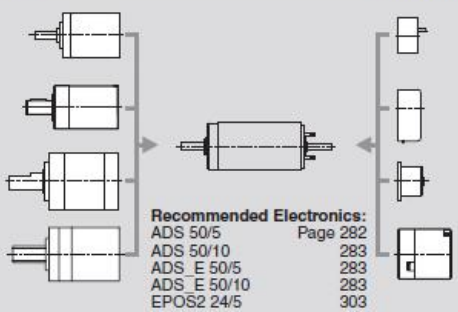
Overview on page 16 - 21

Planetary Gearhead
∅32 mm
0.75 - 6.0 Nm
Page 232 / 234 / 235

Planetary Gearhead
∅32 mm
4.0 - 8.0 Nm
Page 237

Planetary Gearhead
∅42 mm
3 - 15 Nm
Page 240

Spindle Drive
∅32 mm
Page 251 / 252 / 253



Recommended Electronics:

ADS 50/5	Page 282
ADS 50/10	283
ADS_E 50/5	283
ADS_E 50/10	283
EPOS2 24/5	303
EPOS2 50/5	303
EPOS P 24/5	306
Notes	18

Encoder MR
256 - 1024 CPT,
3 channels
Page 265

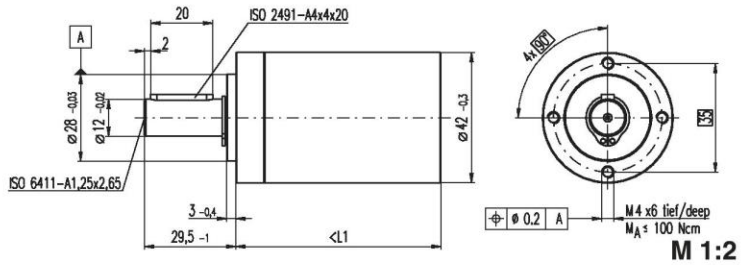
Encoder HED_ 5540
500 CPT,
3 channels
Page 268 / 270

DC-Tacho DCT
∅22 mm
0.52 V
Page 277

Brake AB 28
∅40 mm
24 VDC, 0.4 Nm
Page 316

Planetary Gearhead GP 42 C ∅42 mm, 3 - 15 Nm

Ceramic Version

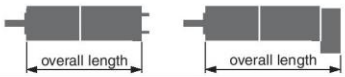


Technical Data

Planetary Gearhead				straight teeth
Output shaft				stainless steel
Bearing at output				preloaded ball bearings
Radial play, 12 mm from flange				max. 0.06 mm
Axial play at axial load	< 5 N			0 mm
	> 5 N			max. 0.3 mm
Max. permissible axial load				150 N
Max. permissible force for press fits				300 N
Sense of rotation, drive to output				=
Recommended input speed				< 8000 rpm
Recommended temperature range				-20 ... +100°C
Extended area as option				-35 ... +100°C
Number of stages	1	2	3	4
Max. radial load, 12 mm from flange	120 N	150 N	150 N	150 N

Stock program
Standard program
Special program (on request)

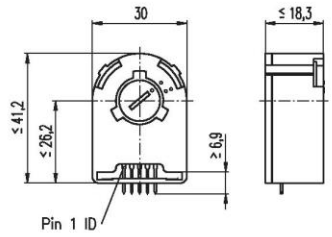
			Order Number									
			203113	203115	203119	203120	203124	203129	203128	203133	203137	203141
Gearhead Data												
1	Reduction		3.5 : 1	12 : 1	26 : 1	43 : 1	81 : 1	156 : 1	150 : 1	285 : 1	441 : 1	756 : 1
2	Reduction absolute		$^{7}/_{2}$	$^{49}/_{4}$	26	$^{343}/_{8}$	$^{2197}/_{27}$	156	$^{2401}/_{16}$	$^{15379}/_{54}$	441	756
3	Mass inertia	gcm²	14	15	9.1	15	9.4	9.1	15	15	14	14
4	Max. motor shaft diameter	mm	10	10	8	10	8	8	10	10	10	10
	Order Number		203114	203116		203121	203125		203130	203134	203138	203142
1	Reduction		4.3 : 1	15 : 1		53 : 1	91 : 1		186 : 1	319 : 1	488 : 1	936 : 1
2	Reduction absolute		$^{13}/_{3}$	$^{91}/_{6}$		$^{637}/_{12}$	91		$^{4459}/_{24}$	$^{637}/_{2}$	$^{4394}/_{9}$	936
3	Mass inertia	gcm²	9.1	15		15	15		15	15	9.4	9.1
4	Max. motor shaft diameter	mm	8	10		10	10		10	10	8	8
	Order Number			203117		203122	203126		203131	203135	203139	
1	Reduction			19 : 1		66 : 1	113 : 1		230 : 1	353 :1	546 : 1	
2	Reduction absolute			$^{169}/_{9}$		$^{1183}/_{18}$	$^{338}/_{3}$		$^{8281}/_{36}$	$^{28561}/_{81}$	546	
3	Mass inertia	gcm²		9.4		15	9.4		15	9.4	14	
4	Max. motor shaft diameter	mm		8		10	8		10	8	10	
	Order Number			203118		203123	203127		203132	203136	203140	
1	Reduction			21 : 1		74 : 1	126 : 1		257 : 1	394 : 1	676 : 1	
2	Reduction absolute			21		$^{147}/_{2}$	126		$^{1029}/_{4}$	$^{1183}/_{3}$	676	
3	Mass inertia	gcm²		14		15	14		15	15	9.1	
4	Max. motor shaft diameter	mm		10		10	10		10	10	8	
5	Number of stages		1	2	2	3	3	3	4	4	4	4
6	Max. continuous torque	Nm	3.0	7.5	7.5	15.0	15.0	15.0	15.0	15.0	15.0	15.0
7	Intermittently permissible torque at gear output	Nm	4.5	11.3	11.3	22.5	22.5	22.5	22.5	22.5	22.5	22.5
8	Max. efficiency	%	90	81	81	72	72	72	64	64	64	64
9	Weight	g	260	360	360	460	460	460	560	560	560	560
10	Average backlash no load	°	0.3	0.4	0.4	0.5	0.5	0.5	0.5	0.5	0.5	0.5
11	Gearhead length L1	mm	40.9	55.4	55.4	69.9	69.9	69.9	84.4	84.4	84.4	84.4

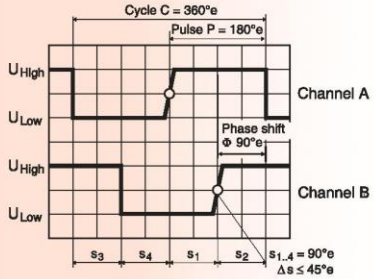


maxon Modular System

+ Motor	Page	+ Sensor	Page	+ Brake	Page	**Overall length [mm]** = Motor length + gearhead length + (sensor / brake) + assembly parts									
EC 45, 250 W	159					185.0	199.5	199.5	214.0	214.0	214.0	228.5	228.5	228.5	228.5
EC 45, 250 W	159	HEDL 9140	273			200.6	215.1	215.1	229.6	229.6	229.6	244.1	244.1	244.1	244.1
EC 45, 250 W	159	Res 26	278			185.0	199.5	199.5	214.0	214.0	214.0	228.5	228.5	228.5	228.5
EC 45, 250 W	159			AB 28	317	192.4	206.9	206.9	221.4	221.4	221.4	235.9	235.9	235.9	235.9
EC 45, 250 W	159	HEDL 9140	273	AB 28	317	209.4	223.9	223.9	238.4	238.4	238.4	252.9	252.9	252.9	252.9
EC-max 30, 60 W	171					105.0	119.5	119.5	134.0	134.0	134.0	148.5	148.5	148.5	148.5
EC-max 30, 60 W	171	MR	264			117.2	131.7	131.7	146.2	146.2	146.2	160.7	160.7	160.7	160.7
EC-max 30, 60 W	171	HEDL 5540	271			125.6	140.1	140.1	154.6	154.6	154.6	169.1	169.1	169.1	169.1
EC-max 30, 60 W	171			AB 20	314	140.6	155.1	155.1	169.6	169.6	169.6	184.1	184.1	184.1	184.1
EC-max 30, 60 W	171	HEDL 5540	271	AB 20	314	164.6	179.1	179.1	193.6	193.6	193.6	208.1	208.1	208.1	208.1
EC-max 40, 70 W	172					99.0	113.5	113.5	128.0	128.0	128.0	142.5	142.5	142.5	142.5
EC-max 40, 70 W	172	MR	265			114.9	129.4	129.4	143.9	143.9	143.9	158.4	158.4	158.4	158.4
EC-max 40, 70 W	172	HEDL 5540	271			122.4	136.9	136.9	151.4	151.4	151.4	165.9	165.9	165.9	165.9
EC-max 40, 70 W	172			AB 28	315	139.0	153.5	153.5	168.0	168.0	168.0	182.5	182.5	182.5	182.5
EC-max 40, 70 W	173	HEDL 5540	271	AB 28	315	162.4	176.9	176.9	191.4	191.4	191.4	205.9	205.9	205.9	205.9
EC-power 30, 100 W	179					88.0	102.5	102.5	117.0	117.0	117.0	131.5	131.5	131.5	131.5
EC-power 30, 100 W	179	MR	264			100.2	114.7	114.7	129.2	129.2	129.2	143.7	143.7	143.7	143.7
EC-power 30, 100 W	179	HEDL 5540	272			108.6	123.1	123.1	137.6	137.6	137.6	152.1	152.1	152.1	152.1
EC-power 30, 100 W	179			AB 20	314	124.2	138.7	138.7	153.2	153.2	153.2	167.7	167.7	167.7	167.7
EC-power 30, 100 W	179	HEDL 5540	272	AB 20	314	145.0	159.5	159.5	174.0	174.0	174.0	188.5	188.5	188.5	188.5
EC-power 30, 200 W	180					105.0	119.5	119.5	134.0	134.0	134.0	148.5	148.5	148.5	148.5
EC-power 30, 200 W	180	MR	264			117.2	131.7	131.7	146.2	146.2	146.2	160.7	160.7	160.7	160.7
EC-power 30, 200 W	180	HEDL 5540	272			125.6	140.1	140.1	154.6	154.6	154.6	169.1	169.1	169.1	169.1
EC-power 30, 200 W	180			AB 20	314	141.2	155.7	155.7	170.2	170.2	170.2	184.7	184.7	184.7	184.7
EC-power 30, 200 W	180	HEDL 5540	272	AB 20	314	162.0	176.5	176.5	191.0	191.0	191.0	205.5	205.5	205.5	205.5
MCD EPOS, 60 W	311					161.0	175.5	175.5	190.0	190.0	190.0	204.5	204.5	204.5	204.5
MCD EPOS P, 60 W	311					161.0	175.5	175.5	190.0	190.0	190.0	204.5	204.5	204.5	204.5

Encoder HEDS 5540, 500 Counts per turn, 3 Channels





Stock program
Standard program
Special program (on request)

Order Number	110511	110513	110515
Type			
Counts per turn	500	500	500
Number of channels	3	3	3
Max. operating frequency (kHz)	100	100	100
Shaft diameter (mm)	3	4	6

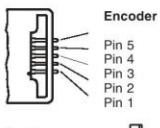
Combination

+ Motor	Page	+ Gearhead	Page	+ Brake	Page	Overall length [mm] / ● see: + Gearhead					
RE 25, 10 W	77					75.3					
RE 25, 10 W	77	GP 26, 0.5 - 2.0 Nm	235			●					
RE 25, 10 W	77	GP 32, 0.4 - 2.0 Nm	237			●					
RE 25, 10 W	77	GP 32, 0.75 - 6.0 Nm	238/240			●					
RE 25, 20 W	79					75.3					
RE 25, 20 W	79	GP 26, 0.5 - 2.0 Nm	235			●					
RE 25, 20 W	79	GP 32, 0.4 - 2.0 Nm	237			●					
RE 25, 20 W	79	GP 32, 0.75 - 6.0 Nm	238/240			●					
RE 25, 20 W	79			AB 28	308	105.7					
RE 25, 20 W	79	GP 26, 0.5 - 2.0 Nm	235	AB 28	308	●					
RE 25, 20 W	79	GP 32, 0.4 - 2.0 Nm	237	AB 28	308	●					
RE 25, 20 W	79	GP 32, 0.75 - 6.0 Nm	238/240	AB 28	308	●					
RE 26, 18 W	80					77.2					
RE 26, 18 W	80	GP 26, 0.5 - 2.0 Nm	235			●					
RE 26, 18 W	80	GP 32, 0.4 - 2.0 Nm	237			●					
RE 26, 18 W	80	GP 32, 0.75 - 6.0 Nm	238/240			●					
RE 35, 90 W	82						91.9				
RE 35, 90 W	82	GP 32, 0.75 - 6.0 Nm	239/240				●				
RE 35, 90 W	82	GP 32, 8 Nm	242				●				
RE 35, 90 W	82	GP 42, 3.0 - 15 Nm	244				●				
RE 35, 90 W	82			AB 28	308	124.1					
RE 35, 90 W	82	GP 32, 0.75 - 6.0 Nm	239/240	AB 28	308	●					
RE 35, 90 W	82	GP 42, 3.0 - 15 Nm	244	AB 28	308	●					
RE 36, 70 W	83						92.2				
RE 36, 70 W	83	GP 32, 0.4 - 2.0 Nm	237				●				
RE 36, 70 W	83	GP 32, 0.75 - 6.0 Nm	239/240				●				
RE 36, 70 W	83	GP 42, 3.0 - 15 Nm	244				●				
RE 40, 150 W	84						91.7				
RE 40, 150 W	84	GP 42, 3.0 - 15 Nm	244				●				
RE 40, 150 W	84	GP 52, 4.0 - 30 Nm	247				●				
RE 40, 150 W	84			AB 28	308	124.2					
RE 40, 150 W	84	GP 42, 3.0 - 15 Nm	244	AB 28	308	●					
RE 40, 150 W	84	GP 52, 4.0 - 30 Nm	247	AB 28	308	●					

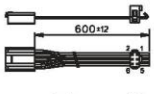
Technical Data

Supply voltage	5 V ± 10 %
Output signal	TTL compatible
Phase shift F (nominal)	90°e ± 45°e
Signal rise time (typical at C_L = 25 pF, R_L = 2.7 kΩ, 25°C)	180 ns
Signal fall time (typical at C_L = 25 pF, R_L = 2.7 kΩ, 25°C)	40 ns
Index pulse width (nominal)	90°e
Operating temperature range	-40 ... +100°C
Moment of inertia of code wheel	≤ 0.6 gcm²
Max. angular acceleration	250 000 rad s⁻²
Output current per channel	min. -1 mA, max. 5 mA

Pin Allocation



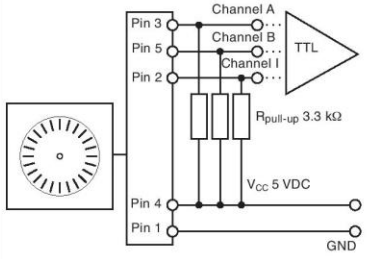
Encoder	Description	Pin no. from 3409.506
Pin 5	Channel B	1
Pin 4	V_{CC}	2
Pin 3	Channel A	3
Pin 2	Channel I	4
Pin 1	GND	5



Cable with plug:
maxon Art. No. 3409.506
The plug (Harting 09185066803) can be fixed in the required position.

Cable with plug: (compatible with Encoder HEDS5010)
maxon Art. No. 3409.504
The plug (3M 89110-0101) can be fixed in the required position.

Connection example



Ambient temperature range δ_U = 25°C

May 2008 edition / subject to change

Description

The AZBDC12A8 PWM servo drive is designed to drive brushless and brushed DC motors at a high switching frequency. To increase system reliability and to reduce cabling costs, the drive is designed for direct integration into your PCB. The AZBDC12A8 is fully protected against over-voltage, over-current, over-heating and short-circuits. A single digital output indicates operating status. The drive interfaces with digital controllers that have digital PWM output. The PWM IN duty cycle determines the output current and DIR input determines the direction of rotation. This servo drive requires only a single unregulated isolated DC power supply, and is fully RoHS (Reduction of Hazardous Substances) compliant.

See Part Numbering Information on last page of datasheet for additional ordering options.

Power Range

Peak Current	12 A
Continuous Current	6 A
Supply Voltage	20 - 80 VDC

Features

- Four Quadrant Regenerative Operation
- Direct Board-to-Board Integration
- Lightweight
- High Switching Frequency
- Wide Temperature Range
- High Performance Thermal Dissipation
- Differential Input Command
- Digital Fault Output Monitor
- Current Monitor Output
- Single Supply Operation
- Compact Size
- High Power Density

HARDWARE PROTECTION
- Over-Voltage
- Over-Current
- Over-Temperature
- Short-circuit (phase-phase)
- Short-circuit (phase-ground)

INPUTS/OUTPUTS
- Digital Fault Output
- Digital Inhibit Input
- Analog Current Monitor
- Analog Command Input
- Analog Current Reference

FEEDBACK SUPPORTED
- Hall Sensors

MODES OF OPERATION
- Current

COMMUTATION
- Trapezoidal

MOTORS SUPPORTED
- Three Phase (Brushless)
- Single Phase (Brushed, Voice Coil, Inductive Load)

COMMAND SOURCE
- PWM

COMPLIANCES & AGENCY APPROVALS
- UL
- cUL
- CE Class A (LVD)
- CE Class A (EMC)
- RoHS

Low Profile, Low Noise Six Degrees of Freedom Inertial Sensor

Data Sheet **ADIS16375**

FEATURES

- Triaxis digital gyroscope, ±300°/sec
  - Tight orthogonal alignment: 0.05°
- Triaxis digital accelerometer: ±18 *g*
- Delta-angle/velocity calculations
- Wide sensor bandwidth: 330 Hz
- High sample rate: 2.460 kSPS
- Autonomous operation and data collection
  - No external configuration commands required
  - Startup time: 500 ms
- Factory-calibrated sensitivity, bias, and axial alignment
  - Calibration temperature range: −40°C to +85°C
- SPI-compatible serial interface
- Embedded temperature sensor
- Programmable operation and control
  - Automatic and manual bias correction controls
  - 4 FIR filter banks, 120 configurable taps
  - Digital I/O: data-ready, alarm indicator, external clock
  - Alarms for condition monitoring
  - Power-down/sleep mode for power management
  - Enable external sample clock input: up to 2.25 kHz
  - Single-command self test
- Single-supply operation: 3.3 V
- 2000 *g* shock survivability
- Operating temperature range: −40°C to +105°C

APPLICATIONS

- Precision instrumentation
- Platform stabilization and control
- Industrial vehicle navigation
- Downhole instrumentation
- Robotics

FUNCTIONAL BLOCK DIAGRAM

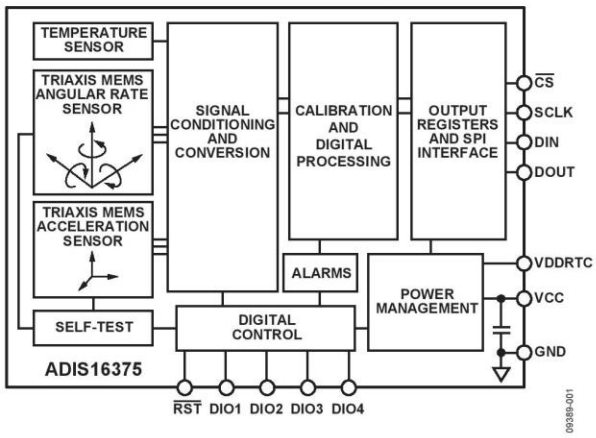


Figure 1.

GENERAL DESCRIPTION

The ADIS16375 *i*Sensor® is a complete inertial system that includes a triaxis gyroscope and triaxis accelerometer. Each sensor in the ADIS16375 combines industry-leading *i*MEMS® technology with signal conditioning that optimizes dynamic performance. The factory calibration characterizes each sensor for sensitivity, bias, alignment, and linear acceleration (gyro bias). As a result, each sensor has its own dynamic compensation formulas that provide accurate sensor measurements over a temperature range of −40°C to +105°C.

The ADIS16375 provides a simple, cost-effective method for integrating accurate, multiaxis, inertial sensing into industrial systems, especially when compared with the complexity and investment associated with discrete designs. All necessary motion testing and calibration are part of the production process at the factory, greatly reducing system integration time. Tight orthogonal alignment simplifies inertial frame alignment in navigation systems. An improved SPI interface and register structure provide faster data collection and configuration control.

This compact module is approximately 44 mm × 47 mm × 14 mm and provides a flexible connector interface that enables multiple mounting orientation options.

Rev. C

One Technology Way, P.O. Box 9106, Norwood, MA 02062-9106, U.S.A.
Tel: 781.329.4700 www.analog.com
Fax: 781.461.3113

BOTA "Rikudo" is a 6 D.O.F. (Degrees Of Freedom) Force/Torque sensor. Precision strain gauges are used to measure all force/torque components through the elastic properties of the sensing beams inside the sensor. Internal electronics carefully amplifies and filters the small amplitude signal of the strain gages to a measurable voltage range and drives it through the internal ADC. Further digital filtering is implemented. The final filtered signal can be, transferred to PC or other microcontroller through various methods. The supported communication protocols are CAN, Ethernet, RS232, RS485 and USB.

The sensor can be calibrated and configured to be used to different force ranges as shown at Table 1.

TABLE 1. RANGES AND RESOLUTIONS.

	SINGLE AXIS OVERLOAD	RANGE	RESOLUTION	PRECISION (PEAK TO PEAK NOISE)
$F_{x,y}$	±500N	±100 N , ±200 N	0.049 N , 0.098 N	0.16 N , 0.32 N
F_z	±1400N	±180 N , ±360 N	0.088 N , 0.176 N	0.30 N , 0.60 N
$T_{x,y}$	±12Nm	±3 Nm , ±6 Nm	0.0015 Nm , 0.003 Nm	0.005 Nm , 0.010 Nm
T_z	±15Nm	±5 Nm , ±10 Nm	0.0025 Nm , 0.005 Nm	0.008 Nm , 0.016 Nm

Other important indices are described at table 2.

TABLE 2. BASIC CHARACTERISTICS.

RESONANT FREQUENCY	**2200 Hz**	
SAMPLING RATE	**Up to 1.2 ksps (each axis) Serial**	***Sampling rate depends on the Communication protocol is used. Higher speeds can also be achieved.***
WEIGHT	**80 gr**	
DIAMETER	**42 mm**	
HEIGHT	**26 mm and 43 mm**	
MOUNTING	**4 x M3 , 4 x M4**	

The aluminum construction provides rigidity to the sensor and compliance effects are no significant. A high yield strength aluminum is used. Using steel instead of aluminum provides greater performance both in the single axis overload and the stiffness but raises the cost. For most of the robotic applications aluminum properties are adequate.

The Serial configurations outputs a 14 bytes stream which includes the 6 force/torque (two bytes each) components, one starting byte and one finishing byte. Figure 1 shows the byte stream.