NTNU
Norwegian University of Science and Technology
Department of Engineering Cybernetics
Department of Electronic Systems
TELE3001 - Bachelor thesis for electrical engineering

Bachelor Thesis

Biped Robot Prototype

Embedded system integration with PID controller

Trondheim, May 2020

Group members:

Jonas Brunsvik Martin Skårerverket Irfan Ljevo

Supervisor:
Torleif Anstensrud

List of Tables

List of Figures

Preface

This project is culmination of bachelor thesis and three years education in electrical engineering with specializations in electronics, instrumentation and automation. This thesis represents 20 study credits which is equal to 500 hours of work accomplished during spring of 2020. The field of robotics was an area of interest for each group member, thus providing motivation for choosing a task in this field as an object of research for Bachelor thesis at the Norwegian University of Science and Technology, NTNU.

We would like to thank our supervisor, Torleif Anstensrud, for providing group with precious insight in workings of the biped robot and the documentation required. Special appreciation goes for his help which was of great importance for the group in times of unforeseen circumstances that occurred during the semester.

Trondheim, May 2020.

Group E2023:
Jonas Brunsvik
Irfan Ljevo
Martin Skårerverket

Abstract

This thesis aims to build upon prior development of the bipedal robot prototype at NTNU, its goals are to merge the software and hardware of the robot in one embedded system, thereby creating embedded development platform for the robot upon which future theoretical work and reliable physical testing can be performed. Including to the creation of the development platform this thesis also investigates the possibility of implementing an PD controller for the control of robot.

The work done in this thesis show how the electrical wiring system of the robot was redesigned with modularity in mind and the implantation of custom printed circuit boards which simplify the connections of the embedded system. Development of new and user-friendly C++ code for each system component and the implementation of Robot Operating System (ROS) in the embedded system creates a solid basis for further development in the future, fulfilling one of the main goals of this work.

The automation part of this thesis shows how the system behavior was identified and how this is used to derive a control strategy for the robot. This research was then used in the development av a PD - controller code and the work present a suggestion on how to implement this code into the embedded system thru ROS.

Culmination of the work done in this thesis resulted in a functional embedded system for control of the bipedal robot and a suggestion on how to utilize a PD – controller for control. It shows that performance testing is necessary in future work in order to fully complete the embedded system, but in its current state it will function as a good development and testing platform.

Sammendrag

Denne oppgaven har som formål å bygge videre på tidligere arbeid utført på bi pedal robot prototypen ved NTNU. Målene denne oppgaven ønsker å oppnå er sammenføyning programvare og maskinvare inn i et integrert system som vil fungere som et grunnlag der teoretisk arbeid og fysisk testing av roboten kan utføres. I tillegg til å oppnå dette ønsker denne oppgaven å utforske muligheten for å implementere en PD – regulator som kan kontrollere robotens gange.

Arbeidet utført i denne oppgaven viser hvordan det elektriske koblings systemet på roboten er redesignet med tanke på modularitet og hvordan kretskort forenkler forbindelsene mellom system komponentene i det integrerte systemet. Utviklingen av ny og brukervennlig C++ kode for hver system komponent og implementeringen av Robot Operating System (ROS) i det integrerte systemet, lager et solid grunnlag for videre utvikling i fremtiden.

Automasjons delen av denne oppgaven viser hvordan system identifikasjon av roboten ble utført og hvordan dette ble benyttet til å utvikle en regulerings strategi av roboten. Resultatene av dette arbeidet ble brukt til å utvikle en PD – regulator kode som kan implementeres i robotens integrerte system gjennom ROS.

Resultatet av arbeidet med denne oppgaven var et funksjonelt integrert system som kan regulere en bipedal robot, i tillegg presenterer den et forslag til hvordan en PD -regulator kan brukes for å regulere roboten. Resultatet viser også at videre testing av det integrerte systemets er nødvendig for å fullføre systemet, men i sin nåværende tilstand vil roboten være en god utviklings- og test plattform.

Table of content

Glossary

SoC System on a Chip.

PCB Printed Circuit Board.

Vias are holes in the circuit boards filled with copper that makes it possible to transfer signals or power from one board layer to another.

Netlist is a list of where all connections on a board leads to and from.

GERBER files File format used for PCB production, contains relevant data for the layers of the PCB and drill guides.

IMU or *Inertial measurement unit* is a device that measures and reports body's angular rate and orientation, using a combination of accelerometers, gyroscopes and magnetometers.

Two's complement A method of representing integers in binary, can represent both negative and positive numbers.

LSB, the *Least significant bit* is the lowest bit in the series of binary numbers.

MSB, the *Most significant bit* is the highest bit in the series of binary numbers.

eQEP the *enhanced Quadrature Encoder Pulse* is a module integrated into the sitra processor that handles encoder pulses.

IDE Integrated Development Environment

Armhf or *Arm high float compiler* is an c++ compiler made to support floating point operations using ARM processors.

I/O Input output

GitHub is web based company that provides storing and documentation of software code.

PWM or *Puls width modulation* is a technique used for controlling DC power to an electrical device with pulsating current.

Matlab is a high-performance language for technical computing that is easy to use and is suitable for automation simulations.

Feedforward controller is type of controller tasked with measuring disturbance and taking corrective action before they upset the process.

Simulink is Matlab based and distributed simulation program, used for simulation of dynamic systems.

PD-controller or Proportional-Derivative controller, is control mechanism used for control and stabilisation of the system.

GPIO or *General Purpose Inputs and Outputs* is interface used by integrated circuits for interacting with different types of software and hardware.

RQT-plot is a program under ROS, that plots data.

Components , when components are mentioned, usually its a reference to all the system components : The servos, IMUS, Encoders, and Motors.

ROS or *Robot operating system* is an environment used for programming complex and robust robot software.

SSH or *Secure Shell* is a program than connects together two or more machines.

C++ class is an expanded concept of data structures, a user defined blueprint or prototype from which objects are created.

ECAD electronic computer-aided design.

dSpace is an open source repository software package.

Global variable is a variable with global scope able to be accessed by any function in the code.

1 Introduction

1.1 Background

In modern industry robots capable of mimicking human behaviour are of great importance not just for industrial purposes but also for medical work. Each year standards and accomplishments in sector of robotics and embedded systems are improving faster than before. Therefore understanding of such systems is crucial for upcoming engineers. Multiple master and bachelor thesis have been written on subject of biped gait robots here at NTNU. Current installations found on the robot are those which were used by last bachelor group in their project. Inertial measurement unit and encoders are also found pre connected to the structure. Pictures depicting robot's condition from before and after bachelor thesis can be found on the GitHub page [24]. Prior work used dSpace ControlDesk for obtaining their own system identification and simulation. Using dSpace allowed for robot to be connected to Simulink model and Matlab workspace, which was basis for the former group's work.

1.2 Preliminary design and work

1.2.1 Frame

The frame of the bipedal robot is built by the Department of Engineering Cybernetics at NTNU and are mostly made of aluminum square tubing. The frame consists of two sets of legs and a torso. When standing perfectly straight robot is 160cm tall and 100cm wide if hip actuators are included, body itself being 58cm wide. All square tubing connections are precisely machined aluminum and the square tubing of robot legs have predrilled holes (5.5mm ø) which makes it easier to mount sensors at specific locations. 45 x 25 cm aluminum plate is mounted on the torso of the frame, the purpose of this plate is to mount the control system of the robot including the power distribution system.

Mounts for the motor on the robot are also custom machined aluminum and includes press fit bearings. A custom machined bracket and actuator is mounted on the ends of the robot legs, this is where the servos on the robot are mounted. Servos are secured in place by an aluminium profile which is closer described in subsection 6.1.

1.2.2 Embedded system

The Embedded system of the bipedal robot are controlled by the single-board computer Beaglebone Black preforming the computations for the I/O operations of the robot.

The bipedal robot has input unit and output units connected via a wire system to the pins on the Beaglebone, this creates the basis for the embedded system.

Input units of the embedded system consist of two LSM9DS1 inertial measurement unit chips (IMU) from STMicroelectronics and two Scancon 2RMHF incremental rotary encoders for relative angle measurements. The output units of the embedded system consist of two Maxon 14887 DC- motors used for actuation of the robot legs, four Futaba S9254 digital servos which actuates the retractable feet of the robot and two ESCON 70/10 servo controllers used for the control of the DC motors.

Software running on the system was developed by a former bachelor's thesis group in 2019, the code is primarily developed in the programming language C++. The current software does not provide real time capabilities for data input or output control of the robot. Real time capabilities of the system are currently only possible through an external dSpace ControlDesk system.

1.2.3 Power supply

The power supply of the bipedal robot is the AIM-TTI QPX600D dual output power supply that has the capability to provide two independent supplies. It has a maximum output of 80V or 50A totaling in 600W on each output track.

1.2.4 Preliminary parts list

Preliminary parts list of the bipedal robot are provided in table 1, this includes the hardware mounted and external hardware.

Description	**Product name**	**Manufacturer**
Flexible actuator coupler	4779823	Ruland
Hip bearings	6004-C	Fag
Electric motors for leg actuation	14887	Maxon Motor
1-to-6 Gearbox for motors	Planetary Gearhead	Maxon Motor
Servo controllers for motors	ESCON 70/10 (422969)	Maxon Motor
Servos for retractable feet	S9254	Futaba Corporation
Encoders for relative leg angles	2RMHF	Scancon
Inertial measurement unit chip	LSM9DS1	STMicroelectronics
IMU Breakout board	Breakout-LSM9DS1	Sparkfun
Power supply	QPX600D	Aim-TTI
BeagleBone Black	BeagleBone Black	BeagleBoard.org

Table 1: Bipedal robot preliminary parts list

1.3 Problem statement

The problem statement of this bachelor's thesis was to further develop the Bipedal robot prototype and to create a solid basis on which future work and research can be performed thru an embedded system.

In addition to the further development of the embedded system and documentation, an effort to implement a PD-controller on the robot was also an area of focus. The areas where this thesis project have focused their efforts to achieve this are listed below.

1.3.1 Hardware documentation and improvements

A detailed overview of how the robot's wires are connected is required, development of new PCBs to replace previous connection and power supply circuits. The robot uses SPI as transmission protocol to transfer data from the IMU to the BeagleBone, this has caused problems in previous work. One task is to improve the current method of transmitting data from the IMU to the BeagleBone or find an alternative solution. Establishment of a new wiring system with the use of ethernet cables to make the system more modular and robust is also a necessary improvement.

1.3.2 Software documentation

The software of the bipedal robot is to be documented and made easily available to help future software development. A GitHub repository are to be created, providing a centralized platform which host all the development code for the robot and useful information about the bipedal robot.

1.3.3 Further development of Embedded System

All previously developed code for the embedded system of the bipedal robot will be reviewed and new code for all the sensors and actuators will be developed where necessary. An emphasis on making the new code lightweight, less complex and more user friendly is important. The goal is to develop code that can easily be understood which creates a basis for further development. All the new code will be developed in the programming language C++.

1.3.4 Real time capability

The bipedal robot real time capability is dependent on the dSpace ControlDesk, the goal is to fully implement the embedded system on the robot including real time capabilities. To achieve this, implantation of the linux middleware Robot Operating System (ROS) is necessary in order to replace the dSpace ControlDesk. This will give the embedded system on the robot the capability to control and display data from the robot in real time.

1.3.5 Servo replacement

One of the servos mounted on the bipedal robot are not functional and there are no spare replacement servos available. It is not possible to obtain the same model of servo which is mounted on the bipedal robot because it is an old model and is no longer commercially available. To future proof the servos on the robot, four new commercially available servos are to be purchased. The servos should have the same specifications or better than the replaced servos and preferably fit the same mounting brackets with minimal modifications.

1.3.6 Research controller implementation

This thesis will focus some efforts on the research into the implementation of a PD controller on the bipedal robot, with the goal that the robot could sustain gait of one or two steps. In order to achieve this, research, system identification and development of a control strategy for the robot is necessary. If the research is completed and time allows the effort will shift focus into the development of control code preferably in the programming language C++ which can be implemented within the Robot Operating System on the embedded system of the robot.

2 Circuit Design

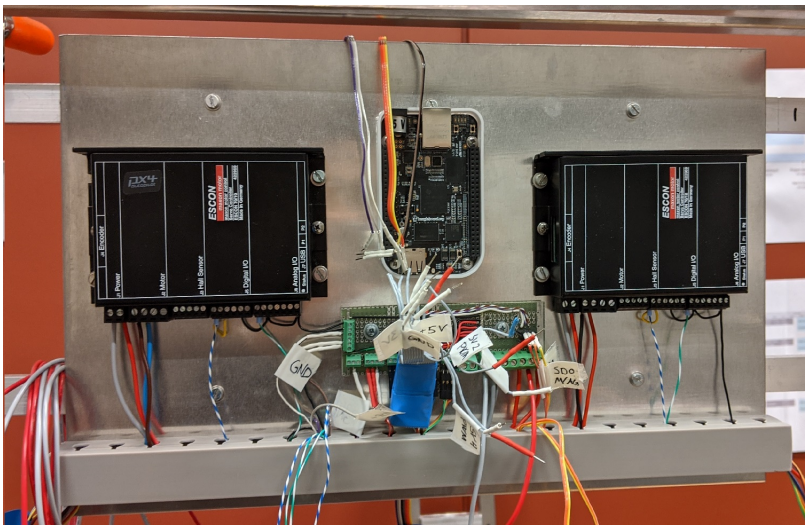


Figure 1: Preliminary circuit design

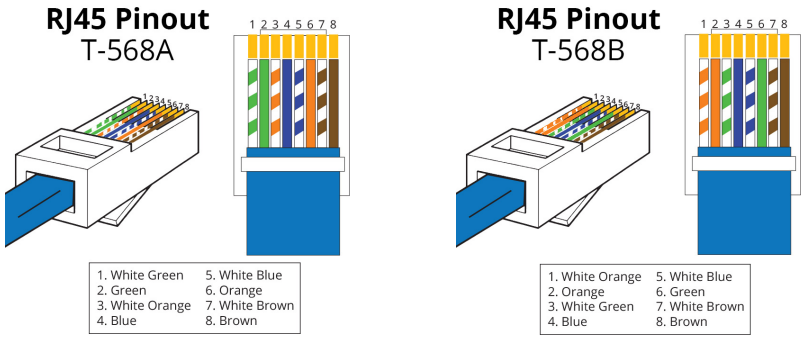
One of the main problems the group had, when first trying to understand the wiring, and how everything was connected, was understanding the documentation. Much of the robot was build up by unreliable solutions with many protoboards in its design. These protoboards was meant as a temporary solution, until a better solution was developed. To replace these protoboards the group decided to develop new PCB boards with connectors for each component.

Since the previous group had issues with wiring and noise on their system, new wiring to each component would need to developed. The new wiring scheme would be made in way that reduced the risk of connecting components the wrong way. This would hinder future groups or people using the robot from damaging the Beaglebone or sensors. The focus of this new system of boards and wiring, was to make it simple, easy to plug and play, and flexible enough for future changes to the robot.

2.1 Ethernet Cables

It was decided to use Ethernet cables were possible to replace the old wiring system on the bipedal robot. Ethernet cables was chosen because of their inherent modularity and availability, this gives the wiring system of the robot good a platform for expansion and further optimization in future work. The RJ45 connectors of the Ethernet cables provides a robust physical connection between the components on the robot which can be trusted.

Traditionally Ethernet cables are used for network connections but can be used for other applications as this project shows. Ethernet cables consist of 8 different wires in 4 twisted pair configurations, this gives us the opportunity to transfer several data lines including power/ground connection thru one shielded cable.



(a) Type A (b) Type B

Figure 2: Ethernet terminating wire configurations

Ethernet cables usually comes in two different wire configurations, Type A and Type B [3]. The most important difference to note is that they have different terminating pinouts which is important to keep in mind when the pinouts of the connection headers are determined. Since the connection headers (RJ45) pinouts in this case are determined by the designer of the custom connector boards, it is not important which type of ethernet wire configuration type is used to connect the sensors on the robot as long as the cable has the same terminating pinout type on both ends. Ethernet cables that are in a crossover configuration, which means that the cable has different terminating pinout types on each end of the cable will not work with the connector boards.

2.2 Software Tools

The (ECAD) electronic computer-aided design software used to design the printed circuit boards in this project was Altium Designer and KiCAD. In this project the ECAD software is used to design circuits with the help of schematics and a PCB layout editor, this is then used to place electronic components and shape the physical size of the PCB to the design specifications and rules determined by the designer.

The different ECAD programs where chosen because the group members responsible for the PCB design had prior experience with these two programs. Since the design workload where distributed between two group members the different connector boards where designed in different ECAD software depending on which group member was responsible for the design.

2.2.1 Altium Designer

Altium Designer is a comprehensive ECAD tool that offers a vast variety of features, Altium Designer is primarily intended for use in industry but is user friendly and offers educational documentation thru its licensing framework [4]. One drawback of Altium designer is the cost, in order to use the software it is necessary to purchase a license which can be expensive. Students at NTNU can use the license purchased by the school, thus making it free to use when connected to the school's internet.

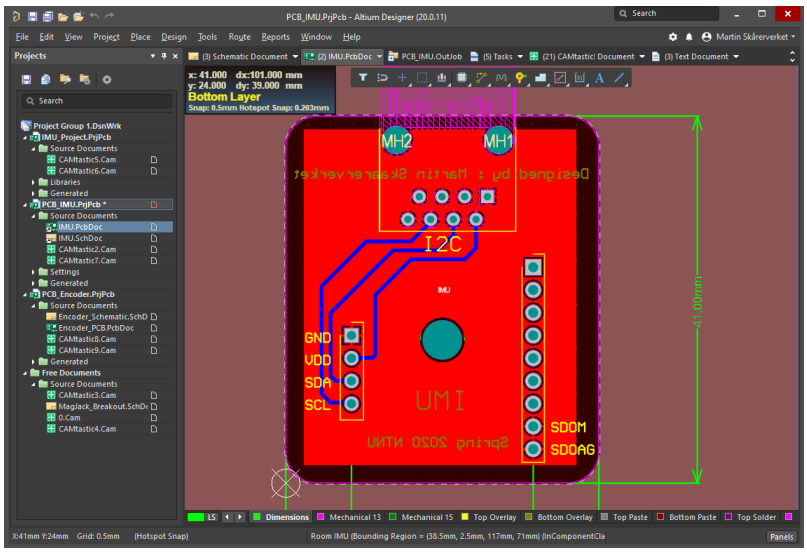


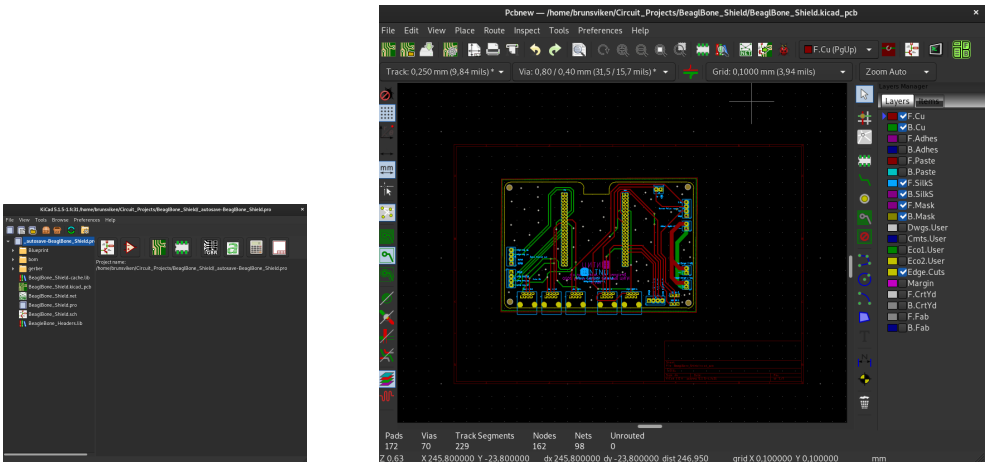
Figure 3: Altium Designer window

The software uses a schematic created by the designer where the wiring between components are determined to produce a layout of the PCB on which you can place components and traces between wire connections. It is up to the designer where the components and wire traces are placed, including the desired dimensions. The software also keeps track of the different layers of the PCB and can specify the thicknesses of copper in the layers.

A major advantage of Altium Designer is that many of the features the program offers, help automate the process of PCB design. Of which the opportunity to define design rules and to chose components from a dynamically updated list from the manufacturers are two examples of which makes the design process more efficient.

2.2.2 KiCAD

KiCad[5] is a ECAD software suit that offers a wide array of useful circuit design programs. KiCad is Open source and free for everyone, and is constantly being updated to improve functionality and adding more futures, KiCad also has an active community of users that can help if problems arise. There is also a lots of "how to" videos on YouTube, to help new users learn the software. KiCad was used because of previous projects done in the class Anvendt Elektronikk in second year[27]. The three main programs in the suit is: The Schematics program, where you draw in all the different components; The PCB designer, where you draw in traces, footprints and vias on the circuit board; And the Gerber Viewer, where you inspect the boards cad files. KiCad also futures a circuit simulation program where circuits can be tested and analysed.



(a) Software suit menu (b) PCB-new software

Figure 4: KiCAD Software Suit

One disadvantage of KiCAD is that some of the components and footprints are made by the "community". Sometimes footprint dimensions could be wrong, or pins labeled with the wrong name. this could lead to chips that do not match footprints, or other similar issues. These issues are not as prevalent as they used to be before, but still there. KiCAD also does not connect the different footprints to components on a BOM list. This means that all the components need to be searched up separately on websites, which can be time consuming work.

2.3 Main Board

The main board functions as a "cape" that connects to Beaglbones P8 and P9 headers. The focus of the design was to make it easy to connect and disconnect all the different components the robot uses. To achieve this the board uses "Ethernet ports" to connect to the IMUs and Encoders, and ordinary "screw Terminals" for the motor and Servo connections.

To power the the Motor the board also splits 48 volts to two separate screw connectors for each motor controller. This circuit is separated from the other part of circuit, with its own Ground, and extra wide traces to handle the extra current the motors use. All the different voltages from IMU :3.3 Volts,Encoder: 5 Volts, and servo: 7.4 Volts, were brought inn through one 4 pin screw terminal called "sensor power inputs". The 5 volt line powers the beagle-bone and the Encoder, the 3.3 volt line goes to each IMU,and the 7.4 Volt powers all the servos. For future changes or problems it was also decided to added extra PWM pins and GPIO pins.

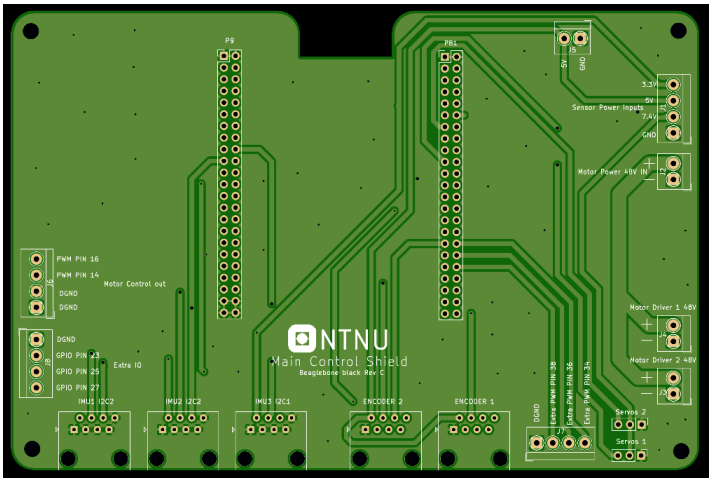


Figure 5: Front Rev. A

The CAD design of the Main board, was created in KiCAD, were the first thing made was a Schematic. The schematic is where all the different components are added, and connected together. The connections are then saved in a Netlist , which is imported into The "PCB Layout editor" where traces, vias and form factor is drawn on The "board.

2.4 IMU Connector Board

The IMU connector board was designed to help facilitate the LSM9DS1 breakout board created by Sparkfun and to connect the LSM9DS1 IMU chip thru R-45 connectors and Ethernet cables to the Beaglebone. The IMU connector board replaces the old prototype board used to connect the IMU to the Beaglebone which was created by a prior bachelor thesis group. It was decided that the new IMU connector board should be a PCB and the IMU connector board was designed with the PCB design tool Altium. The IMU connector board PCBs was produced in China by the manufacturer JLCPCB. The PCBs could also be produced locally at NTNU by the electronics and prototype laboratories, but because of their limited capacity it was decided to use JLCPCB for production.

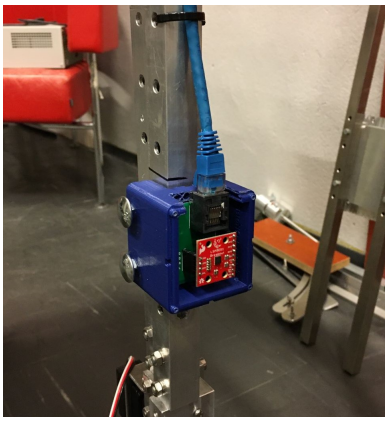
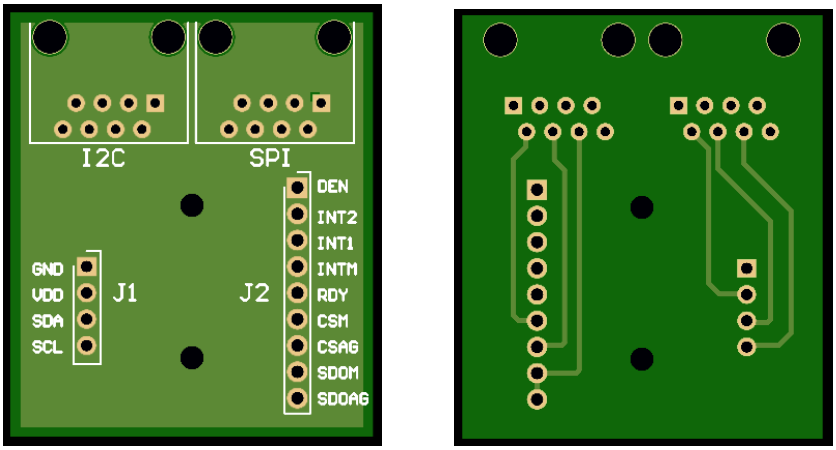


Figure 6: IMU connector board Rev B. mounted on the Robot leg

Initially the connector board was designed to accommodate two communication protocols (I2C and SPI), this was done to be able to test the two different communication protocols without the need to change the hardware. The connector board was also designed to fit into the 3D printed IMU case which was already mounted on the robot.

2.4.1 Revision A

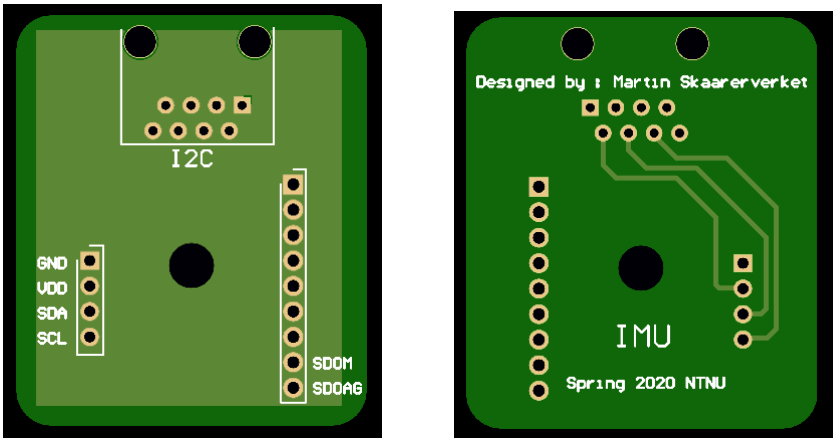


(a) Front side (b) Back side

Figure 7: IMU connector board Rev. A

The first revision (A) of the IMU connector board PCB is a 2-layer PCB which uses the top layer as a ground plane and the bottom layer is used to connect the different data connection pins from the LSM9DS1 breakout board to the pins on the RJ45 headers on the board. The wiring schematic used to create the IMU connector circuit can be viewed in appendix A, Figure 34. The width of the copper traces used to connect the pins on the boards bottom layer is a standard 0.5mm. The board was designed with connections for both I2C and SPI communication, to achieve this it was necessary to use two RJ45 female headers mounted on the board. To utilize SPI, both RJ45 headers needs to be connected to the Beaglebone. When using the I2C protocol, only the RJ45 header labeled I2C needs to be connected. 4-pin and 9-pin single row female headers was placed on the board with the purpose of connecting the Sparkfun breakout board with the IMU connector PCB, the dimensions provided by the Sparkfun website [2] made this easy to achieve. Finally, silk-screening was used to make it easy to identify where the different pins was located, on the board and which RJ45 header is used for the different communication protocols.

2.4.2 Revision B

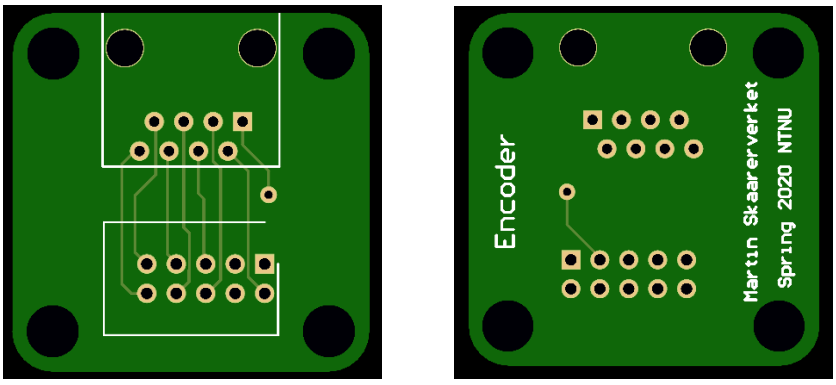


(a) Front side (b) Back side

Figure 8: IMU connector board Rev. B

The second revision (B) of the IMU connector board shares the same layer structure as the first revision (A), with a ground plane on the top layer and the pin connections on the bottom layer. The major difference between the revisions is that the second revision only supports I2C communication, and thus only have one RJ45 female header. The design is also updated with a new schematic which can be viewed in appendix A Figure 35 that connect the SDO M and SDO AG pins to ground, this is done because the LSM9DS1 chip only has two sets of I2C addresses (0x1C/0x6A or 0x1E/0x6B)[2], and to select the I2C address you desire these two pins needs to be grounded. The edges of the PCB have been rounded to give the final product a more professional appearance and the silk-screening have been updated to identify the pins in use. The mounting hole in the middle of the board have also been expanded to 5 mm to accommodate a more common diameter for the mounting screw.

2.5 Encoder Connector Board



(a) Front side (b) Back side

Figure 9: Encoder connector board

The encoder connector board was designed to connect the encoder mounted on the motor, to the main connector board with the Beaglebone. The encoder circuit interfaces the 10-pin connector from the encoder with the RJ45 header which transfers data thru an Ethernet cable to the Beaglebone. The encoder PCB is designed to be as compact as possible and to help make the wiring system of the robot modular. The same 2 -layer PCB design was used to implement the encoder connector circuit as the other PCBs in this thesis, but the encoder board does not implement a ground plane and instead connects all the pins directly with traces. The wiring of the encoder circuit is available in appendix A, figure 36.

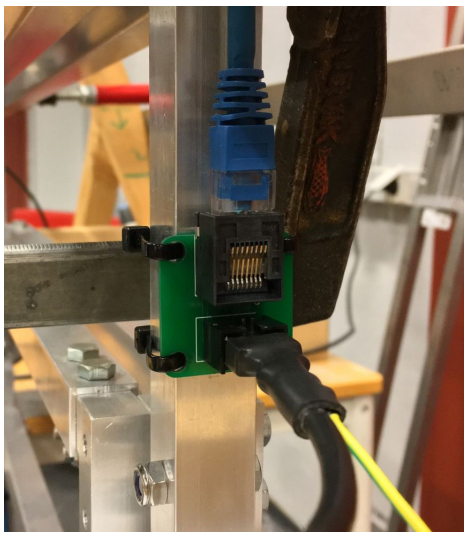


Figure 10: Encoder connector board mounted on the Robot

2.6 Manufacturing PCB JLCPCB

JLCPCB is a PCB prototype and production manufacturer based in Hong Kong with production facilities in Shenzhen, China. The company specializes in production of small batch prototype PCBs and offers fast production and delivery at an affordable price. [6] The members of the group responsible for the PCB design had good prior experiences with the manufacturer, both in quality of the product and fast delivery. This was one of the main reasons for the decision to use JLCPCB as the only manufacturer. JLCPCB was not the only option considered, The Electronics Production lab at NTNU was also considered but it was determined that the limited capacity of the lab could result in long production times.

Ordering from JLCPCB is done thru an automated process, which consist of uploading GERBER files and Drill files produced by the ECAD tools then selecting different options for the appearance and thickness of the PCB. When the uploaded GERBER and drill files are approved for production the PCBs are produced within 48 hours and shipped. The PCBs in this thesis was ordered in two batches, the IMU connector board Rev A were ordered early because it was needed for testing different communication protocols. The main connector board, Encoder connector board and the IMU connector board Rev B were ordered as one batch later in the project to save on the cost of shipping.

2.7 Components PCB Mouser electronics

Mouser electronics were chosen as the distributor for ordering the components on the PCBs, this distributor was chosen because of its large selection of components on offer and large stock of each component. Mouser electronics is an American online distributor of electronic components[7] which has warehouses across the world, more importantly they have a warehouse located in Sweden which all the components in this project was shipped from. This meant fast delivery times, which also were an important reason for choosing Mouser electronics. An overview of the components mounted on the PCBs are available in Appendix B Figure 4,5,6.

2.8 Discussion

The whole wiring system and cable management of the robot was changed, so it would be easier to solve future problems. It would also make the system more resistant to human errors like connecting something wrong, shorts, or cables disconnecting during testing. The boards were continually developed during the whole bachelor period, which meant things changed constantly. Some problems also became apparent after the board was ordered from JLPCB. One of those was that the group had assumed that the encoder could use 3.3 volts, and not 5 Volts. This issue was not fixed after ordering the PCB, and meant the group had to use external logic converters for the signals between Beaglebone and the encoder. Not using logic converters could potentially damage Beaglebones pins, or worse.

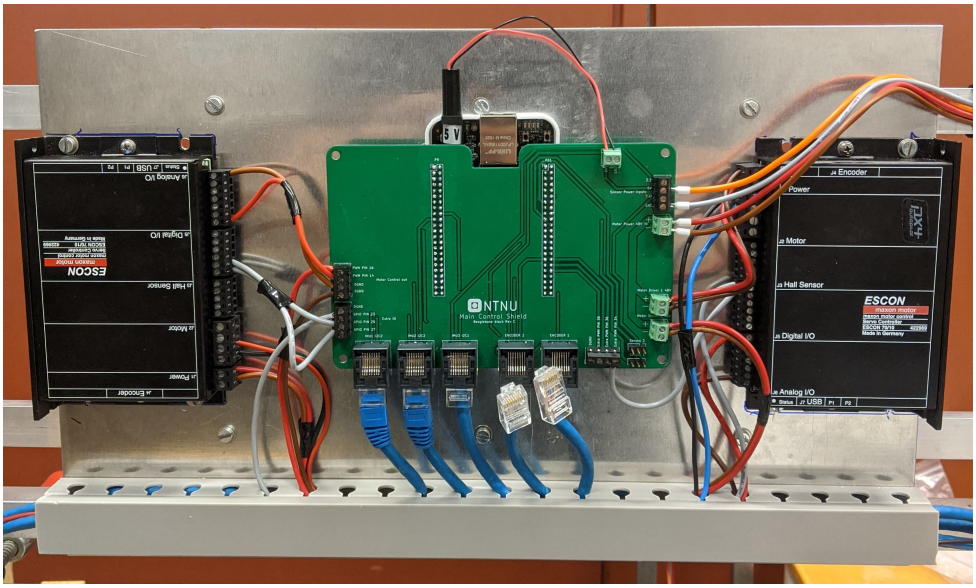


Figure 11: Final circuit design

The last bachelor group used SPI to communicate with the IMU's, but could not get this to work properly. The main issue was believed to be too much noise on the communication cable because of cable length, but also because of the 3.3 volt logic level used. These problems could have been solved in many ways, but the group found it best to remove the old wiring and replace it with twisted pair cabling to reduce noise. It also became apparent that SPI was unnecessarily complex for transmitting the IMU data, since communication was mostly was sent one way. And implementing SPI to Beaglebone in C++ would also be harder than I2C. Ethernet cables were also chosen because of the modularity of the connectors, and that this connector system is readily available easy replace and affordable. The cost of making the circuit boards used in this project is negligible compared to the time saved in figuring out wiring setups in the future. Designing these boards also saves future costs, by reducing the risk of something getting damaged by short circuiting or other faults caused by wrongfully connecting wires. These circuit boards are especially important in reducing these risks, since Beaglebones pins don't handle voltage levels above 3.3 volts very well. The group ended up destroying one Beaglebone because of exactly this issue, where 7.4 volt was connected to one of Beaglebones PWM port, and ended up destroying the Beaglebone.

3 Embedded Development

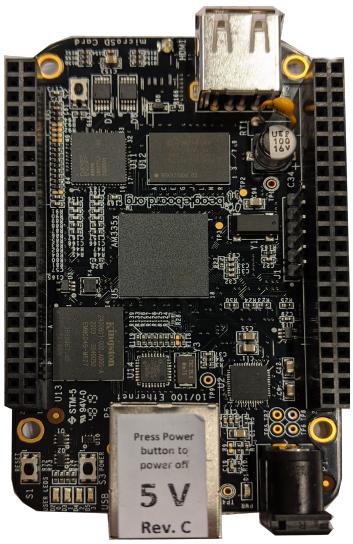


Figure 12: Beaglebone Black

The goal of this project was to read all the sensors, and to control the motor and servos from the Beaglebone. This was to remove the costly dSpace ControlDesk from the project, and make it open source and easy to modify. It was decided to develop new code for all the sensors; servos and motors. Since the servo type was changed and the existing code of the embedded system too was complex, and not user-friendly for people new to Embedded development. The Arduinos were also removed since they represented an unnecessary complexity to the communication system between the Beaglebone and IMU. The previous bachelor group reported slow and erratic SPI communication between the IMU and Beaglebone, and random Beaglebone shutdowns[1]. Because of this, the group decided to redo much of the code written for each module in Beaglebone. Since the code from the last group was written in "Arduino C" and in C++, and most of the members knew how to code in C, the new code was written in standard C++.

To write good code that could be used in the future of this project, the following goals was set:

- The code needed to be easy for beginners to understand and user-friendly.
- It needed to be lightweight and be able to run fast.
- Low latency, so that it do not slow down the robot's response time.
- The code needed to include some basic fault checking, for debugging purposes and robustness.

The coding effort was split into 4 different parts: The IMU, encoder, servo, and motor code. Focus was first placed on making a simple code run flawlessly on Linux with each module, and then gradually implement fault checking and new futures to the code.

3.1 Beaglebone Black

The Beaglebone Black is an open-source hardware System on Chip (SoC) development board made for intermediate level developers[8][15]. The board comes with a AM335x 1GHz ARM® cortex-A8 processor as main processing unit and also with two separate 32-bit processing units (PRU) made to handle time sensitive work loads. Beaglebone black comes with 2x 46 pin headers, that include: 2x I2C busses, 2x SPI Busses, 12 PWM pins and other useful Digital communication pins[17].

3.2 Linux and Development Tools

Beaglebone black can run a variety of different Linux distributions. Determine the right distribution that can run ROS, but also interfaces with all the PINS correctly were important. Ubuntu is an Linux distribution based on Debian, another more general distribution[13]. Ubuntu works flawlessly with ROS, and is a widely used distribution with vast community support and software support in general. this distribution was installed on the Beagleboard Black from the Beaglebone wiki[14], where the process is well documented and explained.

Code for all the system peripherals were developed in Visual Studio Code and Visual Studio 2019. The group chose two different ways of compiling code, both methods worked well.

One member chose to use Visual Studio 2019 IDE for development, this solution compiles locally on the computer with the armhf compiler and then transfers the binaries to Beaglebone thru a ssh connection. This process is mostly automated by the IDE and the user only has to specify the ssh target connection and which compiler to use. To access this functionality in Visual Studio 2019 it is necessary to install the tool-set named “Linux Development with C++” in Visual Studio[26].

The second member chose to use Visual studio Code (VSCode) in a Linux environment with the following extensions : C/C++ with IntelliSense, that implements c style development to the IDE; and remote SSH, that set up a SSH server on Beaglebone. This makes it possible to change files using a GUI environment instead of in ”nano” or ”vi” using a terminal environment. VSCode does not compile the code without editing the corresponding build task json file, and that is a tedious task. Calling the Beaglebone’s armhf compiler in the terminal were used. The binaries were then executed with “./binaryname”.

3.3 IMU

The inertial measurement unit used in the embedded system of the bipedal robot is the LSM9DS1 iNEMO system-in-package module developed by STMicroelectronics, this module has the ability to sense a total of 9 degrees of freedom (3-axis accelerometer, 3-axis gyroscope, and 3-axis magnetometer) and includes support for both SPI and I2C for communication [22].

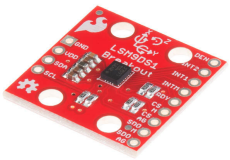


Figure 13: Sparkfun LSM9DS1 Breakout Board

The IMU are implemented on the robot with a breakout board for the LSM9DS1 chip produced by Sparkfun, this simplifies the process of connection to the pinouts of the chip and the interfacing with the IMU connector board which pass the data through ethernet cables to the Beaglebone[2].

3.3.1 IMU code

The Linux kernel has its own i2C c++ libraries [18] to interface code with the I2C pins on the Beaglebone. These libraries greatly reduced the workload and complexity of the code necessary to interface the beaglebone with the IMU. The libraries are included like this :

```
#include <linux/i2c-dev.h>
#include <sys/ioctl.h>
```

Listing 1: C++ code using listings

From these libraries ioctl() were used to set up the register on the IMU, and i2c communication on Beaglebone. Ordinary system calls like read(), write(), are then used to interface with the sensor [21].

```
void setRegSensor(char reg,char value){

    char *bus ="/dev/i2c-2";

    if((file = open(bus, O_RDWR)) < 0)
    {
        printf("Failed to open bus. \n");
        exit(1);
    }
```

Listing 2: Open I2C bus

In order to access the I2C addresses of the IMU, one first need to open the I2C bus on which the IMU are connected. The code in listing 2 shows how the library is first used to open the specified i2c bus system address and specifies the access mode as read/write, this is then combined in an if statement which is done for the purposes of error checking.

```
ioctl(file, I2C_SLAVE, addr);

    char config[2] = {0};
    config[0] = reg;
    config[1] = value;
    if(write(file, config, 2) < 0)
    {
        printf("Failed to open bus. \n");
        exit(1);
    }
    else
    printf("sucessful init \n");
```

Listing 3: Write to I2C address

When I2C bus is open, the IOCTL system call is used to specify the I2C address where the IMU writes data. In order for the IMU to output data to the I2C register, it is necessary to write 8-bit values to the different registers of the IMU chip [23] . This is how to configure the IMU and enable/disable functionality of the LSM9DS1 chip. The code in listing 3 show how the write function uses an array containing the register address which will be configured in the first position and the value we wish to write to this register in the second position. In addition to this the code also uses the same error checking methods with the inclusion of an if statement.

```
char dataBuffer[5] = {0};

void ReadSensor()
    {
        char regis[1] = {0x28};
        write(file, regis, 1);

        if(read(file, dataBuffer, 6) != 6)
        {
            printf("Input/Output error \n");
            exit(1);
        }

    }
```

Listing 4: Read IMU I2C address

When the IMU are configured to the desired data outputs, the IMU chip will output raw sensor data in 16-bit two's complement format for each output axis staring at specified register addresses. In order to read the values for all the axis's it is necessary to read a total of 6 bytes per function (accelerometer, gyro ect.) in the I2C register. The code in listing 4 shows that in order to read values from theses bytes from the I2C register, you first have to write to the register address where the desired data starts. In this specific case the accelerometer data starts at the hex value 0x28 address in the I2C register and ends at 0x2D address, this is 6 bytes of data where the accelerometer data is contained. The code uses the read function to fill an empty array which in this case are named dataBuffer with the data from the accelerometer data registers, this is an efficient way of storing the data before it is converted into readable values in another function. This procedure for reading data from the IMU is the same regardless of reading the data from the gyro or the magnetometer, the only difference is the hex address where the code starts to read values.

```
#define g_sens_accs 0.061
```

.

Listing 5: Define linear acceleration sensitivity

It is important to define the sensor characteristics for the sensitivity configured of the raw output data. This value is used to convert the raw data into human readable data. listing 5 shows the defined linear acceleration sensitivity when the accelerometer is configured to have a sensitivity of ±2g. This value has the unit mg/LSB. Values for sensor characteristics can be found in the LSM9DS1 datasheet on page 12 [23].

```
{
    s_ax = ((dataBuffer[1] << 8) | dataBuffer[0]);

    ax = (s_ax*g_sens_accs)/1000;

}
```

Listing 6: Convert raw data from IMU register

Values from the read function is stored in a array called dataBuffer, the code in listing 6 shows how the first two bytes of this array are converted from 16-bit two's complement to a human readable integer which has the unit of the gravitational force equivalent g. It achieves this by preforming a binary leftshift of 8 places on the LSB (dataBuffer[1]) and then use the boolean operation OR on the MSB and LSB. The result is then stored in the variable s_ax which is then multiplied with the defined linear sensitivity and divided by 1000, the last step is preformed in order to achieve the desired unit g.

This section describes the most important elements of the code running on the embedded system involving the IMU. Complete code can be found in the bipedal robot GitHub repository [24] and in appendix C.1.

3.4 Encoder

The encoder used in this project is a Scancon 2RMHF industrial encoder that outputs two square wave pulse signals, 3000 times per revolution. To read these signals, the eQEP modules on the Beaglebone are used. The eQEP module reads the two pulses from the encoder and transform this into a number that increases 12000 times per revolution. This number is then stored in memory and is accessible through the interface file "/position" in the "sys/devices/platform/ocp/..." file system.

Figure 14: Scancon 2RMHF encoder

Encoders on the robot uses separate eQEP pins on the Beaglebone, so each encoder gets its own interface file in their own folder in the /ocp file system.

3.4.1 Encoder code

The eQEP module have ability to produce encoder readings in two different modes, incremental or absolute, for this project the absolute mode is the best selection. Because the encoder readings start at zero and is incremented or decremented determined by the movement of the encoder. This will give the opportunity to convert the encoder readings into real angular position of the robot legs in either direction.

To set the mode of the eQEP module it is necessary to open the file location where the modes are determined. The code shown in listing 7 use the fopen function to open the specified path where the mode is determined and specify to write at this location, this is then stored in the file setmode. The fprintf function then prints the integer 0 in the setmode file at the correct path for mode selection. In this case the integer 0 results in the mode being set to absolute.

```
FILE* setmode = fopen((this->path + "/mode").c_str(), "w");

        fprintf(setmode, "%u\n", 0);

.
```

Listing 7: File pointer in Code

To access the reading from the eQEP modules the code in listing 8 shows how the fopen function is used to open the specified file location of the readings, then reads the value and stores in the file fp. The fscanf function then scans the file fp and stores the values as decimal integers in the position variable.

```
FILE* fp = fopen((this->path + "/position").c_str(), "r");

        fscanf(fp, "%d", &position);
```

.

Listing 8: File pointer in Code

Based on work done by prior a bachelors project, one has to apply the equation 1 to the readings from the eQEP module in order to convert the readings to the exact angular position of the bipedal robot legs[1]. The equation takes in consideration that the motors mounted on the robot has a gear reduction of 6 and multiplies this with the reading per revolution of the encoder which is 4×3000. 360 degrees divided by this, results in a value which can be used to convert the eQEP readings to degrees.

$$\Theta = \frac{360^\circ}{6 \times 4 \times 3000} = 0.005 \tag{1}$$

The code in listing 9 show how the result of equation 1 is multiplied with the readings of the encoder and printed out. The printed result will then be represented in degrees.

```
std::cout << encoder.getPosition()*0.005 <<std::endl;
```

.

Listing 9: Printing out converted encoder readings

This section describes the most important elements of the code running on the embedded system involving the encoders on the robot. Complete code can be found in the bipedal robot GitHub repository [24] and in appendix C.2.

3.5 Servo

Four Turnigy TGY-SC340V 7.4 Volt servos were used in the project to control the actuators at the end of the robot legs. To control the servos two PWM signals is used. Each signal control one set of actuators on each leg. Beaglebone comes with plenty of PWM pins, pin P8_13 and P8_19 were selected to be used for the PWM signals for servo control.

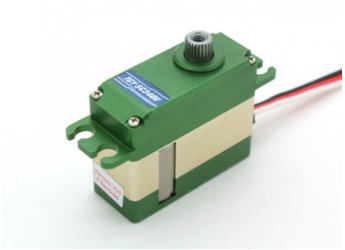


Figure 15: Turnigy TGY-SC340V servo

3.5.1 Servo code

To produce PWM outputs on the Beaglebone the code first needs to set the selected pin to pwm mode in the state interface file in the "/sys/devices/platform/ocp/... path". This is done using the ofstream object that outputs "pwm" to this file location. Shown in listing 10.

```
std::ofstream pwmstate;
pwmstate.open("/sys/devices/platform/ocp/ocp:" + m_Pinpath + "/state");
pwmstate << "pwm";
```

Listing 10: file stream than changes state

A frequency is set using the path /sys/class/pwm/... /period file, with the same ofstream technique as setting the state above. The last things to set was the DutyCycle (position) in "/dutyCycle" and to enable pwm with "1", in the "enable" file using the technique.

The frequency commonly used to control the type of servos mounted on the robot is 50 Hz, in order for the bealebone to produce the desired frequency of the pwm signal the frequency needs to be represented as a period in nanoseconds. 50 Hz equals a period of 20000000 nanosconds, this value is written to the appropriate path shown in listing 11.

```
int frequency = 20000000;

std::ofstream freqSet;
freqSet.open("/sys/class/pwm/pwm-" + m_PinpathNumber + "/period");
freqSet << frequency;
freqSet.close();
```

Listing 11: set frequency of the PWM signal

To activate the actuation of the servos, one has to change the duty cycle of the pwm signal. The duty cycle determines the position of the servo, the group conducted testing of the servos at different duty cycles and determined that servos would need to preform two different positions, one position where the servo actuator is extended and one position where the servo actuator are retracted. The range of which the servos could safely operate was between approx. 4.1% duty cycle which result in the actuator being retracted and 9.5% duty cycle which result in the actuator being extended, This corresponds to 820000ns and 1900000ns pulse width. The duty cycle was calculated using equation 2

$$Duty\ Cycle\ (\%) = \frac{Pulse\ width}{Period} \times 100\% \tag{2}$$

The calculated pulse widths are set using the code shown in listing 12.

```
 void setDutyCycle(int dutyC)
{
    std::ofstream DCfile;
    DCfile.open("/sys/class/pwm/pwm-" + m_PinpathNumber + "/duty_cycle");
    DCfile << dutyC;
    DCfile.close();
}
```

Listing 12: set Duty cycle example

This section describes the most important elements of the code running on the embedded system involving the servos. Complete code can be found in the bipedal robot GitHub repository [24] and in appendix C.3.

3.6 Motor

The motor used in the embedded system of the robot are the Maxon 14887 DC motor which has the capability to output 150W, the motor is connected to a gearbox which gears the motor 1-to-6. The motor is controlled with the help of an ESCON 70/10 servocontroller. This servocontroller needs a PWM signal as input, where the duty cycle determines the current output of the motor. The servo controller also has input pins for the determination of the output direction of the motor.

(a) Maxon Motor 14887

(b) ESCON 70/10 Servocontroller

Figure 16: Motor and servocontroller Embedded system

Before the code for the motor is run its important to configure the servo controller with the ESCON studio software. Work done by a prior bachelor thesis group determined that the maximum required amperage for the motor should be 8.3A[1], this should be configured with the ESCON software including preforming a autotuning.

3.6.1 Motor code

Since the servocontroller controls the motor, the code running on the Beaglebone only needs to configure three output pins. Two general purpose I/O (GPIO) pins for the control of direction and one pwm output pin for the current output of the motor. The PWM pin is configured using the exact same method as shown in the section about setting PWM pins in the servo code above.

To configure the GPIO pins the same method was also used with some minor modifications, the code implements the ability to toggle the selected GPIO pin. Depending on the configuration of the servocontroller, the toggling of pins will affect the motors direction. The code is made to be modular and control of the torque of the motor is also possible. This can be done by changing duty cycle of the PWM output of the duty cycle, where a lower duty cycle means lower torque.

Code showing how the GPIO pins are toggled to change the direction of the motor are shown in listing 13

```
control.GPIOtoggle(9,25,0);   //set GPIO pin 25 on cape 9 to 0
control.GPIOtoggle(9,23,0);   //set GPIO pin 23 on cape 9 to 0
std::cin.get();
control.GPIOtoggle(9,25,1); //set GPIO pin 25 on cape 9 to 1
control.GPIOtoggle(9,23,1); //set GPIO pin 25 on cape 9 to 1
std::cin.get();
```

Listing 13: Toggling of GPIO pins for motor control

This section describes the most important elements of the code running on the embedded system involving the motor and servo controller. Complete code can be found in the bipedal robot GitHub repository [24] and in appendix C.4.

3.7 Instruments and Testing

3.7.1 Oscilloscope

A Siglent SDS1104X-E oscilloscope borrowed from one of the group members was used during coding and testing. PWM signals from the Beaglebone, and square wave signals from the encodes where measured with it.

3.7.2 Lab power supplies

two Lab bench top supplies were also used to power the servos and motors during testing. one was a Siglent SPD3303C power supply borrowed from one of the team members, and the other was the Aim TTI QPX600D power supply.

(a) Siglent SPD3303C

(b) Siglent SDS1104X-E scope

Figure 17: Instruments

3.8 Discussion

The group continued to use Beaglebone instead of other boards like raspberry PI, since it was a good choice and filled much of the project's connection needs and processing speed. It was also a decent amount of documentation about developing on the Beaglebone, and especially Derek Molloy's videos and website stood out as a valuable source of knowledge and documentation about Beaglebone development.[20]

3.8.1 Choosing the right distribution

The group chose to run Ubuntu on Beaglebone, This decision was based on using ROS to connect all the sensors and actuators. Since ROS primarily runs on Ubuntu and running other Linux distributions could potentially lead to extra unnecessary work, it was clearly the best choice. Even though the group chose Ubuntu, the group did use some time researching other distributions, one distribution that later in the project stood out, was Angström, and RTOS. Angström being the most interesting with its connections to Yocto project[19], a project to make it easier do develop software on embedded platforms. Choosing those distributions would mean increased time learning to use these distribution, and potentially loosing ROS as a tool.

3.8.2 Choosing developer tools

A decent amount of time was spent on choosing good developer tools for this project. The group ended up using two different programs with different solutions. One of them, was Visual Studio Code with remote ssh connection to Beaglebone, and the other, using a cross compiler in Visual Studio, with connection to Beaglebone. The group could have chosen to use the same developer tools the last bachelor group used, but using familiar IDE's was better than learning a new ones. some time was also spent learning about cross compilers and tool chains worked, and knowledge from learning how these tools and concepts worked helped the group later when problems arose during the coding phase of the project.

3.8.3 Real time units

It was chosen not to use the Beaglebones Programmable real time units or PRUs, Since getting to know how to code these cores would be time consuming and the group didn't want to invest that much time into something potentially unnecessary. The last bachelor group also believed that the main process would be more than fast enough for this application, and that it would be unnecessary to implement the PRU cores [1]. Even though the PRU's were not implemented they could later become relevant if the main processor gets too much load, or that the control system needs a faster response time from Beaglebone.

3.8.4 Removing Arduino and changing to I2C

In light of the problems with erratic behavior from the Arduino part of the system, it was decided to remove to the Arduino from the system. The Arduino became unnecessary, since the group decided to send the IMU data directly to Beaglebone, and that Arduino would slow down communication by being an extra complexion. After some research and reading through the IMU datasheet, it was also decided to replace SPI with I2C. It would be much easier to use I2C code than SPI code on Beaglebone, and I2C would also be more than fast enough for our IMU transmitting speed. The communication to and from Beaglebone to IMU would also mostly go one way, since the IMU continually sends data one way, and Beaglebone only sends data when configuring the IMU. The IMUs could also not send data faster than 952Hz, so a faster communication protocol would be unnecessary. SPI is neither suitable for communication over distances, so using it on this project would complicate the problem even further.

3.8.5 Issues and learning

Since neither of the group's members had coded on such a complex embedded system like Beaglebone before, it took some time to learn the key concepts. Familiarity to more simpler Atmel micro-controllers and how they worked proved useful. Especially how registers work, how data-sheets presented registers , and how to change these values in the registers proved valuable.

During coding for the servo and motor, the group got stuck on some hard problems. The biggest issue was that the Beaglebone used during the last bachelor assignment was damaged, and after much time spent debugging the group noticed that pin 13 on Beaglebone would not respond to PWM output requests. This issue lasted a few days, but solved fast after testing the same code on another Beaglebone and seeing it worked. The group also noticed that Beaglebone used a few run cycles to enable PWM on each pin, and that this time differed from time to time.

3.8.6 Instruments

During coding for the encoder, servo and motor, a oscilloscope was used to view the output from PWM pin on Beaglebone,and at the encoder outputs. This was really useful during coding phase, since it was impossible know if problems arose from connection issues or coding. Especially looking at duty Cycles and frequencies during the problems with pin 13 and with testing the Encoders and motor outputs and input signals, having a oscilloscope available proved useful.

4 Controller code

Writing C++ code for controller starts off with defining controller class. It is possible to make a functioning controller code without using classes, but for simplicity purposes using classes is a good way of storing multiple functions in one place.

```
#ifndef pd.h
#define pd.h
```

Listing 14: Class header

Worth mentioning is that there are two different files. One which stores controller class and second one where function content is defined. Therefore header guard like 'ifndef' has to be used to protect the header file from being included more than once. After performing such check and concluding that 'pd.h' is not used, that same token can be used for defining the class file. Dividing controller code into two files is done so that code can be easily inspected and understood, same effect can be achieved if everything is in one file.

Class is divided into two sections, private and public. Private section cannot be accessed from outside of the class while public can. All functions are defined in public section so that they can be used and given functionality later in code.

4.1 Functionality

Running the code first time, controller algorithm will require previous error and control signal values which are not available in the beginning. That will cause a problem, therefore error values have to be initialised in the beginning of the code. That also implies that such function should be executed only once and not be called again during process.

```
PD_Controller::Initialise()
{
  float error = 0.0;
  float previous_error_1 = 0.0;
  float previous_error_2 = 0.0;
  float previous_controlSignal=0.0;
}
  PD_Controller(float kp, float kd)
  {
    Initialise();
    KP=kp;
    KD=kd;
  }
void PD_Controller::SetOutputRange(float min_Output, float max_Output)
{
  float MAX_OUTPUT=max_Output;
  float MIN_OUTPUT=min_Output
}
```

Listing 15: Initialisation function

Same task has to be performed with other parameters like gain, signal limitation and setpoint.

```
  void Set_Setpoint (float setpoint)
  {
    float Set_Setpoint = setpoint;
  }
```

Listing 16: Updating setpoint value

```
float PD_Controller::getControllValue(float set_SetPoint, float sensor){
  float error = setPoint - sensor;
  float controlSignal = previous_controlSignal+(KP*(error -

  previous_error_1))+(KD*(error-2*previous_error_1+previous_error_2));

 if (controlSignal>MAX_OUTPUT)
 {
   controlSignal=MAX_OUTPUT;
 }
  else if (controlSignal<MIN_OUTPUT)
  {
    controlSignal=MIN_OUTPUT;
  }
  return controlSignal;
}
```

Listing 17: Control signal function

Function which is performing control signal calculation is the one that must be placed in loop since it has to be updated constantly. Input arguments for this function are setpoint designated by the user, and a sensor which tracks deviation from the desired setpoint. Function performs calculation of the control value, compares it to the signal limitation designated by the user and at the end returns control signal value.

```
void PD_Controller::Update()
{
  float previous_error_2=previous_error_1;
  float previous_error_1 = error;
  float previous_controlSignal=controlSignal;
}
```

Listing 18: Updating previous terms

Error and control signal terms have to be updated constantly since those are main component of control algorithm. Having a separate function for performing such task is not necessarily something that has to be done, but it was chosen to do so due to the simplicity in potential troubleshooting.

4.2 Discussion

Writing controller code and deciding upon controller algorithm can be approached in many different ways. Group decided to use this specific algorithm because it practically shows how backward approximation is performed in C++ code. Much of the time was spent on understanding how to use class programming to produce controller code that is easy to use and understand. In addition considerable efforts were made in deciding which controller algorithm would be best to use. Upon review it was decided that using Seborg's *Process Dynamics and Control* book[34], page 136, would be best since detailed explanation of equation behind algorithm is provided in the book, something that is not often the case with online examples. Further explanation of the algorithm is provided in chapter 7

5 ROS - Robot Operating System

Contrary to what the name states ROS – Robot Operating System is not an operating system, but rather a middleware which function as a framework for development of robot software[9]. ROS was created with flexibility in mind, meaning that the user has the freedom to implement their specific robot application within the ROS framework however they wish using configuration, tools and libraries which interact with the ROS core.

ROS is built on the foundation of open-source software and collaboration, this has resulted in a big community with a vast amount of open documentation, example code, support and tutorials. This has made it easier beginners to implement ROS on their embedded system, the modularity of ROS also makes it possible to implement as much of the embedded system as the user wants or have the knowledge to accomplish.

5.1 Framework

The framework of the system is based on a graph model with nodes and topics (Figure 18). In order to create a ROS client, it is necessary to have a process called ROS Master. The ROS master process Is responsible for setting up the node-to-node communication trough topics in a publisher or subscriber configuration. The communication between nodes do not need to pass through the ROS master, making it a decentralized architecture which is beneficial for robotic control systems utilizing a network of sensors and actuators connected to a SoC computer[11].

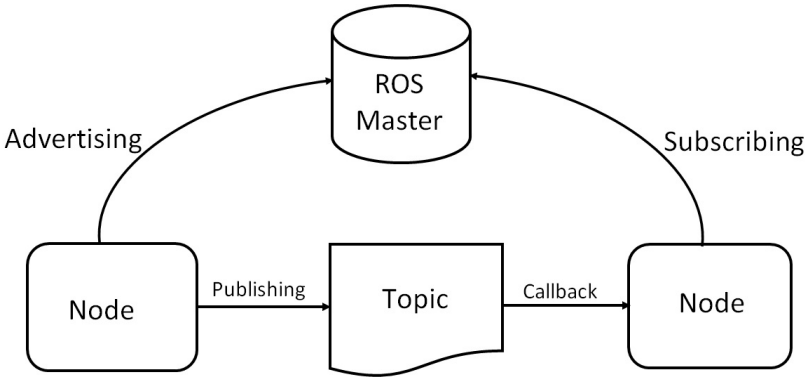


Figure 18: ROS graph model

5.1.1 Nodes

Nodes are independent processes running inside the ROS graph network, such as code gathering data from a sensor or code controlling the actuation of a motor. The nodes can be configured to be a publisher, subscriber or both at the same time. A node can publish messages to a topic which another node subscribes to, this creates a network of communication where nodes can publish and subscribe to different topics independent of each other.

5.1.2 Topics

Topics are unidirectional buses which nodes either subscribe to or publishes messages to. This means that topics decouple the source of the data which in turn result in nodes not being aware who they communicate with. A result of this nodes can subscribe to the relevant topics for their process or publish data to relevant topics independently of each other. A single topic can have multiple publishers and subscribers.

5.1.3 Messages

Messages are the data structure which the nodes communicate with the topics, these can be defined with standard data types such as strings, integers, floats ect.

5.2 Implementation of ROS - Embedded System

Much of the code used to set up ROS nodes, to transmit and to receive data from each node in ROS was based on the tutorial code from the ROS wiki page [10]. First a functioning ROS node code was added, and then the ros code was modified to fit the needs of each sensor, servo or motor code.

5.2.1 Initializing ROS nodes

The publisher code in ROS functions as a node that “publishes” data to a “topic” where other nodes like the subscriber node can “subscribed” to. ROS has its own library with data-types and functions that automate much of the ”transmitting” am ”receiver” code, you just have to include:

```
#include "ros/ros.h"
#include "std_msgs/String.h"
```

Listing 19: Ros libraries

To having a running node in the ROS environment, ROS needs to be initialized with the following code:

```
//init function
ros::init(argc,argv,"ServoPublisher");
//node handler
ros::NodeHandle n;
//Publisher
ros::Publisher chatter_pub = n.advertise<std_msgs::String> \\
("chatter", 1000);
// Subscriber
ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
//speed function
ros::Rate loop_rate(100);
```

Listing 20: Ros init code

First the init function that names the node are added, this function sets up parameters to the terminal with argc and, argv. Then "NodeHandle" object is added, this object handles process control, and does the actual initializing. The subscriber nodes need to use ros::Subscriber, while the publisher node needs to use ros::Publisher, or if the node needs both, it can publish and subscribe at the same time on different Topics, the code only needs to set this up correctly.

5.2.2 IMU and Encoder Publishers

Since the IMU only needed basic register edits in the start of the code. and after this basicly sent data one direction, the node needed just to function as a Publisher. The IMU code was added by pasting in the class code to the ROS cpp file. later this can be changed by including a header file instead.

```
while(ros::ok())
 {
 std_msgs::String msg;
 std::stringstream ss;
 ss << acceleration.GetAccelerationX() << std::endl;
 ss << acceleration.GetAccelerationY() <<std::endl;
 ss << acceleration.GetAccelerationZ() << std::endl;
 msg.data = ss.str();
 chatter_pub.publish(msg);
 ros::spinOnce();
 loop_rate.sleep();
}
```

Listing 21: Part of the IMU ros code

The IMU code in ROS is the same as in the basic C++ code, the difference is that the output is piped to a msg.data object that is published to the appropriate topic with a chatterpub.publish(msg) function instead.This is all done inside a While loop, that repeats while the condition Ros::ok() function returns 1. Extra data could also be sent with the ROS_INFO() function, like a count of how much data is sent, the date, or status reports.

The Encoder ROS code was similar to the IMU code where both of them functions as publishers. The difference was that the encoder only needed to publish one value.

```
while (ros::ok())
    {
        std_msgs::String msg;
        std::stringstream ss;
        ss << encoder.getPosition()*0.03<< std::endl;
        msg.data = ss.str();
        chatter_pub.publish(msg);
        ros::spinOnce();
        loop_rate.sleep();
    }
```

Listing 22: Part of the encoder ros code

5.2.3 Servo and Motor Subscribers

Both the servo and motor needed only to receive data and not send data to topics. Implementing the servo and motor code into a subscriber node seemed like the best solution.

The subscriber node functions almost the same way as the publisher node, except that it calls a function to receive the message instead.

```
ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
```

Listing 23: Servo main code

Main() performs the initialization of the ROS framework, and the init code for either the motor or servo. Inside the subscriber function is where the data is received. Every time a topic is changed, the code runs the function ChatterCallback() which receives the data. in this function is also where the motor and servo code functionality is added.

```
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
    std::string servo_pos = msg->data;
    if (servo_pos == "0"){
        servoA.setDutyCycle(1900000);
        servoB.setDutyCycle(1900000);
    }
   else if(servo_pos == "1" ){
        servoA.setDutyCycle(820000);
        servoB.setDutyCycle(820000);
    }
}
```

Listing 24: Servo subscriber function

The servos receive a string with either 1 or 0, by which the code responds by changing the duty-cycle accordingly. The ROS motor code functions almost the same way as the servo code, except in the project example code it receive an encoder position, and turns when the received encoder number is greater than 1000. This number can be changed, so the robot legs can move back and forth between two angels.

```
void chatterCallback(const std_msgs::Int32::ConstPtr& msg)
{
    int encoderPos = msg->data;
    if (encoderPos < 1000){
        control.toggleMotorPin(0);
    }
    else{
        control.toggleMotorPin(1);
    }
}
```

Listing 25: ROS motor subscriber function

5.2.4 Digital controller

Implementing PD controller code in ROS is done similarly as in chapter 4. Main difference on this occasion is that controller class and functions definitions are not placed in two different folders. Function definitions are now placed directly in class itself, so using scope operator is not necessary, since those function descriptions are no longer outside of class. Controller node must subscribe itself to encoder value in order to be able to calculate error. Therefore it is crucial to ensure those values are properly obtained. Subscriber function itself cannot obtain data, therefore callback function is required. If multiple subscriptions are required, callback functions also have to be multiplied, specially when two different types of data are being subscribed to.

```
void encoderCall_back(const std_msgs::Int32::ConstPtr& encoderPosition)
{
  std_msgs::Int32 encoder //Topic name
  encoder.data=encoderPosition->data;
  ROS_INFO("controller_recieves_encoder_data: %d", encoder_msg.data)
}
```

Listing 26: Encoder callback functions

Callback functions for encoder value is taking information sent by the node and storing it for use in controller equation. Main ROS function is where controller function is going to be called and where controller node is going to me created. Node itself will subscribe to encoder publisher, while it calculates control signal and then publishes it to a topic for other nodes to subscribe to.

Important to understand is what each function does and where it should be placed. All initialisation functions that are responsible with giving values to the terms, like gain, max/min output, initial error values. All of those must be called in main and run only once.

```
  while (ros::ok())
  {
    myPD.getControllValue(set_SetPoint, encoderPosition);
    myPD.Update();


    std_msgs::Int32 message;
    message.data = myPD.getControllValue;


    controller_pub.publish(message.data);   // publish
    ros::spinOnce();
    loop_rate.sleep();

   ++count;
  }
  return 0;
```

Listing 27: Calculating and publishing control signal value

Creating 'myPD' object allows for functions to be called upon in while loop. Functions that are present in while loop must always be in the loop since these need to be updated constantly. In the code, value given by the controller is set to be an integer, but in perfect conditions it would be desirable for value to be a float. Since float and integer values introduce considerable delay in Beaglebone, for intents of testing implementation of the controller, control value can be set to string.

5.3 Discussion

There are clear advantages for using ROS as the cross process communication framework, first it is easy to expand upon and very modular. Including additional sensor nodes or other servos and motors is easy to add, and is done quickly. the only downside is that one needs to use the make build system, which is another build tool to learn.

An alternative to ROS is to implement custom thread programming to handle the processes, this would result in using a lot of time on research and testing since the group members had limited prior thread programming knowledge and experience. Instead of using time learning thread programming, time was better spent on creating an initial framework in ROS, thus utilizing the time to develop lightweight code that would easily be implemented in ROS and be the groundwork for future development.

The code developed by a prior bachelor thesis group could not be implemented into ROS, since the code did not include all the components in the embedded system, and did not lend it self to be separated into individual nodes. Therefor new code was developed for all the components of the robot and with implementing the control and monitoring of the embedded system with ROS in mind.

Much of the specific ROS code written was not meant to be final code used for the control system of the robot, but rather a proof of concept of which one can build upon in the future. This would show how to set up a communication framework in ROS, and to show how the ROS communication could be implemented later when a proper control with PID controllers and walking algorithms are added to the project.

There are also disadvantages of using ROS in the embedded system. One thing is ROS having a lot of code overhead [12]. This was suspected of slowing down the Beaglebones response time. The group was not able to document this thoroughly due to not having access to the robot and conduct performance tests on the physical system.

6 Installing new servos

The already existing servos on the robot had to be replaced because one of the servos was not working. The brand of the already existing servos is Futaba S9254-type. Since these servos are not on the market anymore, a decision was made to replace them with a new type. In order to make the replacement as easy as possible, the goal was to find new servos with the exact specifications and dimension as those already installed.

A replacement servo was found after searching around on the internet. The new servos are named Turnigy SC-3404V and four servos where ordered from a website named HobbyKing.com. Table 2 compares the different specs between the servos.

Specs	Futaba S9254	Turnigy SC-3404V
Voltage	4.8V	6 $\sim$ 7.4V
Speed	0.06 sec/30°	6V: 0.05sec/60° 7.4V: 0.04sec/60°
Torque	3.38kg-cm	6V: 2.8kg-cm 7.4V: 3.6kg-cm
Motor	Coreless	Coreless
Weight	49g	33g
Geartype	Plastic	Metal

Table 2: Specs overview for servos

As shown in table 2, the new servo fulfills the requirements set by the already existing servo. Both torque and speed is stronger and the Turnigy weighs almost 33% less than the Futaba. A significant difference is the woltage required to upheld these specs. The new servos can be run on either 6V or 7.4, while the Futaba needed 4.8V.

One key difference between the two servos are the dimensions. The new servos are slightly smaller, with dimension for the new servo shown in figure 19 and table 2. That means the robot need new holders on the legs for the servos, since the new servos are to small to be attached.

A	34 mm
B	35 mm
C	29 mm
D	15 mm
E	49 mm
F	19 mm

Table 3: Dimensions of the servo

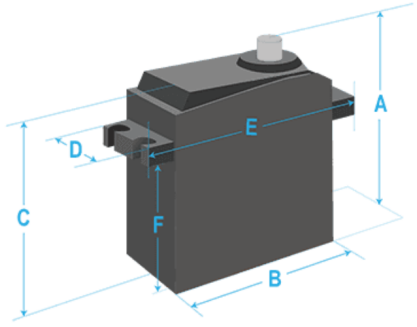


Figure 19: Different lengths on servo

6.1 Designing new holders for servos

The current holders for the servos are represented in figure 20 and 21. There are four holders, made out of aluminum. Putting in new holders does not require any redesign of the holder it self. The only change necessary is the design of the servo placement, which is a cut-out on the plate, making it possible to put the servo inside and through it.

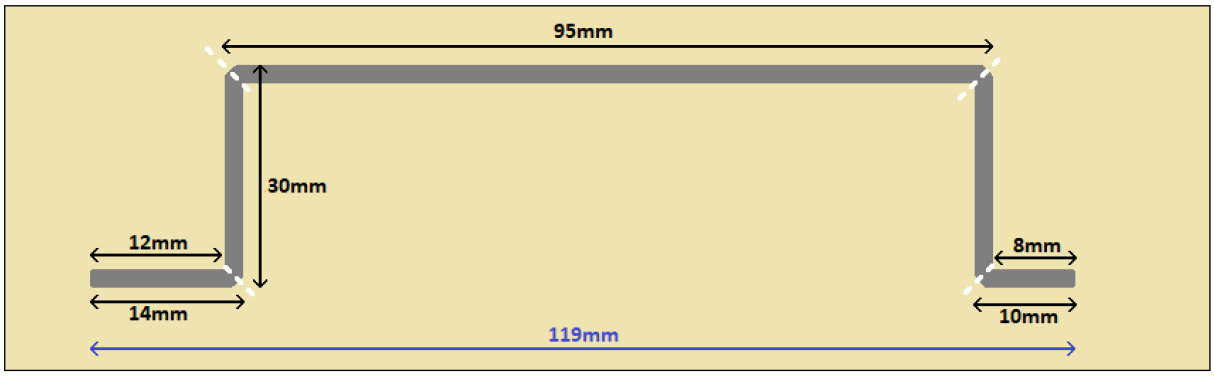


Figure 20: Current servo bracket (side view)

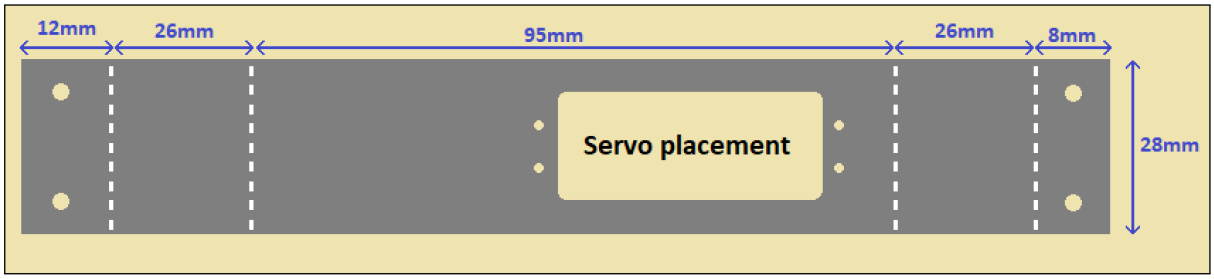
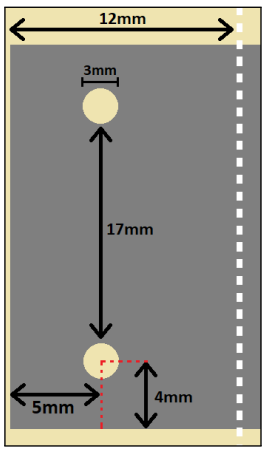
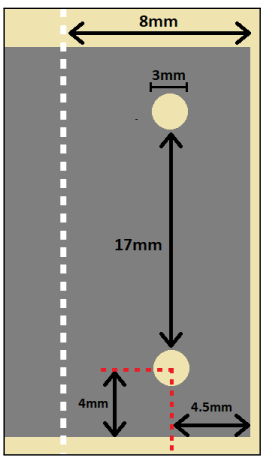


Figure 21: Current servo bracket (top view)



(a) Left side



(b) Right side

Figure 22: Dimensions of screw holes on the left and right side

In order to design a new servo holder, a series of measurements was conducted in order to obtain precise dimension of the Turnigy-servos. These measurements are shows in figure 23. This figure is the exact same figure as figure 21, except the measurement for the new servo placement has replaced the dimension of the metal plate.

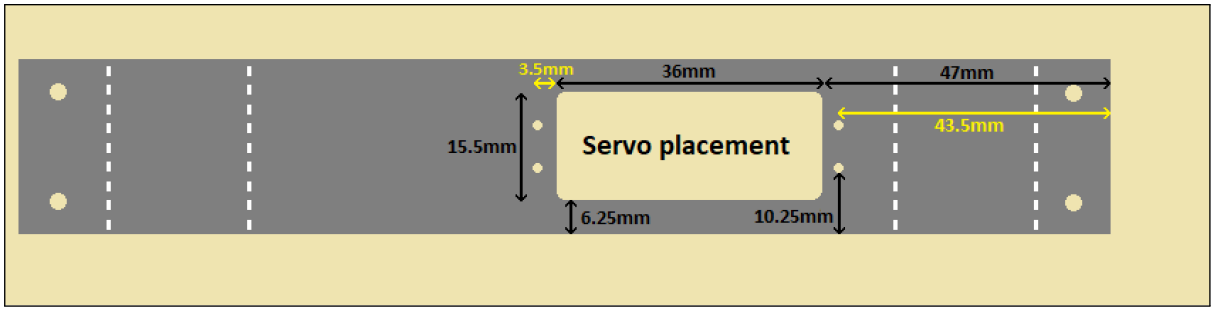


Figure 23: Dimension, lengths and widths of new servo holder.

Following these drawings and measurements should produce four new sets of holders required for the new servos. The screws used to connect the former servos and holder are compatible with this new design and are not subject to replacement.

7 System identification

In order to achieve implementation of controller and a successful process control, it is an imperative to come up with good equation describing robots behaviour. Model of outer leg will be presented together with analysis of its mass and gravity centers. Since legs are different in terms of its size and weight, two different dynamics have to be presented. Due to the unforeseen change of circumstances that occurred through the process, only model of one leg will be presented. It is worth mentioning the same process and principles can be applied to the other leg, with a change in certain physical parameters like weight, length, center of mass etc.

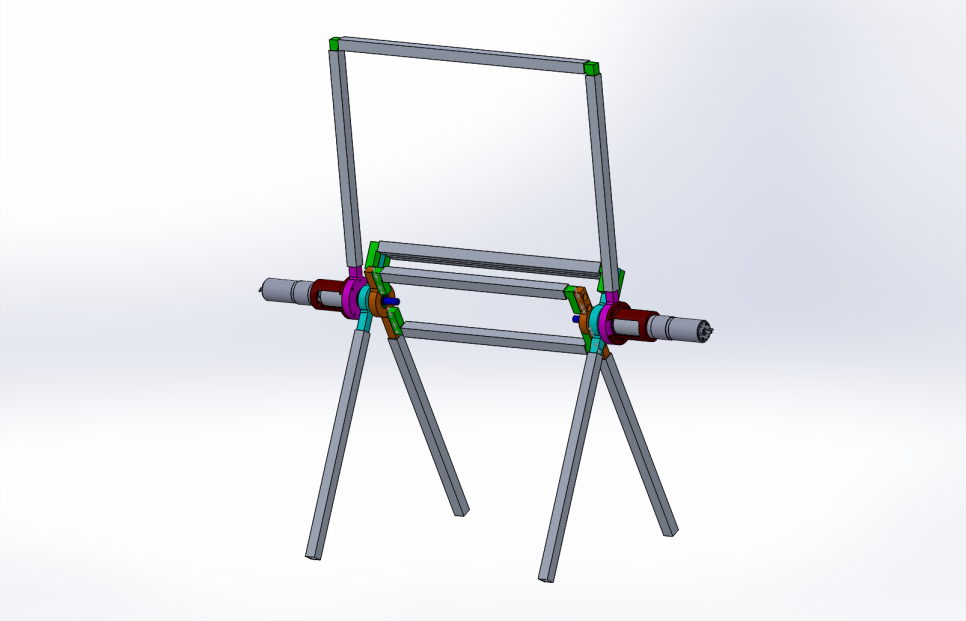


Figure 24: Robot portrayal

7.1 Method

Modeling the leg was based on the dynamics of the swing using differential equations, which are further going to be exploited in obtaining Laplace transformed equation in s-domain. Before any efforts are given in deriving equations, it must be noted that observing robot itself will provide valuable information regarding its dynamics. While observing movement, placement and actuation of robots legs it is concluded that legs can be perceived as pendulum. Further reviewing of physical attributes led to the conclusion that robot legs due to mechanical behaviour and construction can best be descried as a physical pendulum.

7.2 Actuator

Electrical motors are converting electrical energy into mechanical thus executing tasks that are set upon. If it is desired to properly control certain aspect of DC motor performance, inner workings and physics have to be properly understood. In this particular case where it is an imperative to obtain relations between input voltage/current and output torque.

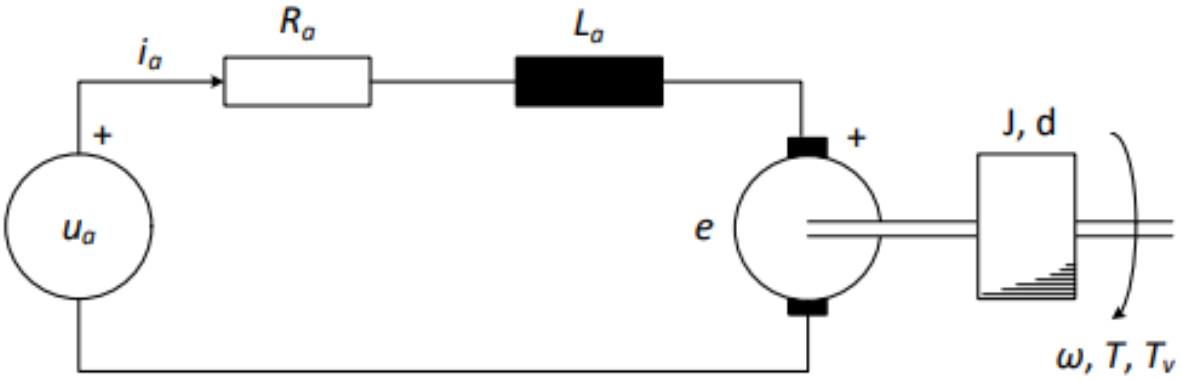


Figure 25: Motor scheme

Kirchoffs second law states:

$$U_a - U_{Ra} - U_{La} - e = 0 \tag{3}$$

Further more:

$$U_a - Ri_a - L\frac{di_a}{dt} - Ke\omega = 0 \tag{4}$$

In instance of DC motor there are two aspects that have to be addressed properly. As mentioned earlier DC motors convert electrical energy into mechanical, therefore moment law has to be addressed. These two aspects when combined will make it possible to derive transfer functions addressing different relations between different inputs and outputs. Moment balance equation states:

$$J_m\frac{d\omega}{dt} = K_t i - T_L - d_m\omega \tag{5}$$

Where:

ω : Angular velocity
J_m : Motor moment of inertia
d_m : Motor damping constant
T_L : External disturbance/load torque
K_e : Voltage constant
K_t : Torque constant

All of the different values that were mentioned can be found in producers datasheet for that particular motor variant. Some datasheets may or may not give information regarding only K_t and not K_e. Those terms in practice are dependant on armature current and wear and tear that happens over time. If there is no effect loss in motor coil than those two constants can be set equal to each other. That is something that is rarely the case but is worth mentioning especially when trying to simplify the motor model.

Torque generated directly by motor itself without gearbox can be presented:

$$T = K_T i \tag{6}$$

This is a very important relation since it shows that torque is dependant on armature current. In the case of this project and motors used on biped robot, it can be said that motors are underpowered in regard to robot. In Master thesis *Stable Gaits for an Underactuated Compass Biped Robot with a Torso*[28], chapter 4, it was concluded that torque value over two performed steps was 2.06 Nm, with its maximal torque of 2.6Nm. For simplicity and phase margin reasons through the project maximum value of 2.6Nm is rounded to 3Nm.

7.2.1 Gearhead

From datasheet and previous experiments executed on robot, it can be concluded that actuators tasked of driving robot legs are greatly underpowered. Therefore it is very important to include a gearbox which is going to amplify torque produced by the motor. Gearbox found installed on motor is Maxon 1 to 6 gearhead.

$$T = 6K_T i \tag{7}$$

7.3 Application of Newtons second law to the pendulum

Application of Newtons second law which states that $F{=}ma$ cannot be used directly in the case of physical pendulum, as it would be in a case of mathematical pendulum. Reason for such is that robot legs are rigid object which swing freely on certain pivot point. Mass is distributed evenly along the length of the pendulum, thus different places on the pendulum have different speeds. Their angular velocity is the same through the motion. Therefore instead of using typical equation for Newtons second law as mentioned in the text above, an alternative modified equation shall be used as one presented:

$$\sum \tau = I\alpha \tag{8}$$

Equation (8) dictates that *torque* equals *moment of inertia* times *angular acceleration*. Further more torque can be represented:

$$\sum \tau = -mgsin\theta d = I\alpha \tag{9}$$

In order to derive motion equation torque will be set equal to zero value for the reason of simplicity in deriving.

$$I\alpha + mgsin\theta d = 0 \tag{10}$$

$$\alpha + \frac{mgsin\theta d}{I} = 0 \tag{11}$$

Dynamics of the pendulum that have been derived do not take into consideration any damping in the system. If realistic simulation of the systems dynamics is desired, one must include damping. There are multiple types of damping which are encountered in such systems. Two that are most relevant are damping caused by the friction at the pivot point, and that one caused by objects movement through some fluid. In this case fluid being air. Following equation takes into account both viscous friction (air) and dry friction (motor shaft), making it a much closer approximation to the practical pendulum.

The basic characteristics of viscous friction are well defined considering that, for low velocities, the friction force is constant in modulus and always acts in the opposite direction to the velocity. The differential equation of the oscillatory motion considering viscous damping is given by equation (12), where $b\dot{\theta}$ is the damping term given by the friction between the pendulum and the air, and $D|\dot{\theta}|\dot{\theta}$ is term given by mechanical friction.

$$\alpha + b\dot{\theta} + D|\dot{\theta}|\dot{\theta} + \frac{mgd}{I}sin\theta = 0 \tag{12}$$

Where:

$$\omega^2 = \frac{mgd}{I}$$

When attempting to simulate dynamics of the system, equation must first be linearised. Therefore upon linearization of the dry friction element the same will be equal to zero. Reason for performing such action is an effort of simplifying simulation as much as possible.

Dry friction results in a nonlinearity. With it, the system acquires a non-smooth, discontinuous character. If the coefficient of dry friction is sufficiently small, the oscillating body slides harmonically and its velocity is zero only for the instants at which the direction of motion reverses. Another reason why viscous friction is going to be equal to zero or at least can be approximated to zero, is because of the D term which is extremely small, thus making the approximation even more reasonable.

Since θ is representation of angle/position it can be derived further in order to obtain angular acceleration. In addition $sin\theta$ has to be linearized using Taylor series. Maxon motors which are used in actuation of the robot legs are lacking strength in order to provide greater angles and position. It is capable of delivering torque not greater than 3Nm. Practical experiments have showed that under normal conditions the same motors are not able to drive motor legs to angles greater than 20 degrees. Following statements can be expressed:

$$\alpha = \ddot{\theta}$$
$$sin\theta = \theta - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} - \frac{\theta^7}{7!} + ... \approx \theta$$

If same statements are substituted, following equations are derived:

$$\ddot{\theta} + b\dot{\theta} + \omega^2\theta = 0 \tag{13}$$

7.4 Performing calculations

7.4.1 Differential function

Taken into consideration everything that happened, and fact that physical examination and measuring of the legs was not possible for extended period of time, values used while performing calculations are approximated to the nearest round number. Following values are obtained and used further in assignment.

(m) Mass = 2000g

(L) Length = 0.75m

(d) Mass center distance from pivot point = 0.2 m

Most important component to derive is moment of inertia. Due to the fact that center of mass is lifted up 20 centimeter from the center point, parallel axis theorem has to be applied in order to derive proper moment of inertia.

$$I = I_{cm} + md^2 \tag{14}$$

Inserting different values and substituting certain equation aspects gives:

$$I = \frac{1}{12}mL^2 + m(\frac{1}{5}L)^2 \tag{15}$$
$$I = \frac{37}{300}mL^2 = 0.13875Nmkg^2$$

$$\omega^2 = \frac{mgd}{I} = 28.28$$

Damping constant describing viscous friction acting on the pendulum was obtained from bachelor rapport that was completed a year before. Damping constant was chosen to be:

$$b = 0.2130$$

With all calculations performed, differential equation describing pendulum dynamics is as follows:

$$\ddot{\theta} + 0.2130\dot{\theta} + 28.28\theta = 0 \tag{16}$$

Current differential equation has to be run and inspected in MATLAB. Simple simulink model of the pendulum is created. Plotting pendulum behaviour was done in a way that pendulum is lifted to a certain angle and than released, causing it to swing until any motion has died out due to damping. Starting value of the first integration block which is integrating $\ddot{\theta}$ to $\dot{\theta}$ is set to value describing angle one desires to release pendulum from. In this case that is 1.04 radians.

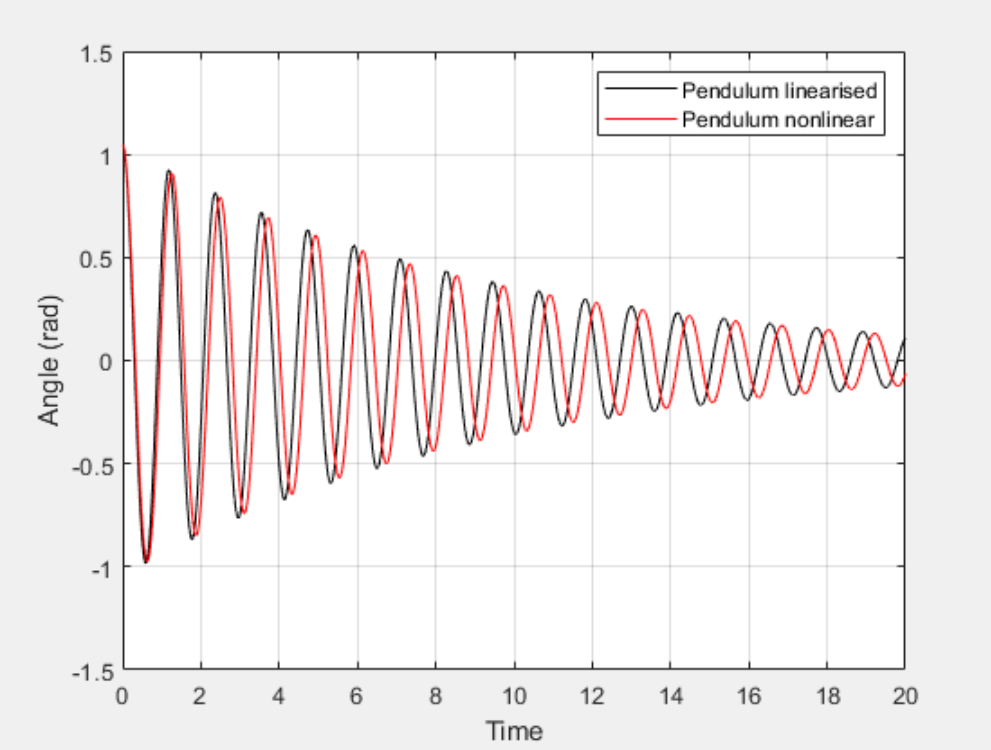


Figure 26: Pendulum swinging motion until settling

Upon inspection it is apparent that desired behaviour has been obtained. Satisfactory pendulum behaviour was accomplished for both systems. Pendulum starts at an angle and swings until all energy has left the system due to damping.
Leg dynamics may change later due to the fact that IMU stations are going to be connected to the legs, together with the cabling. Effect of viscous friction and its constant can also be noticed while observing oscillations. Dry friction and viscous friction introduce

linear and exponential damping of the system. Since dry friction is linearised and thus equal to zero, the system is left only with the viscous friction which can be seen on the graph.

7.4.2 Pendulum transfer function

After differential function was obtained it must be converted from time domain to s-domain using Laplace transform. Equation (13) is set equal to zero for deriving purposes, making it easier to manipulate. In reality zero value is going to be substituted with torque applied from the motor, also acting as an input for the pendulum. First equation has to be transformed using Laplace:

$$\mathcal{L}\{\ddot{\theta} + b\dot{\theta} + \omega^2\theta = T\} \tag{17}$$

$$s^2\theta(s) + bs\theta(s) + \omega^2\theta(s) = T(s)$$

Pendulum transfer function:

$$\frac{\theta(s)}{T(s)} = \frac{1}{s^2 + 0.2130s + 28.28}\frac{rad}{Nm} \tag{18}$$

7.4.3 Actuator transfer function

As mentioned earlier actuator converts electrical energy into mechanical, therefore combination of Kirchoff's second law and moment law has to be performed. Multiple transfer functions that can be derived depending on what aspect of motor performance one desires to exploit. Since leg actuators take current as input either through certain motor controller or some other device, it is important to obtain transfer function which takes current as an input and torque as output. Performing Laplace transform on equation (7) will give the desired transfer function:

$$\frac{T(s)}{I(s)} = 6K_t \tag{19}$$

7.5 Discussion

MATLAB and Simulink was used in order to execute simulations on how system will behave in practice. Exploiting MATLAB's strength in automation simulations and calculation is reason why it is recommended for such tasks. Obviously due to the many challenges faced in the semester, physical and practical examination was not possible. Accordingly it was not possible to determine if simulations are good representation of the real world model. Work of groups before and with the help of supervisor it was possible to compare those simulations to the work done prior and thus determine if simulation is good enough to represent that system.

8 Process control

Multiple factors have to be taken into account when trying to derive control strategy for this particular robot. Starting from the stability of the construction itself. The most prevailing fact is that robot as it is right now is inherently unstable and is not actuated with proper equipment capable of delivering optimal torque to the legs.
It is expected that closed loop controller will be able to stabilize and control biped through couple of gait steps, but it is to expect that small numerical deviation due to the legs hitting the ground, will quickly build up over time making gait unstable and unable to contain.

Therefore decision was made that PD feedback controller will be best to utilize. Reason behind such thinking is as mentioned fact that sustainable gait is not possible, and fact that actuators tasked of achieving the gait are not strong enough to do so. Therefore controller itself will never be able to achieve full elimination of steady state error, thus integral term tasked of eliminating the error is not desired since its purpose will never be fully achieved.

8.1 Simulation

Before transfer functions are used in simulations, decision was made to connect in series only PD controller and pendulum transfer function, and to use saturation block which would simulate motor limitation in torque. Reason for doing so is based upon simplicity. later motor transfer function would be used to compare effects of torque limitation.

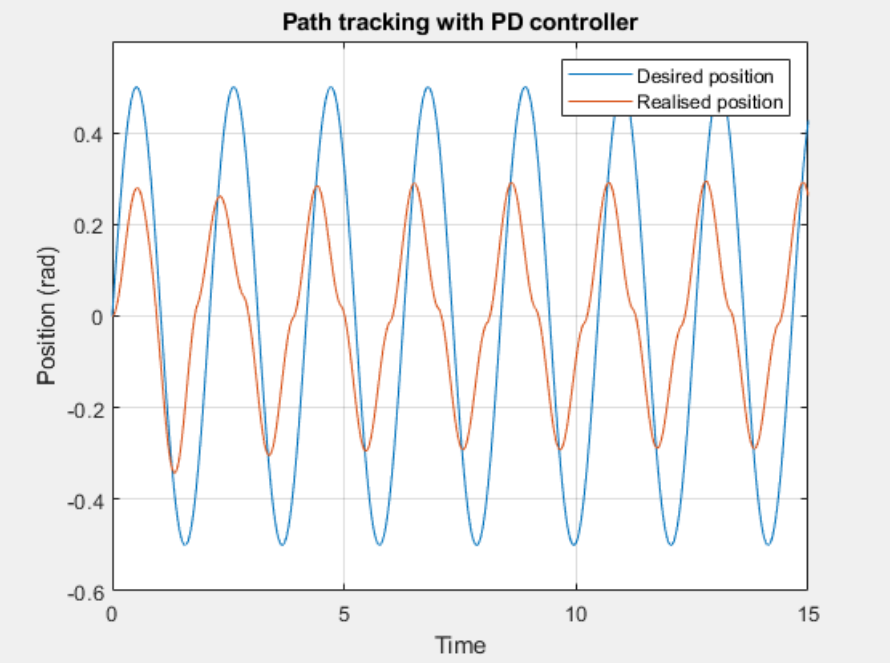


Figure 27: Tracking desired angle

Simple sinus function is used as desired path legs should follow. In the first simulations sinus signal with amplitude of 0.5 and frequency of 3 rad/s is set as a desired path. Controller parameters are set to Kp=10 and Kd=1. Actuators are unable to follow desired path in addition to phase shift. Increasing controller parameters to abnormal values will hit the saturation point causing minimal to no improvement in performance.

8.1.1 Natural frequency

Pendulum transfer function is a second order function with complex poles. Therefore such function will have noticeable frequency at which body resonates. It was decided to use body's resonant frequency to try getting better performance from the actuator, in regards to path tracking and realising higher amplitude. Bode plot of pendulum function gives following graph:

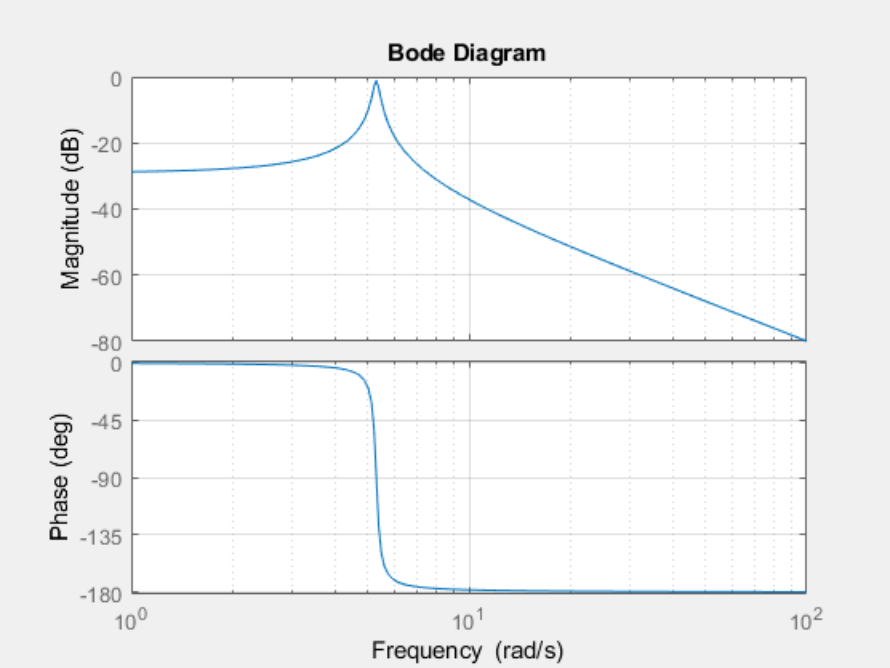


Figure 28: Bode plot based on pendulum transfer function

Magnitude plot shows resonance peak to be at frequency of 5.32 rad/s. If this information is further exploited by setting this frequency in the sinus wave, than it is to expect that motor achieves greater angles with ease.

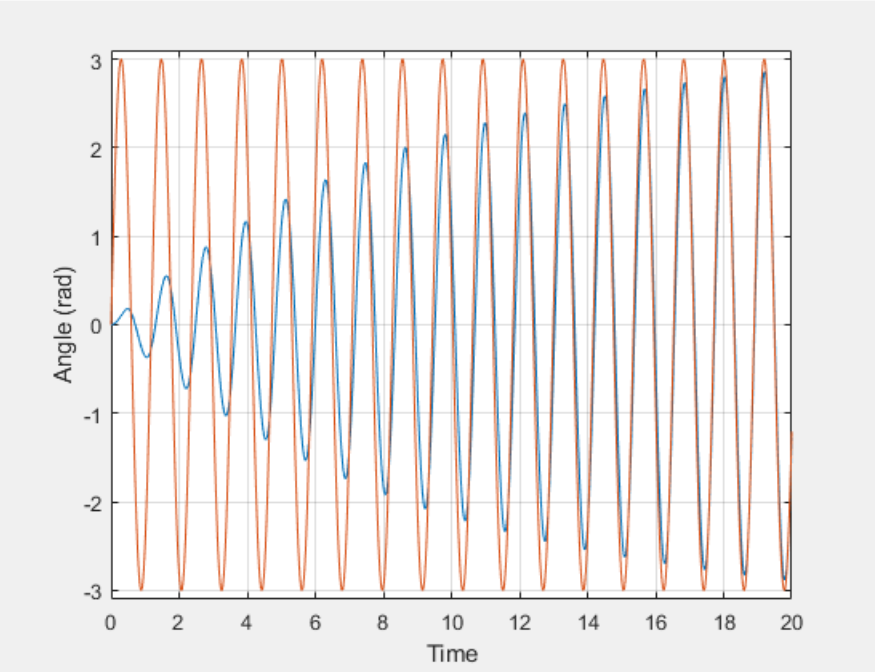


Figure 29: Actuator response to input resonance frequency

Example of running resonance frequency as motor input is presented in the graph. When running at resonant frequency, motors have no problems reaching desired angles. In this simulation angle of 170 degrees is achieved, with the usage of pendulum resonant frequency. By exploiting this fact, actuators are able to lift the legs almost vertically. Of course this goes to prove how one can exploit body's natural frequency in helping underpowered actuators, but does not imply that such method should be used. Running actuators at resonant frequency will cause excessive oscillations in the system and will result in grater wear and tear on components.

8.2 Revised simulation

With models for motor controller, motor, and pendulum obtained, original simulation can be revised and run again. The result is as expected earlier, with PD controller maintaining stability and offering sufficient path tracking. Area where it lacks the most is in amplitude or delivering desired angle. There are multiple reasons for such behavior. One is that PD controller cannot eliminate error, and the second one is the motor itself which is not capable of delivering grater angles.

In revised simulations motor transfer function was correctly modeled in a way that input signal is limited to the value which will give 3Nm as output. Controller parameters which are giving best response are Kp=60 and Kd=18. Everything above that does not give any improvement and in practical trials might lead to instability.

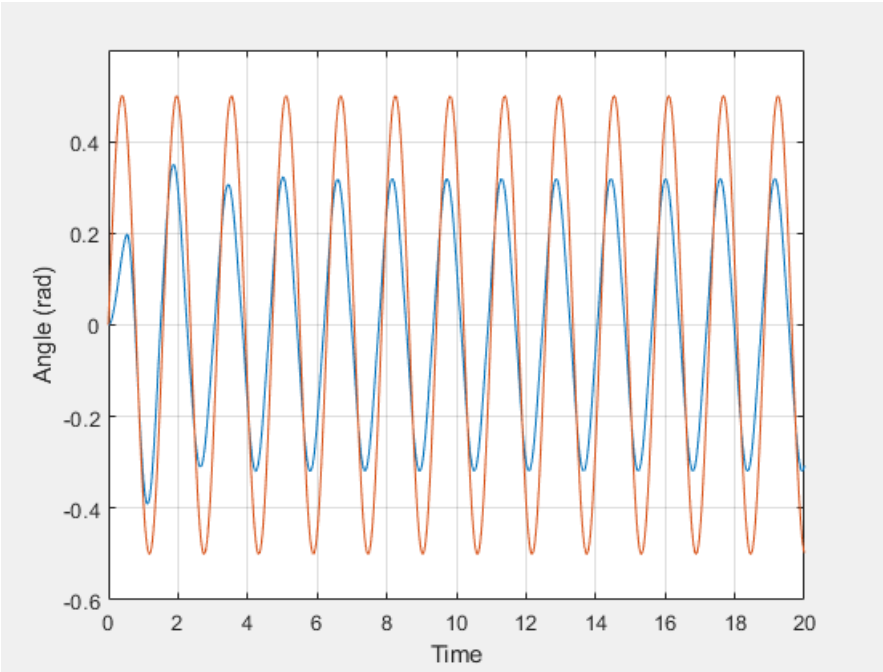


Figure 30: Revised simulation

8.3 Feedforward control

Feedback control is standard procedure used for controlling and stabilising any given system. In attempt of trying to improve stability and control quality, feedforward controller in addition with feedback controller can produce certain aspects worth investigating. Knowing that change in setpoint is bound to happen at some particular time can indicate that feedforwar control should be investigated.
In the case of controlling pendulum it is known that setpoint change in angle of the motor shaft will happen, therefore implementing reference point feedforward controller could be very useful if it is possible to derive. In order to derive reference point feedforward controller, process must first be inverted:

$$H_{ff} = \frac{1}{h_{pros}} \tag{20}$$

This causes some immediate problems. First one is that process dynamics depicted in the transfer function are never fully known thus obtaining perfect transfer function is very rare. Second is that by inverting the process it is to expect to have poles in the right half plan and to get time delay to be positive, all of which will cause process to be unstable. When inverting the process function it is obvious that such function is not realisable as a controller function.

$$\frac{1}{h_{pros}} = s^2 + 0.2130s + 28.28 \tag{21}$$

Such function is not realisable, reason being fact that system itself has more zeros than poles. Failure to obtain realisable function with use of lead - lag units has caused need to use direct, by hand, derivation in order to achieve some type of success with the feedforward controller. Referanse signal used is the sinus signal with amplitude 0.2 and frequency of 9 rad/s. Deriving by hand three different functions are derived:

$$y(t) = 5.65sin(9t)$$

$$\dot{y}(t) = 10.83cos(9t)$$

$$\ddot{y}(t) = -97.48sin(9t)$$

Sinus functions are summed together and added into the control loop. This method is of course unorthodox for deriving feedforward controller but is essential in order to acquire functioning feedforward.

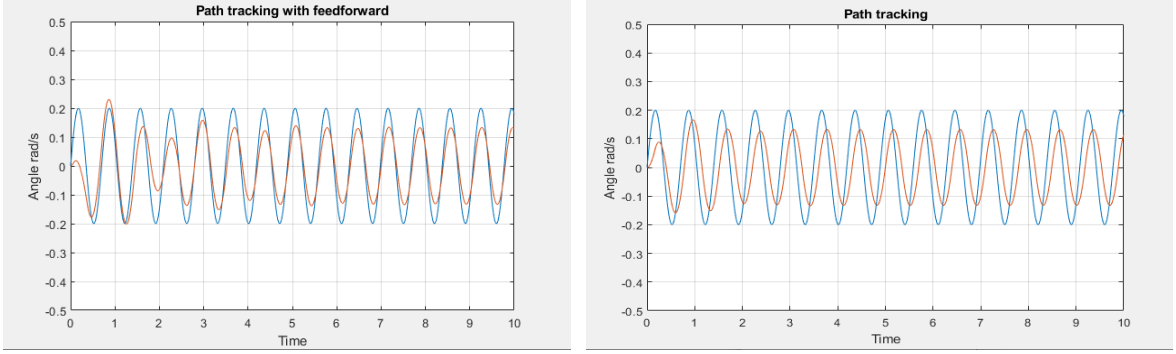


Figure 31: Control loop with feedforward

Motor alone before feedforward controller was connected, was not capable of following designated path. Problems with error and phase shift are apparent. Despite the fact that addition of an extra controller does not eliminate error, it reduces it and it corrects for the phase shift. Easier way of looking at it is if two plots are show each representing deviation from the desired path.

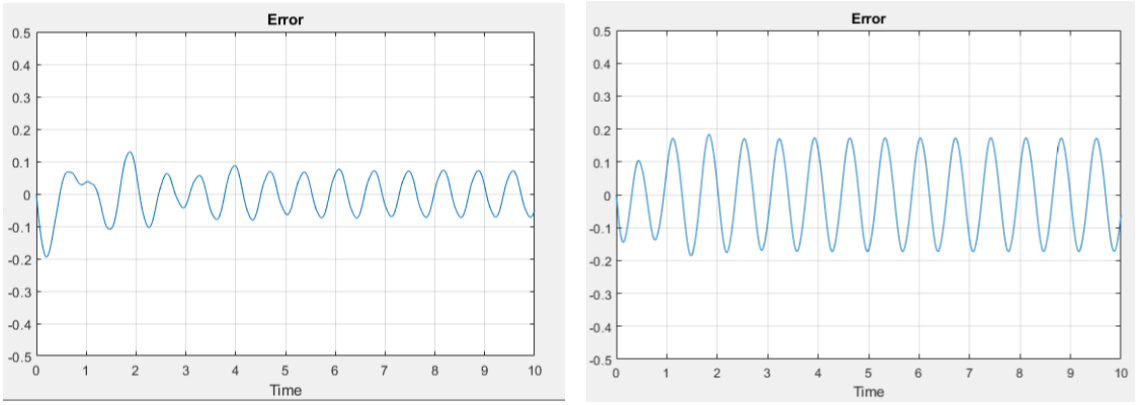


Figure 32: Error value with and without feedforward controller

It can be noticed that feedforward improves system control and helps feedback controller in reducing the error and phase shift. Due to the limitations system has inherently, different frequencies of the setpoint signal will give different degree of success when trying to control the system. Therefore it cannot be said that feedforward controller helps in every instance.

8.4 Discretization

Implementation of controller and process in single-board computer like Beaglebone will make process control possible. To achieve that task all systems must be discretized first including the controller. There are multiple methods one can exploit to achieve such task. Method chosen in this instance is trapezoidal approximation method also known as Tustin's method. Beaglebone Black offers readings at sampling frequency of 100Hz, which is sufficient for ensuring system stability. As observed on bode plots of the closed loop system, figure (28), body operates at low frequency, therefore with sampling frequency provided by processor, problems caused by sampling are not to be expected. After discretization of motor and pendulum function following equation have been obtained:

$$H_{sys}(z) = \frac{1.068 \cdot 10^{-5} z^2 + 2.135 \cdot 10^{-5} z + 1.068 \cdot 10^{-5}}{z^2 - 1.995 z + 0.9979} \tag{22}$$

For implementation on single-board computer like Bealgebone, PD controller has to be discretized. Standard PID algorhitm states:

$$u(t) = K_p \cdot (\frac{1}{T_i} \int_0^t e(t)dt + T_d \frac{de(t)}{dt}) \tag{23}$$

This equation will be further exploited in deriving digital controller. In book *Process Dynamics and Control* book[34], discretization of position and velocity algorithm is explained. Due to the previous positive experiences with the velocity algorithm it was chosen as a equation that will be used in the controller. But in order to do so, first position equations have to be derived:

$$p_k = p + K[e_k + \frac{\Delta t}{\tau_I} \cdot \sum_{j=1}^{k} e_j + \frac{\tau_D}{\Delta t}(e_k - e_{k-1})] \tag{24}$$

$$p_{k-1} = p + K[e_{k-1} + \frac{\Delta t}{\tau_I} \cdot \sum_{j=1}^{k-1} e_j + \frac{\tau_D}{\Delta t}(e_{k-1} - e_{k-2})] \tag{25}$$

If equation (24) is subtracted from equation (25) it gives digital PID controller algorithm:

$$\Delta p_k = p_k - p_{k-1} = Kc[(e_k - e_{k-1}) + \frac{\Delta t}{\tau_I} e_k + \frac{\tau_D}{\Delta t}(e_k - 2e_{k-1} + e_{k-2})] \tag{26}$$

Final version of the equation (26):

$$p_k = p_{k-1} + Kp(e_k - e_{k-1}) + Ki \cdot e_k + Kd(e_k - 2e_{k-1} + e_{k-2}) \tag{27}$$

Equation shows how backwards approximation is performed. Previous values of error and control signal are required for control signal calculation to be performed. For better understanding of the controller code described in chapter 4, parameters used in controller class can be related to equation subjects:

$$\begin{gathered} p_k = \text{controlSignal} \\ p_{k-1} = \text{previous controlSignal} \\ e_{k-1} = \text{previous error 1} \\ e_{k-2} = \text{previous error 2} \\ e_k = \text{error} \end{gathered}$$

Since integration term will not be used, Ki term will be set to zero. Equation uses backwards approximation therefore two previous error values are needed in calculations, which is not possible to obtain in the beginning of the calculations. That is reason why initialisation function has to be performed prior to the rest of the code.

8.5 Discussion

Determining strategy on how system will be controlled and which controller to use, was done based upon fact that robot is underactuated and underpowered. Group came to conclusion that due to fact that hip actuators are not strong enough, using controller with integral term would be redundant. Actuators found on the robot don't have the power needed to execute thing commanded by the potential PID controller. Therefore decision was made to use Proportional-Derivative controller.

Difficulties were met while trying to determine how to implement feedforward controller in simulated model due to the fact that function obtained with inverting the system could not be used. In addition to that, efforts in using lead lag components proved itself not to be helping. In later calculations feedforward controller was obtained and it proved that including it in control loop is helpful.

Discretization was performed using sampling rate provided by ROS loop rate in Beaglebone. Same Simulink model used for performing simulations was discretized and proved that sampling is fast enough, since system dynamics and stability does not change noticeably. Controller algorithm was area where difficulties started to appear since many different methods are available. Choosing one with desirable functionality and simplicity was an idea that was used as guideline.

9 Discussion

From the start of this bachelor it was agreed upon that different group members would work on separate parts, but still learn from each other by working towards adding these parts together into a complete system. The group members first worked together in the start with all the common tasks like learning Github, Linux and C++. And later spilt into two groups. The automation group that worked with controllers and control theory. And the Instrumentation/electronics group that worked with Improving the wiring and coding the different modules into ROS. Because of the outbreak of the Corona Virus, much of cooperation between the groups did not work properly during this time, mostly because of the groups being forced to work at home, and not having access to the lab, and a place to work together.

Since the bachelor group consisted of engineering students with different expertise, and the robot had problems in multiple areas, it was easy to direct problems to each member. Working with embedded systems and reworking the wiring was done by electronics and instrumentation students , and the control systems and ROS part given to the automation students. Even though the two groups worked with different problems, many components and platforms was worked on together. Both groups worked with code that ultimately was going to run on a Beaglebone.

The redesign of the robots wiring system proved to be a valuable decision; this gave a detailed overview of the wiring of the robot at an early stage of the project. The lessons learned when troubleshooting and improving the wire system was used throughout the development of the connector circuitry and the development of the embedded system. The group acknowledges that the wire system is not fully complete and will need some revisions in the future, but it presents a good groundwork and enables testing and further software development to be performed.

Development of completely new code for the embedded system was a big task and proved to be time consuming, but developing code with clear ground rules and with ROS implementation in mind contributed to greater learning and valuable knowledge for the group members responsible for the coding. The decision to have a focus on documenting the code in a centralized GitHub repository proved to be very important when the group was forced to individually work from home due to the outbreak of coronavirus. The Github repository will also act as resource for future projects on the bipedal robot which was the long term goal of the documentation efforts.

Upon research of prior bachelor and master theses, it was concluded that robot is inherently unstable. So much so that it is highly unrealistic to expect that such system could be capable of performing gait over multiple steps. Therefore group decided not to focus on making the robot walk but instead performing research that would make future work easier. Searching through old lectures in mathematics and control theory proved itself to be an effective decision. Much of the material used for understanding and deriving initial equations of the simulated model was obtained from old Mathematics/Physics lectures describing pendulums. Understanding syntax and algorithm of the digital controller is something that automation engineers are familiar to, through lectures and exercises. Area which was considerably difficult was understanding C++ code used to program digital controllers and how to implement the same code in ROS. Such difficulties were also magnified by the fact that each group member had to work individually from home.

10 Conclusion

The primary goal of this bachelor's thesis was to develop a robust foundation for further development on this project. This meant having wiring that is simple enough for new students to understand, but also modular enough to not impede future changes. The implementation of a new wiring system, which incorporates ethernet cables and PCBs for connections greatly improved the modularity of the system and created an easy to understand overview of the robots wiring. The main connector board also reduces the chances of incorrect wiring that could lead to hardware damage. The new wiring system therefore is an important part of making the robot more robust and easier to develop in the future.

The goal for the embedded part of the project was to develop code that was easy to use, but also inherently good enough for further development. Even though the group met many hindrances during development the group ended up with fully functional C++ code for each sensor, servo, and motor. This code was short, easy to understand, and simple enough to perform further development. Documentation of the code in a GitHub repository will also aid the future development on the robot.

The secondary goal of this bachelor's thesis was to preform research into implementation of a PD – controller on the bipedal robot with the aim of the robot performing one or two steps. The research presented in this thesis pertaining to the system identification, control strategy and development of a PD controller code, grant an insight into how the bipedal robot could be controlled with a PD controller implemented in the embedded system. The research also presents a suggestion for how the PD controller would be implemented in ROS.

In conclusion the current bipedal robot prototype has a functional embedded system with new modular wiring and the theoretical groundwork for the implementation of a PD controller for the robot is completed. The current system fulfills the main goals set for this thesis, but the group acknowledges that the limited testing of the physical system hinders the group to conclude that the control system of the robot is fully completed and ready for further development without modification uncovered by future testing.

11 Future Work

11.1 Circuit design

This section presents suggestions for future work on the circuit design of the bipedal robot.

11.1.1 Main Connector board

There are multiple things that would be favorable to improve on the main board. First removing the need to supply 3.3 volts from an external supply to the IMUs, and replace this board trace with one to Beaglebones 3.3 volt pin. This is possible because all three IMUs do not draw more than 13.8 mA [22] which is low enough for Beaglebones 3.3 Pin.

Adding a logic converter from 5 volts to 3.3 Volts to the encoder inputs in the main board, or the encoder connector board, would fix the voltage logic error the group created. This should be done immediately, so that Beaglebone doesn't damage its pins.

Adding headers pins on the top of the main board would make it possible to connect other shields, if needed, and could also be useful in debugging or other times, access to Beaglebones pins are needed.

11.1.2 IMU connector board

Designing a new IMU connector board which directly incorporates the LSM9DS1 inertial measurement unit chip in its circuitry will eliminate use of the Sparkfun breakout board currently connected to the IMU connector board. This could greatly reduce the overall size of the IMU connector board. Research into alternative IMU chips should be performed before incorporating the current IMU chip in a custom connector circuit.

11.1.3 Power supply board

A Larger project for the Robot could be to create a power-supply board to the main board. This power-supply would feed it 48Volts for the motors, 7.4 Volts for the servos, and 5 Volts for Beaglebone and the encoders. Adding a board like this could be a great challenge for a bachelor group, and would decrease the number of cables going to the robot.

11.2 Embedded system

This section presents suggestions for future work on the embedded system of the bipedal robot.

11.2.1 Beaglebone

Further performance testing of the Beaglebone Black should be performed, to verify that the beaglebone can run the completed embedded system without crashes or performance issues over time. The group recommends researching alternative beagleboards for example the Beaglebone AI or Beaglebone Blue if this is the case.

11.2.2 Code improvements

The code developed for the system components in the embedded system are meant to be a basis for future work, the code shows the most important elements of how to program the system components. Therefore, future development of the embedded code should focus on enabling more functionality and configurability to the system and simplify the use of the code through custom headers, use of pointers, references ect.

11.2.3 PRU implementation

Research into the capabilities of the PRU-ICSS units on the Beaglebone should be done in order to fully optimize the response time of the embedded system and decrease the amount of processing overhead of the Beaglebones AM3358 processor.

11.3 ROS improvements

To read data from the sensors, it is possible use ROS's Rqt-plot, a graphical plotter included in the full version of ROS. Rqt can run on a separate computers ROS and receive data from another ROS system on Beaglebone through Ethernet. Rqt can then plot all the data graphically in real time on a computer screen.

To reduce the Beaglebones processor load from the different ROS nodes, it would help to change much of the ROS code towards using pointers and references instead. This reduces the memory usage and load on the system, and makes the code run faster.

Much of the code in ROS is written as a proof of concept. The code need much rework and another ROS setup needs to be created, since the current setup was lost when the group shorted the Beaglebone. To run ROS, Ubuntu should also be flashed to a memory card and installed on the Beaglebone. Tutorials on how to install Ubuntu on Beaglebone black can be found here. [14].

11.4 System control

11.4.1 Identification

Specially problematic when establishing control of the system, was how to design and simulate feed forward controller. System itself isn't realisable when it is inverted, meaning that approximation of such system must be derived. Deriving approximation itself was not giving desirable results. When the problem was solved, simulations proved that such implementation has an effect on system stability. Future work in this area could be based on assessing benefits provided by such controller, what are benefits precisely and are they practically going to be realisable in a desired time period.
Insuring that performed MATLAB/Simulink simulations are done in parallel with physical trials is something that was not possible to do. That is why ensuring that mathematical model obtained does describe that found in practice is something which is an imperative.

11.4.2 Digital controller

Digital controller code is available on GitHub page together with same one used in ROS environment. Code could be used as basis and expanded upon with extra functionality like feedforward controller, reset option, different operation modes. Strategy regarding extra functionality must be based on system and time limitations. Avoiding developing unnecessary extra functionality is something that has to be kept in mind, since robot itself is unstable and underpowered.

As mentioned earlier, ROS is capable of achieving communication between nodes, but doesn't yet possess full functionality. In that stage digital controller is connected with topics that will feed controller with needed data. Since Beaglebone experiences considerable delay when transferring integer and float data it is not fully possible to test its functionality. Testing of the controller code was not practically possible due to time constrains and equipment malfunction. Therefore one way of testing it could be to use controller code as an On/Off controller, where encoder values are used as triggers for simple commands to the motor.

Improving ROS controller code is also highly encouraged. As of right now a lot of global variables are used, which can be improved aesthetically and practically by using separate class to store publishers, subscribers and node handles. Making it easier if later more topics are to be subscribed and published to.

11.4.3 Sensors

Full implementation of controller in its full potential cannot be executed without sensor strategy. Deciding how to use two different sensors with different information regarding robot's physical condition is important part of control strategy. Expanding knowledge and implementing it on that topic could be something to keep in mind when doing future work regarding control.

References

[1] Hjulstad, J. *et al.* (2019), *Instrumentation of biped prototype - Project description report*, Bachelor Thesis, Norwegian University of Science and Technology, Trondheim, Norway.

[2] Sparkfun 2020, *SparkFun 9DoF IMU Breakout - LSM9DS1* , viewed 7 May 2020, `https://www.sparkfun.com/products/13284`

[3] ShowMeCables August 2017, *RJ45 Pinout*, viewed 7 May 2020, `https://www.showmecables.com/blog/post/rj45-pinout`

[4] Altium ltd, 2020, *Altium Designer overview* , viewed 7 May 2020, `https://www.altium.com/altium-designer/`

[5] KiCad 2020, *About KiCad* , viewed 8 May 2020 `https://www.kicad-pcb.org/about/kicad/`

[6] JLCPCB(JiaLiChuang (HongKong) Co., Limited) January 2020, *JLCPCB website*, viewed 6 May 2020, `https://jlcpcb.com/`

[7] Mouser electronics May 2020, *Mouser electronics website*, viewed 7 May 2020. `https://no.mouser.com/`.

[8] The Open Organization 2020, *What is open hardware?*, viewed 8 May 2020, `https://opensource.com/resources/what-open-hardware`

[9] Open Robotics 2020, *About ROS*, Viewed 8 May 2020, `https://www.ros.org/about-ros/`

[10] *ROS wiki*, Viewed 18 May 2020, `http://wiki.ros.org/ROS/Tutorials`

[11] Open Robotics 2020, *ROS concepts*, Viewed 8 May 2020, `http://wiki.ros.org/ROS/Concepts`

[12] , *ROS overhead article*, viewed 18 may 2020, `http://design.ros2.org/articles/changes.html`

[13] Canonical Ltd 2020, *Debian is the rock on which Ubuntu is built*, viewed May 6 2020 `https://ubuntu.com/community/debian`

[14] eLinux.org 2020, *Installing Ubuntu on Beaglebone*, viewed 8 May 2020
https://elinux.org/BeagleBoardUbuntu

[15] Christensson, Per, November 2015, *System on chip design*, viewed 8 May 2020
https://techterms.com/definition/soc

[16] BeagleBoard.org 2020, *About BeagleBoard*, viewed 8 May 2020
https://beagleboard.org/about

[17] BeagleBoard.org 2020, *Beaglebone comparison.*, viewed 8 May 2020
https://beagleboard.org/bone

[18] The kernel development community 2020, *I2C and SMBus Subsystem*, viewed 8 May 2020
https://www.kernel.org/doc/html/v4.14/driver-api/i2c.html

[19] The Yocto project, *Linux Distribution*,viewed 11 may 2020
https://www.yoctoproject.org/about/

[20] Derek molloys webisite, *website for Beaglebone learning*, viewed 18 may 2020
http://derekmolloy.ie/tag/beaglebone-black/

[21] Hildreth, Derek July 2017. *Working with I2C sensor devices*, viewed 13 May
https://www.nutsvolts.com/magazine/article/working-with-i2c-sensor-devices

[22] STmicroelectronics 2020. *LSM9DS1 Overview*, viewed 14 May
https://www.st.com/en/mems-and-sensors/lsm9ds1.html

[23] STmicroelectronics 2015, *LSM9DS1 Datasheet*, viewed 19 May
https://www.st.com/resource/en/datasheet/lsm9ds1.pdf

[24] Github 2020, *NTNU-Biped-Robot GitHub repository*, viewed 19 May
https://github.com/cactiCode/NTNU-Biped-Robot

[25] About the PRU cores *Beaglbone PRU*, viewed 11 may 2020
https://beagleboard.org/pru

[26] Microsoft Corp. 2017 *Linux development with C++ in Visual Studio*, Viewed 12 May 2020
https://devblogs.microsoft.com/cppblog/linux-development-with-c-in-visual-studio/

[27] Erik Lillebø Hole 2019 *Introduksjon til Kicad, og footprints*, Norwegian University of Science and Technology, Trondheim, Norway.

[28] C.F. Sætre, *Stable Gaits for an Underactuated Compass Biped Robot with a Torso*,pp. 63-76, Master Thesis, June 2013, Norwegian University of Science and Technology, Trondheim, Norway.

[29] K. Bjørvik, P. Hveem, *Reguleringsteknikk*, Høgskolen i Sør-Trøndelag, August 2014.

[30] T. Anstensrud, *Styresystemer og reguleringsteknikk: Santidsdatateknikk Forelesning 9*,Norges Teknisk-Naturvitenskapelige Universitet, Februar 2019.

[31] F. Dessen, *Styresystemer og reguleringsteknikk: Reguleringsteknikk Forelesning Foroverkobling*, Norges Teknisk-Naturvitenskapelige Universitet, Februar 2019.

[32] R. Berge, *Matematikk/Fysikk: Fysikk Forelesning 24c Fysisk Pendel*, Norges Teknisk-Naturvitenskapelige Universitet, November 2019.

[33] C. Islek, *Industriell Automasjon: Motordrifter Forelesning Notat Chopere (DC-DC)*, Norges Teknisk-Naturvitenskapelige Universitet, Oktober 2019.

[34] D.E. Seborg, T.F. Edgar, D.A. Mellichamp, F.J. Doyle III, *Process Dynamics and Control*, Third edition, John Wiley Sons Inc, 2011

[35] C.F. Sætre, T. Anstensrud, L. Paramonov, A. Shiriaev, *Steps in model-based trajectory searching as a tool for biped design*, Department of Engineering Cybernetics, Norwegian University of Science and Technology.

[36] R. Poley, *Introduction to Control using Digital Signal Processor*,2007

[37] G.D. Davison, *The Damped Driven Pendulum: Bifurcation Analysis of Experimental Data*, Reed College, December 2011.

A Appendix - Schematics PCB

A.1 Main Connector Board schematics

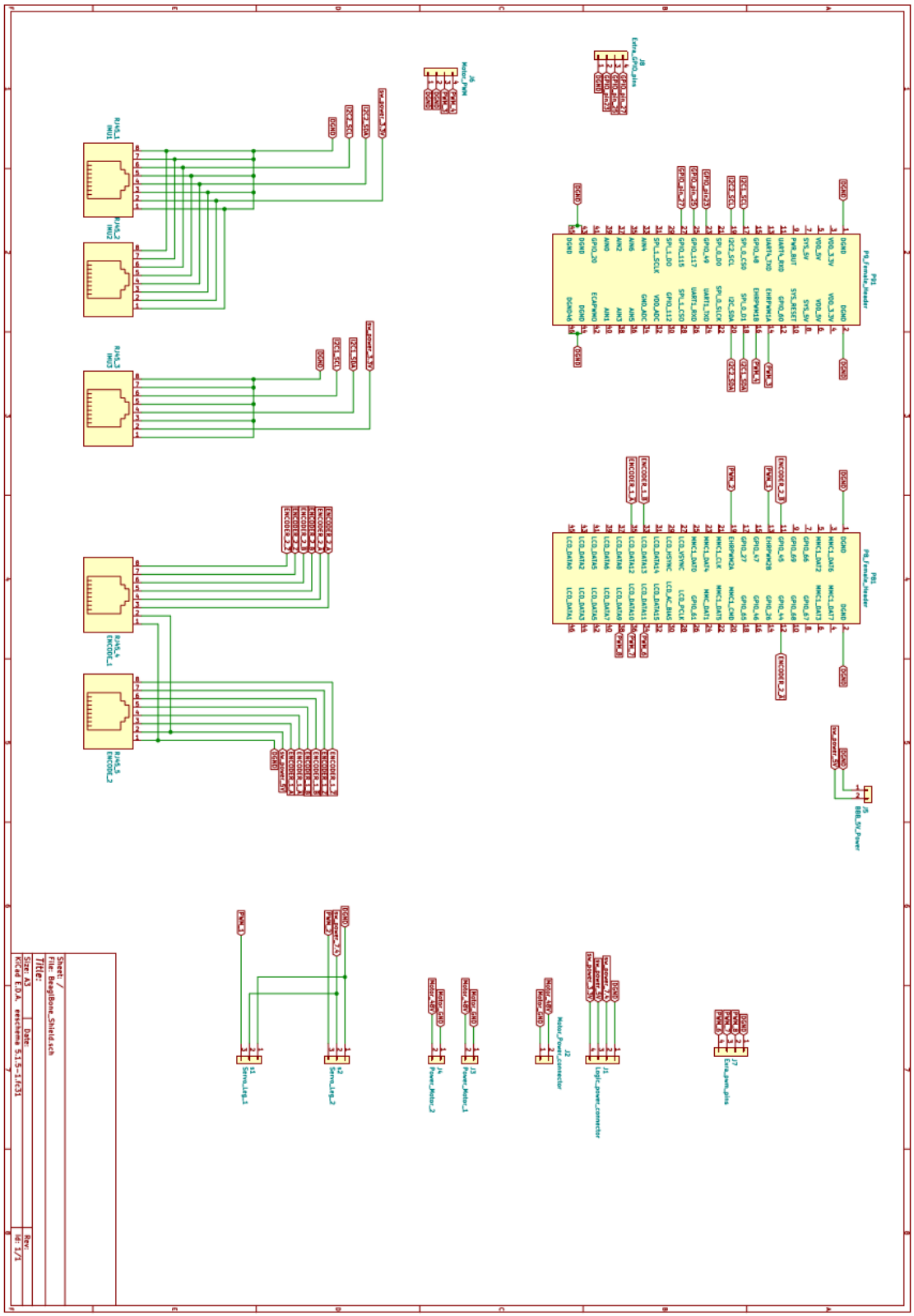


Figure 33: Schematic Main connector board

A.2 IMU Connector Board schematics

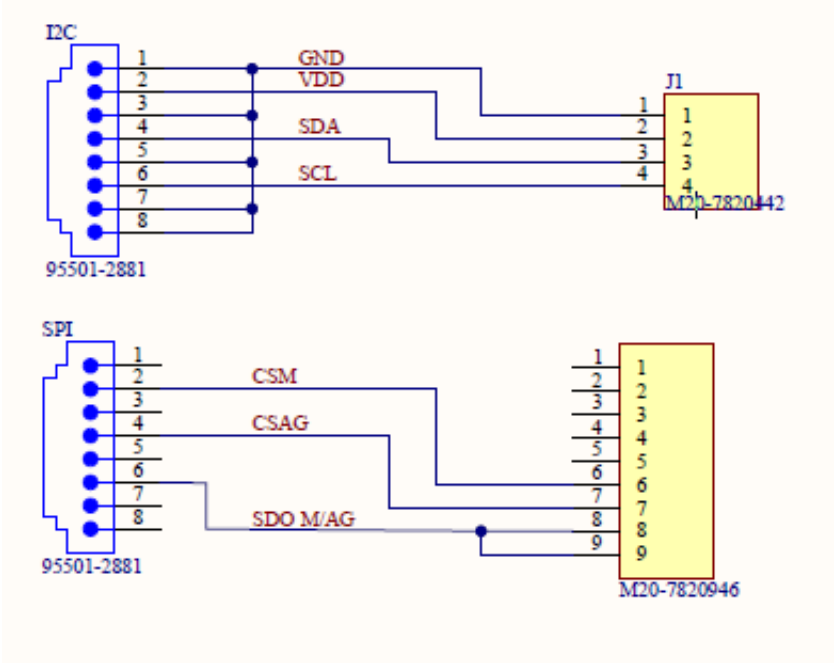


Figure 34: Schematic IMU connector board Rev. A

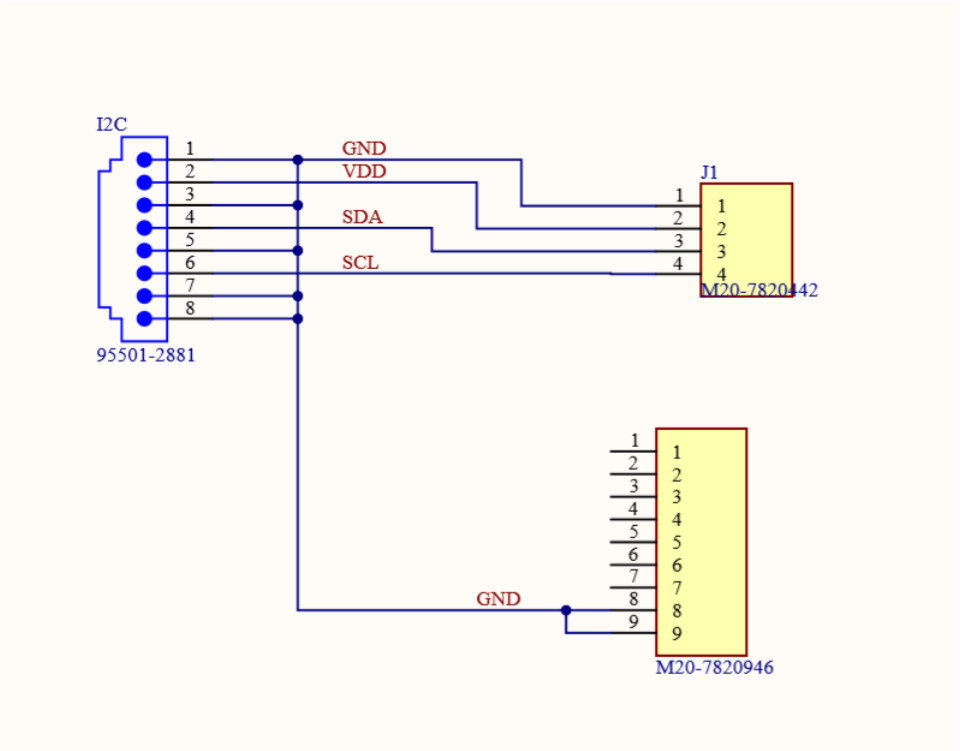


Figure 35: Schematic IMU connector board Rev. B

A.3 Encoder Connector Board schematics

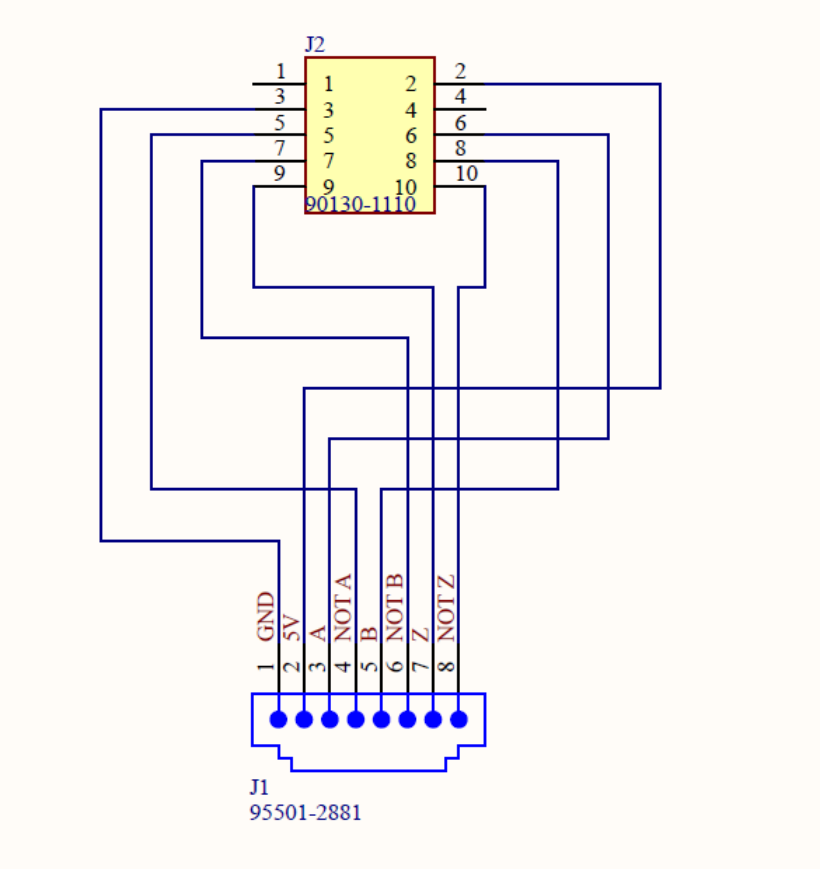


Figure 36: Schematic Encoder connector board

B Appendix - Components PCB

B.1 Components Main Connector Board

Component	Manufacturer	Manufacturer nr	Mouser nr
RJ45 Header	Molex	95540-7886	538-95540-7886
2 pos terminal block	Wurth Elektronik	691214110002	710-691214110002
4 pos terminal block	Molex	39357-0004	538-39357-0004
2x24 male pin connector	Molex	90131-0132	538-90131-0132
9 pin female pin connector	3M	929870-01-09-RA	517-929870-01-09-RA

Table 4: Components Main connector board

B.2 Components IMU Connector Board

Component	Manufacturer	Manufacturer nr	Mouser nr
4 - pin Vertical Socket	Harwin	M20-7820442	855-M20-7820442
9 -pin Vertical socket	Harwin	M20-7820946	855-M20-7820946
RJ45 Header	Molex	95540-7886	538-95540-7886

Table 5: Components IMU connector board

B.3 Components Encoder Connector Board

Component	Manufacturer	Manufacturer nr	Mouser nr
RJ45 Header	Molex	95540-7886	538-95540-7886
2x5 - pin Header	Molex	90130-1110	538-90130-1110

Table 6: Components Encoder connector board

C Appendix - Embedded system code

C.1 IMU

```
#include <iostream>
#include <unistd.h>
#include <cmath>
#include <linux/i2c-dev.h>
#include <sys/ioctl.h>
#include <fcntl.h>

#define g_sens_accs 0.061 //value from datasheet page 12
#define PI 3.14159 // constant pi



class LSM9DS1
{
    private :
    int file;
    char addr;
    float ax,ay,az;
    short s_ax,s_ay,s_az;
    float pitch,roll;

    char dataBuffer[5] = {0};

    void convertData()
    {
        s_ax = ((dataBuffer[1] << 8) | dataBuffer[0]);
        s_ay = ((dataBuffer[3] << 8) | dataBuffer[2]);
        s_az = ((dataBuffer[5] << 8) | dataBuffer[4]);
        ax = (s_ax*g_sens_accs)/1000;
        ay = (s_ay*g_sens_accs)/1000;
        az = (s_az*g_sens_accs)/1000;

        roll = atan2(ay,az);
        pitch = atan2(-ax, sqrt(ay * ay + az * az));

        //convert from radians to degrees
        pitch *= 180.0 / PI;
        roll  *= 180.0 / PI;

        // print converted axis values
        printf("\033[2J\033[1;1H");
        printf("X-axis: %f \n", ax);
        printf("Y-axis: %f \n", ay);
        printf("Z-axis: %f \n", az);
```

```
    printf("roll : %f \n", roll);
    printf("pitch : %f \n", pitch);
}

public :
LSM9DS1(char address)
{
    addr = address;
}
void setRegSensor(char reg,char value)
{
    char *bus ="/dev/i2c-2";
    //Open I2C bus, checks connection
    if((file = open(bus, O_RDWR)) < 0)
    {
        printf("Failed to open bus. \n");
        exit(1);
    }
    ioctl(file, I2C_SLAVE, addr);

    char config[2] = {0};
    config[0] = reg;
    config[1] = value;
    if(write(file, config, 2) < 0)
    {
        printf("Failed to open bus. \n");
        exit(1);
    }
    else
    printf("sucessful init \n");

}
void ReadSensor()
{
    char regis[1] = {0x28};
    write(file, regis, 1);

    if(read(file, dataBuffer, 6) != 6)
    {
        printf("Input/Output error \n");
        exit(1);
    }
    convertData();
}
void closeSensor()
{
    close(file);
```

```
    }
};

int main(int argc, char* argv[])
{
    char addresse = 0x6b;
    LSM9DS1 acceleration(addresse);
    char aksereg = 0x20;
    char setBit  = 0xc0;
    acceleration.setRegSensor(aksereg,setBit);
    int count = 0;

    while(count < 10)
    {
        acceleration.ReadSensor();
    }
    acceleration.closeSensor();

    std::cin.get();
    return 0;
}
```

C.2 Encoder

```
// Language dependencies
#include <cstdint>
#include <cstdlib>
#include <cstdio>
#include <iostream>

// POSIX dependencies
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

#define eQEP0 "/sys/devices/platform/ocp/48300000.epwmss/48300180.eqep"
#define eQEP1 "/sys/devices/platform/ocp/48302000.epwmss/48302180.eqep"
#define eQEP2 "/sys/devices/platform/ocp/48304000.epwmss/48304180.eqep"


// Constructor for eQEP driver interface object
class Encoder
{
 public :
    Encoder(int EncoderNumber)
    {
        switch (EncoderNumber)
        {
            case 0:
            path = eQEP0;
            break;
            case 1:
            path = eQEP1;
            break;
            case 2:
            path = eQEP2;
            break;
            default :
            std::cout << "invalid Encoder number" << std::endl;
            break;
        }
    }
void initEncoder (int32_t pos, int32_t period)
 {
  setMode();
  setPeriod(period);
  setPosition(pos);
  }
```

```
    int32_t getPosition()
    {
        int32_t        position;
        FILE* fp = fopen((this->path + "/position").c_str(), "r");

 if(fp == NULL)
  {
   std::cerr << "[eQEP " << this->path << "] Unable to open position" << st
   return 0;
  }

        fscanf(fp, "%d", &position);
        fclose(fp);

        return position;
    }

 private :
    std::string path;

void setMode()
 {
   FILE* setmode = fopen((this->path + "/mode").c_str(), "w");
   if(setmode == NULL)
   {
     std::cerr << "[eQEP " << this->path << "] Unable to open " << std::end
     return;
   }
    fprintf(setmode, "%u\n", 0); //absolute mode
    fclose(setmode);
    }

void setPosition(int32_t position)
 {

  FILE *setpos = fopen((this->path + "/position").c_str(), "w");
  if(setpos == NULL)
 {
    std::cerr << "[eQEP " << this->path << "] Unable to open " << std::endl
    return;
 }
    fprintf(setpos, "%d\n", position);
    fclose(setpos);
 }


void setPeriod(uint64_t period)
 {
```

```
    FILE* setp = fopen((this->path + "/period").c_str(), "w");
    if(setp == NULL)
    {
     std::cerr << "[eQEP " << this->path << "] Unable to open " << std::endl
     return;
    }


     fprintf(setp, "%llu\n", period);
     fclose(setp);
 }
};

int main ()
{
     Encoder encoder(2); //encoder number 2
     encoder.initEncoder(0, 10000000L);

     while(1)
     {
          std::cout << encoder.getPosition()*0.03 <<std::endl;
          printf("\033[2J\033[1;1H");
     }
     std::cin.get();
     return 0;
}
```

C.3 Servo

```
#include <iostream>
#include <fstream>
#include <string>

class servoControl
{


private:

    int m_frequency = 20006000;
    int m_dutyCycle;
    int m_PIN;
    int m_pinbool;
    std::string m_Pinpath;
    std::string m_PinpathNumber;
    std::string m_pinChip;


    void setPin(int pin)
    {
        switch (pin)
        {
            case 13:
                m_Pinpath = "P8_13_pinmux";
                m_PinpathNumber = "7:1";
                m_pinChip = "pwmchip7";
                m_pinbool = 1;
                break;
            case 19:
                m_Pinpath = "P8_19_pinmux";
                m_PinpathNumber = "7:0";
                m_pinChip = "pwmchip7";
                m_pinbool = 0;
                break;
            case 29:
                m_Pinpath = "P9_29_pinmux";
                m_PinpathNumber = "1:1";
                m_pinChip = "pwmchip1";
                m_pinbool = 1;
                break;
            case 31:
                m_Pinpath = "P9_31_pinmux";
                m_PinpathNumber = "1:0";
                m_pinChip = "pwmchip1";
                m_pinbool = 0;
                break;
```

```
            default:
                std::cout << "Not an implemented pin"<<std::endl;
                break;
        }
    }

void setPWMstate()
{
 std::ofstream pwmstate;
 pwmstate.open("/sys/devices/platform/ocp/ocp:" + m_Pinpath + "/state");

 pwmstate << "pwm";
 pwmstate.close();
}

void enablePWM()
{
 std::ofstream pwmEnable;
 pwmEnable.open("/sys/class/pwm/pwm-" + m_PinpathNumber + "/enable");
 pwmEnable << 1;
 pwmEnable.close();
}
void setPWMfreq(int frequency)
{
 std::ofstream PWMexportStream;
 PWMexportStream.open("/sys/class/pwm/" + m_pinChip + "/export");
 PWMexportStream << m_pinbool;
 PWMexportStream.close();

 std::ofstream freqSet;
 freqSet.open("/sys/class/pwm/pwm-" + m_PinpathNumber + "/period");
 freqSet << frequency;
 freqSet.close();
}

public:
 servoControl(int pinNumber)
{
 m_dutyCycle = 1000000;
 setPin (pinNumber);
 std::cout<< "Running with pin " << pinNumber << std::endl;
}
void setDutyCycle(int dutyC)
{
 std::ofstream DCfile;
 DCfile.open("/sys/class/pwm/pwm-" + m_PinpathNumber + "/duty_cycle");
 DCfile << dutyC;
 DCfile.close();
```

```
}

void servoInit()
{
 setPWMstate();
 setPWMfreq(m_frequency);
 setDutyCycle(m_dutyCycle);
 enablePWM();

}
};



int main()
{
    int pinN,pinN2;
    std::cout << "Pin Nr :";
    std::cin >> pinN;
    std::cout << std::endl;
    std::cout << "Pin2 Nr :";
    std::cin >> pinN2;
    std::cout << std::endl;;
    servoControl servoA(pinN); //initalize with given pin nr
    servoA.servoInit();
    servoControl servoB(pinN2); //initalize with given pin nr
    servoB.servoInit();

    for(int i = 0;i < 5;i++)
    {
        servoA.setDutyCycle(820000);
        servoB.setDutyCycle(820000);
        std::cout << "trying 82" << std::endl;
        std::cin.get();
        servoA.setDutyCycle(1900000);
        servoB.setDutyCycle(1900000);
        std::cout << "trying 19" << std::endl;
        std::cin.get();
    }
    std::cout << "end of test !"<< std::endl;
    return 0;
}
```

C.4 Motor

```
#include <iostream>
#include <fstream>
#include <string>


class motor
{
public:

motor(int pinNumber)
{
 m_dutyCycle = 1000000;
 setpwmPin (pinNumber);
 std::cout<< "Running with pin " << pinNumber << std::endl;
}
void setDutyCycle(int dutyC)
{
 std::ofstream DCfile;
 DCfile.open("/sys/class/pwm/pwm-" + m_PinpathNumber + "/duty_cycle");
 DCfile << dutyC;
 DCfile.close();
}
void setPWMfreq(int frequency)
{
 disablePWM();
 std::ofstream freqSet;
 freqSet.open("/sys/class/pwm/pwm-" + m_PinpathNumber + "/period");
 freqSet << frequency;
 freqSet.close();
 enablePWM();
}

void GPIOtoggle(int cape_nr,int pin_nr,int state)
{
 //intit addresses, states, and direction,
 setGPIOpin(cape_nr,pin_nr);
 setGPIOstate();
 //exportIOpin();
 setGPIOdirection("out");

 // Toggeling pin
 std::ofstream pinstream;
 pinstream.open("/sys/class/gpio/gpio"+ m_GPIONumber + "/value");
 pinstream << state;
 pinstream.close();
}
void toggleCurrentGPIOPin(int state)
```

```
{
    // Toggeling pin
 std::ofstream pinstream;
 pinstream.open("/sys/class/gpio/gpio"+ m_GPIONumber + "/value");
 pinstream << state;
 pinstream.close();
}
void motorInit()
{
 setPWMstate();
 exportPWM();
 setPWMfreq(m_frequency);
 setDutyCycle(m_dutyCycle);
 enablePWM();
}
     private:

          int m_frequency = 20000000;
          int m_dutyCycle;
          int m_PIN;
          int m_pinbool;
          std::string m_Pinpath;
          std::string m_PinpathNumber;
          std::string m_pinChip;

          std::string m_GPIONumber;
          std::string m_IOpinpath;

void setGPIOpin(int cape_nr,int IO_pin)
{
switch (cape_nr)
{
case 9:
 switch (IO_pin)
 {
  case 23:
  m_IOpinpath = "P9_23_pinmux";
  m_GPIONumber = "49";
  break;

  case 25:
  m_IOpinpath = "P9_25_pinmux";
  m_GPIONumber = "117";
  break;

  case 27:
  m_IOpinpath = "P9_27_pinmux";
  m_GPIONumber = "115";
```

```
   break;

   default:
   std::cout<<IO_pin << "not implemented pin"<< std::endl;
   break;

  }
break;
 case 8:
 switch (IO_pin)
 {
 case 34:
 m_IOpinpath = "P8_34_pinmux";
 m_GPIONumber = "81";
 break;

 case 36:
 m_IOpinpath = "P8_36_pinmux";
 m_GPIONumber = "80";
 break;

 case 38:
 m_IOpinpath = "P8_38_pinmux";
 m_GPIONumber = "79";
 break;

 default:
 std::cout<<IO_pin << "not implemented pin"<< std::endl;
 break;
 }
break;
default:
std::cout<<"pin : " <<cape_nr << " is an invalid cape_nr !" <<std::endl;
break;
}
}
void setGPIOstate()
{
std::ofstream IOstateStream;
IOstateStream.open("sys/devices/platform/ocp/ocp:" + m_IOpinpath + "/state"
IOstateStream << "default";
IOstateStream.close();
}
void exportIOpin()
{
std::ofstream IOexportstream;
IOexportstream.open("/sys/class/gpio/export");
IOexportstream << m_GPIONumber;
```

```
IOexportstream.close();
}
void setGPIOdirection(std::string direction)
{
std::ofstream IOdirStream;
IOdirStream.open("/sys/class/gpio/gpio" + m_GPIONumber + "/direction");
if (IOdirStream.fail())
{
exportIOpin();
setGPIOdirection(direction);
return;
}
IOdirStream << direction;
IOdirStream.close();
}


void setpwmPin(int pin)
{
switch (pin)
{
 case 13:
  m_Pinpath = "P8_13_pinmux";
  m_PinpathNumber = "7:1";
  m_pinChip = "pwmchip7";
  m_pinbool = 1;
  break;
 case 19:
  m_Pinpath = "P8_19_pinmux";
  m_PinpathNumber = "7:0";
  m_pinChip = "pwmchip7";
  m_pinbool = 0;
  break;
 case 29:
  m_Pinpath = "P9_29_pinmux";
  m_PinpathNumber = "1:1";
  m_pinChip = "pwmchip1";
  m_pinbool = 1;
  break;
 case 31:
  m_Pinpath = "P9_31_pinmux";
  m_PinpathNumber = "1:0";
  m_pinChip = "pwmchip1";
  m_pinbool = 0;
  break;
 case 14:
  m_Pinpath = "P9_14_pinmux";
  m_PinpathNumber = "4:0";
```

```
  m_pinChip = "pwmchip4";
  m_pinbool = 0;
  break;
 case 16:
  m_Pinpath = "P9_16_pinmux";
  m_PinpathNumber = "4:1";
  m_pinChip = "pwmchip4";
  m_pinbool = 1;
  break;
 default:
  std::cout << "Not an implemented pin"<<std::endl;
  break;
}
}

void setPWMstate()
{
std::ofstream pwmstate;
pwmstate.open("/sys/devices/platform/ocp/ocp:" + m_Pinpath + "/state");
//f   inn feiltest
pwmstate << "pwm";
pwmstate.close();
}

void enablePWM()
{
std::ofstream pwmEnable;
pwmEnable.open("/sys/class/pwm/pwm-" + m_PinpathNumber + "/enable");
pwmEnable << 1;
pwmEnable.close();
}
void disablePWM()
{
std::ofstream pwmEnable;
pwmEnable.open("/sys/class/pwm/pwm-" + m_PinpathNumber + "/enable");
pwmEnable << 0;
pwmEnable.close();
}

void exportPWM ()
{
std::ofstream PWMexportStream;
PWMexportStream.open("/sys/class/pwm/" + m_pinChip + "/export");
PWMexportStream << m_pinbool;
PWMexportStream.close();
}
```

```
};
int main()
{
    int pinN = 29;
    motor control(pinN);
    control.motorInit();


    control.setDutyCycle(1000000);   // 80%
    control.setPWMfreq(80000000);    //100Hz
    control.GPIOtoggle(9,23,1); // MOTOR OFF

    while(1)
    {
    control.GPIOtoggle(9,25,0);   //set GPIO pin 25 on cape 9 to 0
    control.GPIOtoggle(9,23,0);   //set GPIO pin 23 on cape 9 to 0
    std::cin.get();
    control.GPIOtoggle(9,25,1); //set GPIO pin 25 on cape 9 to 1
    control.GPIOtoggle(9,23,1); //set GPIO pin 25 on cape 9 to 1
    std::cin.get();
    }
    return 0;
}
```

D Appendix - PD - Controller code

D.1 PD - controller

```
#include "pd.h"

//*********************
//Constructor function
//*********************

PD_Controller::PD_Controller(float kp, float kd)
{
  Initialise();
  KP=kp;
  KD=kd;
}

//Initialisation

PD_Controller::Initialise(){
  float error = 0.0;
  float previous_error_1 = 0.0;
  float previous_error_2 = 0.0;
  float previous_controlSignal=0.0;
}

//************************
//Configuration functions
//************************


void PD_Controller::Set_Parameters(float kp, float kd)
{
  float KP=kp;
  float KD=kd;
}

void PD_Controller::SetOutputRange(float min_Output, float max_Output)
{
  float MAX_OUTPUT=max_Output;
  float MIN_OUTPUT=min_Output;
}

//*************************
//Primary operating functions
//*************************

void PD_Controller::Set_Setpoint (int set_SetPoint)
{
```

```
  int Set_Setpoint=set_SetPoint;
}


float PD_Controller::getControllValue(float set_SetPoint, float sensor)
{
  float error = set_SetPoint - sensor;
  float controlSignal = previous_controlSignal
  +(KP*(error - previous_error_1))
  +(KD*(error-2*previous_error_1+previous_error_2));

 if (controlSignal>MAX_OUTPUT)
 {
   controlSignal=MAX_OUTPUT;
  }
  else if (controlSignal<MIN_OUTPUT)
  {
    controlSignal=MIN_OUTPUT;
  }
  return controlSignal;
}

void PD_Controller::Update()
{
  float previous_error_2=previous_error_1;
  float previous_error_1 = error;
  float previous_controlSignal=controlSignal;
}
```

D.2 Header file

```
#ifndef pd.h
#define pd.h

class PD_Controller
{
  private:
  float KP,KD;
  float controlSignal;
  float Set_Setpoint;
  float previous_controlSignal;
  float error;
  float previous_error_1;
  float previous_error_2;
  float MAX_OUTPUT;
  float MIN_OUTPUT;


  public:
  PD_Controller(float, float);
  float getControllValue(float);
  void Set_Parameters(float, float);
  void SetOutputRange(float, float);
  void Set_Setpoint(float);
  void Update();
  void Initialise();
};

#endif
```