Automatic Control and Robotics

Implementation of a robot platform to study bipedal walking

Master Thesis

Autor: Dimitris Zervas
Director/s: Dr. Manel Velasco and Dr. Cecilio Angulo

Convocatòria: April 2016

Escola Tècnica Superior
d'Enginyeria Industrial de Barcelona

Abstract

On this project, a modification of an open source, 3D printed robot, was implemented, with the purpose to create a more affordable bipedal platform proper for studying Bipedal Walking algorithms.

The original robot is a part of an open-source platform, called *Poppy*, that is formed from an interdisciplinary community of beginners and experts. One of the robots of this platform, is the *Poppy Humanoid*. The rigid parts of the Poppy Humanoid (as well as the rest of the Poppy platform robots) are 3D printed, a key factor of lowering the cost of a robot. The actuators used though, are expensive commercial DC-motors that increase the total cost of the robot drastically.

This high cost of the actuators of Poppy, led this project to modify cheaper actuators while maintaining the same performance of their predecessors. Taking apart the components of the cheaper actuator, only the motor, the gears and the case that host them were kept, and a new design was made to control the motor and to meet the requirements set from the commercial motors. This new design of the actuator include a 12-bit resolution magnetic encoder to read the position of the shaft of the motor, a driver to run the motor, and also an embedded *Arduino* micro-controller. This feature of an Arduino as part of the actuator, gives the advantage over the commercial motor, as the user has the freedom to upload his own codes and to implement his own motor controllers.

The result is a fully programmable actuator hosted on the same motor case. The size of this actuator though, is different from the commercial one. In order to mount the new actuators to the platform, Joan Guasch designed proper 3D printed parts. Apart of these parts, Joan also modified the leg design, in order to add another joint on the ankle (roll) as this Degree of Freedom (DoF) is important for Bipedal Walking algorithms and was missing from the original Poppy Humanoid leg design. The modified robot, is called *Poppy-UPC* and is a 12 DoF biped platform.

For the communication between the motors and the main computer unit, a serial communication protocol was implemented based to the RS-485 standard. Multiple receivers (motors and sensors) can be connected to such a network in a linear, multi-drop configuration. The main computer unit of Poppy-UPC is an *Odroid-C1* board. Essentially, this board is a Quad-core Linux computer fully compatible to run *ROS*. Odroid is acting as the master of the network and is gathering all the informations of the connected nodes, in order to publish them in ROS-topics. That way, the Poppy-UPC is connected to the ROS environment and ROS packages can be used for any further implementation with this platform.

Finally, following the open-source spirit of the Poppy platform, all the codes and information are available at *https://github.com/dimitris-zervas*.

Acknowledgement

I wish like to thank both Dr. Manel Velasco and Dr. Cecilio Angulo for their excellent tutoring and advices during this project. They provided an environment to work, that any student would highly appreciate. Special thank to Joan Guasch not only for his contribution to this robot, but also for the endless brainstormings.

Last but not least, I would like to thank my family for supporting me all these years. I couldn't achieve what I did without them.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction and Motivation

Humanoid robots is one of the most interesting areas of robotics in the last years and still under heavy research. The progress of these robots though is impressive. Nowadays, humanoid robots have been seen running, climbing stairs, playing football, driving a car and most recently, walking outdoors semi-autonomous, on an uneven terrain. It is a common belief that the innovation curve of humanoid robotics area is now growing massively.

It is also fair to say that these robotics systems are as impressive as complex. To conclude to that, someone only need to think the number of Degrees of Freedom (DOF) these robots have. Very simple bipeds without any upper part (only legs) start with 6 degrees of freedom [37] up to 12 degrees of freedom (6 per leg) [25]. And of course, for a fully humanoid biped this number increases drastically. But the number of the DoF is not the only difficulty that someone might encounter. The hardware to support all these DoF or the limitations of the existing technologies is some of them.

As any new research area, it takes advantage of new technologies but also lead to new ones. For example, technologies such as 3D-printing and new light weight materials are improving the performance of the robots. The team behind the "New Generation ATLAS" from "Boston Dynamics"[12][30] took advantage of that and used 3D printing to create the legs, so the actuators and hydraulic lines are embedded in the structure, rather than made out of separate components. Or the mechanical limitation that the typical robotics actuators have, led to the design of the *Series Elastic Actuators*[23]. These force controllable actuators allow low impedance algorithms that result in efficient and graceful walking that is robust to disturbances and rough terrain. Finally, just recently, the *Italian Institute of Technology* (IIT) and control systems manufacturer *Moog* formed a new robotics lab to develop "next-generation actuation and control technologies for autonomous robots"[21].

As it might be clear by now, it makes sense that this research area cannot be easily accessible for everyone (yet) and the main reason for that is the high cost of these robots. Most of these robots are prototypes and developed in various labs with unique designed components. The majority of the complete developed platforms come -of course- from the research area, while the few commercial options, are either extremely expensive [15][28] or improper for studying humanoid robots algorithms and with restricted access to the hardware[2]. This limited access to a humanoid platform is according to the author the biggest problem for students and researchers that want to engage on

that area.

There is an interesting option though that, most important, is open-source and easily accessible. It is called *Poppy Humanoid*[17]. Poppy Humanoid is 85cm high and is particularly lightweight (3.5kg). This robot has 25 degrees of freedom with a multi-articulated trunk (5 DoFs).It is using 3D-printed parts and for actuators it has the known to the academic area *Dynamixel*[1] motors. Of course the Poppy humanoid can not be compared to the humanoid robots that lead this technology, but according to the author's belief, is one of the most interesting platform for someone to start studying humanoid robots and in particular, bipedal walking and its fundamentals.

Unfortunately, even though the printing of the parts might not be that expensive, the actuators that are used increase the cost of the robot significant. And this fact leads to the *introduction* of this project.

The project is about the modification of the *open-source* humanoid robot called Poppy, with the goal to make it more *cost-effective* and with more *versatile hardware*, in order to be more attractive to people that want to study *Bipedal Walking*. In particular, the structural part is based on the Poppy humanoid and since study Bipedal Walking is the main goal for this robot platform, only the lower part (legs) of Poppy is considered.

The most expensive parts of Poppy, are the *Dynamixel* actuators that are used. With reducing the cost in mind, the first modification it was decided to be the change of the motors. After an extensive search on the market, it was observed that the reason the Poppy team are using these actuators is because they are the only choice on the market[2], based on the requirements of a humanoid robot and a reasonable price. Since according to the goal of this project this price is high enough, the decision was made to buy cheaper motors and modify them to meet at least the performance quality of its predecessors.

The new motors that were bought, are from the market of hobbyists RC air-planes. These motors are powerful enough but since they are not intended for robotics applications, their control system lacks a lot of capabilities. For that purpose all the electronic parts were taken out and a new control system of the motor was designed. The base of this new control system is an embedded micro-controller and in particular the same used by the *Arduino UNO*[3] board. This seems to be a very interesting feature, as the user can re-program directly each of the actuators of the robot, according to his need.

This new system is hosted on a designed Printed Circuit Board (PCB) that fits in to the original case of the motor. As the size of the new motor is not the same with Dynamixel one, modifications also on the 3D-printed parts of Poppy were made in order to mount the new motors on.

For that purpose, Joan Guasch, a colleague from this master degree, designed 3D-printed parts that were mounted to the motors, allowing them to fit to the rest of the Poppy. Joan, apart of this design, he also made some modification to the rigid parts of Poppy, such as adding an extra ankle joint (roll) to each of the legs. The new Poppy design is called **Poppy-UPC**. Joan's thesis is now about Poppy-UPC and in particular the simulation of the platform in the *Gazebo* environment[4].

[1] www.robotis.com/xe/dynamixel_en
[2] At least at the time the Poppy robot was designed.
[3] www.arduino.cc/en/main/arduinoBoardUno
[4] gazebosim.org

Poppy-UPC can be commanded by either an Odroid board or a normal PC. The communication between the computer unit and the motors and sensors of Poppy-UPC is also presented on this project. A library written in C++ was created for that purpose. This library then was integrated to a ROS package, that is capable of sending commands to the motors as well as *publishing* information to *topics*. This *messages* can easily be used from other ROS packages related to robotics, such as kinematics, dynamics or motion planning libraries.

The high cost of a biped platform, even for the open-source one, is the main motivation for this project. The idea of providing a platform to study bipedal walking by keeping the cost for creating it low, open-source and easily modified while keeping the standards high, compared to the rest of the platforms, is what drove the author to implement this project.

1.1 Objectives

The overall objective of this project is to reduce the cost of an open-source design of a humanoid robot, keeping the quality on the same standards. These modification are made under the scope of making a biped platform proper for studying bipedal walking algorithms. As the motors of the original design is the most expensive part, the main objective is to replace these motors. The new motors is not only desired to be cheaper but also more "open" than their predecessors.

1.2 Scope of the project

The main objective of this project is the replacement of the original expensive actuators with cheaper, but of the same performance standards. The original motors, sense the position of the shaft by using an encoder of high accuracy (0.088 deg), drive the motor using typical mosfets and control all the hardware with an embedded micro-controller. The motors can communicate with other devices through a serial communication protocol. They also come with an implemented *PID* controller where the user can tune its gains. Apart of the gains there are more parameters that can be commanded (set the goal-position, read the feedback etc.).

The motors that were bought to replace the original ones, barely have some of these features as they are not intended for robotic applications but rather for use in RC-airplanes. In order to keep the performance standards similar to the more expensive actuators, all the electronic parts were taken out. What was kept was the motor with the gear system and the outer case. *The main part of the scope of this project was to design a new system that would make the cheap actuators as good as the expensive ones.*

This new system, incorporates a high accuracy encoder as well. It also consist of a full bridge motor driver that apart of the *H-bridge* it also offers various protections to the motor (e.g. over-heating shut-down). Finally, to control the motor, a micro-controller was also used, in particular the same that is used on the *Arduino UNO* boards. This micro-controller is running 25% faster than the typical Arduino and for that purpose, a new boot loader had to be created.

A **P**rintied **C**ircuit **B**oard (PCB) was also designed to host this new system. This board has the

proper size in order to fit in the outer case of the motor. The details of the design of this board are part of the scope of previous work that was done related to this project and as such, are included to the appendices of this document.

The communication between the motors, could be implemented either through a I^2C or *Serial* port. An implementation of both option was done and a comparison between them and the reasons why Serial communication was chosen, is explained later on.

The main computer unit is an Odroid-C1 board, but based on the design of the communication protocol, a 'normal' PC can be used as well. A library written in C++ was created for the communication between the Odroid board and the motors. Since Odroid is fully capable of running ROS, a ROS-package was created that uses this library. The role of this package is to create the link between the library and the ROS environment, as it can send data to the motors (e.g. instructions) and publish the received information (e.g. feedback) to ROS topics.

As for the control of the DC-motor, parameter estimation is implemented to find the parameters of a linearised model. These parameters are used to simulate the response of the motor and to tune the gains of the controller. These gains are used afterwards to the real motor.

Due to the limited time concepts such as,

- Calculating important variable for most of the bipedal walking algorithms (e.g. Center of Mass, Center of Pressure),
- Dynamic analysis,
- Implementation of a Bipedal Walking algorithm,

are not include in the scope of this project. At the end of this document, a forward kinematics analysis is given. As there was no time to test it to the real robot, it is not included to the scope of the project. Nevertheless it might be useful for any further improvements of this biped platform.

1.3 Document Structure

This thesis consists of eight chapters.

- On **Chapter 2**, some basic terminology about the fundamentals of bipedal walking are given. It also presents stability margins criteria and simplified models that are used in bipedal walking algorithms.
- On **Chapter 3**, it is shown the state of the art of biped robots and is briefly shown how the concepts of Chapter 2 were implemented on real robots.
- **Chapter 4** presents the original Poppy Humanoid as well as the Poppy-UPC biped.
- **Chapter 5** is explaining the communication network that was implemented and how the nodes are configured in order to comply to the protocol's rules.

- **Chapter 6** is about the modelling of the dc motor, it shows how to linearise it based on input-output data and also offers some controllers that can be used.
- **Chapter 7** is presenting a theoretical approach of the Forward Kinematics.
- Finally, **Chapter 8** is about the conclusion of this project and any possible further improvements.

Chapter 2

Bipedal Walking Fundamentals - Terminology

On this chapter the fundamentals of Bipedal Walking will be given, along with the basic terminology. These terms are going to be used extensively in the rest of this thesis.

The definition of the position of the human body (and hence any humanoid robot) start with the body, set in the *anatomical position*, in which the person is standing with the feet together and the arms by the side of the body. This position, together with the three reference frames is shown in Fig. 2.1. These three frames are:

- The **Sagittal** plane, parallel to the x unitary vector of the frame.
- The **Transverse** plane, parallel to the z unitary vector of the frame.
- And the **Frontal** or **Lateral** plane that is parallel to the y unitary vector of the frame.

Motions of any biped, are described relative to these three planes.

2.1 Terminology used in Gait Analysis

In order to describe the human gait, some terms will be first introduced.

Gait

"Human gait refers to locomotion achieved through the movement of human limbs ... Human gaits are the various ways in which a human can move, either naturally or as a result of specialized training."[1]. The walk is a gait which keeps at least one foot in contact with the ground at all times.

[1]Source: en.wikipedia.org/wiki/Gait_(human)

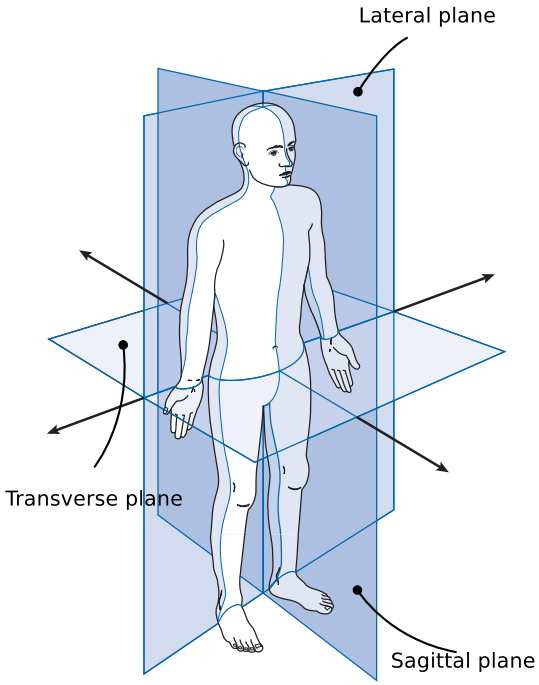


Fig. 2.1: The anatomical position with the three reference frames.

Double and Single Support

The Double Support is the state that the human, or the biped, is supported by both feet. It consists of two distinct contact surfaces, but is not necessary that any of the feet is in full contact with the floor (In Fig. 2.2b only the toes of the right foot are in contact). Similarly, the Single Support is the state where the human or biped, is supported by only one of the feet.

Support Polygon

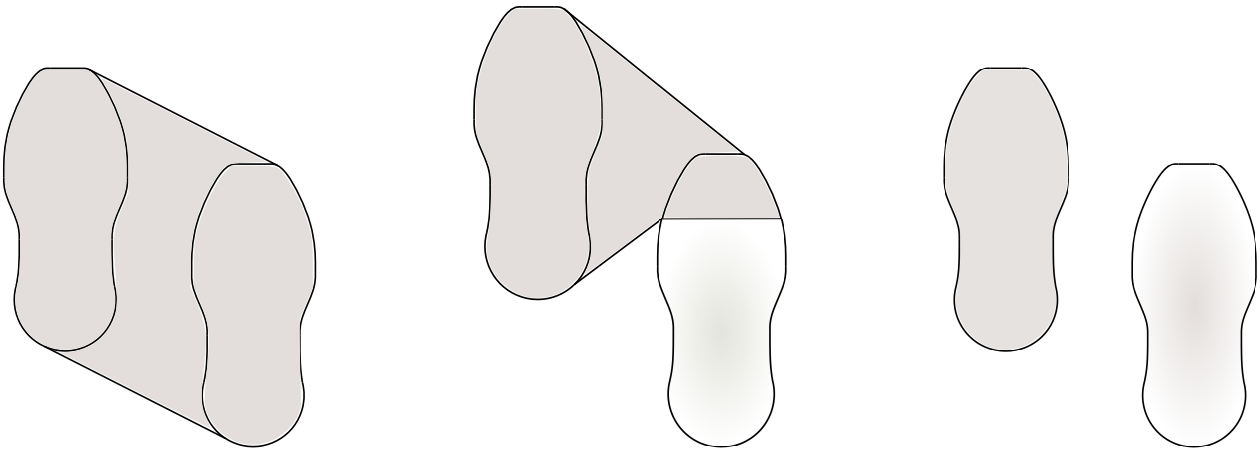
The Support Polygon is the convex hull that covers all the area of the floor that supports the human or the biped. Fig. 2.2 shows three examples of Support Polygons in both the Single and the Double Support state.

Swing leg

It is the leg that is moving in the air in order to achieve a step.

Stance leg

It is the leg that fully supports the human or the biped during the movement of the swing leg.



(a) Doubles Support **(b)** Initial Swing **(c)** Single Support

Fig. 2.2: Typical shapes of Support Polygon

Gait Cycle

Walking is a repetitive change of the human or biped gait that changes from the Double Support to the Single Support state, and so on. The swinging of the leg, is distinguished to three stages:

- Initial swing
- Mid-swing
- Terminal swing

During the *Initial* and *Terminal* swing, the gait is still in *Double Support* state while only during the *Mid-swing* the gait is in *Single Support* state.

The gait cycle is defined as the time interval between two occurrences of one of the repetitive phases of walking, as it is shown in Fig 2.3.

2.2 Stability Criteria for Bipedal Walking

According to J. Pratt and Tedrake [13] *Fall* is defined "...when a point on the biped, other than a point of the feet of the biped, touches the ground" and therefore, "...stability of the biped, is defined in terms of whether or not the biped will fall down".

[1]Image was taken from the book "Gait Analysis, An Introduction", Michael W. Whittle, 4th Edition

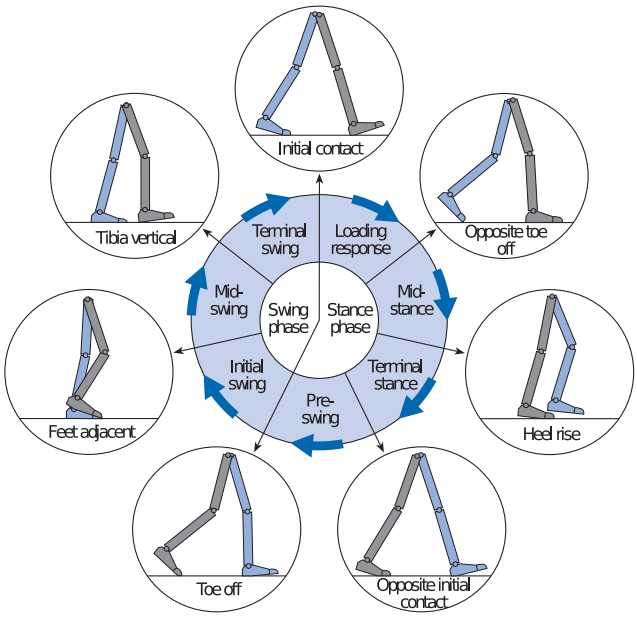


Fig. 2.3: Positions of the legs during a single gait cycle by the right leg (gray).[1]

In literature, there are mainly three important *stability margins* for Bipedal Walking. These are:

- The eigenvalues of the Poincare return maps. [19].
- The Zero Moment Point, ZMP and the Center of Pressure, CoP. [36]
- The reservation of Angular Momentum [22].

2.2.1 Eigenvalues of Poincare Return Maps

Given a continuous dynamic system, a *Poincare section* is defined as a lower dimension subspace that forms an intersection of a periodic orbit (such as a limit cycle) in the state space, transversal to the flow of the system. Considering a periodic orbit that start its cycle from within this section (its initial condition in state space are part of the section) and within a small deviation from the point of intersection where the limit cycle passes, the orbit will return and cross again this section, trying to approach the limit cycle again. Defining the discrete event X_{n+1} as the next point where the periodic orbit will cross the Poincare section, it can be observed that the orbit follows a linear relation such as,

$$X_{n+1} = KX_n \tag{2.1}$$

where X is the vector of deviations from the point where the limit cycle crosses the Poincare section and K is a liner return matrix. Fig.2.4 shows a Poincare section and two consecutive discrete events of the orbit crossing this section. One of the eigenvalues of K will be 1.0 (which correspond to the

case of the orbit being on the limit cycle). If the magnitude of the rest of eigenvalues are less than one, then the limit cycle is stable. Then the magnitude of the largest eigenvalue of K describes a stability margin for a periodic system.

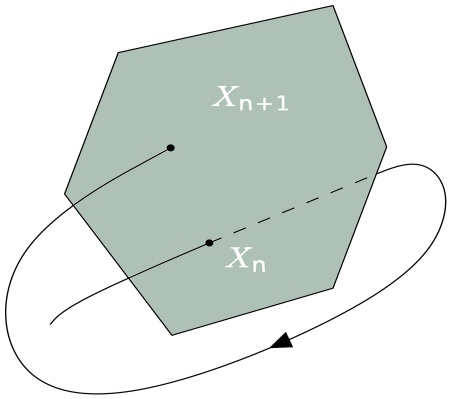


Fig. 2.4: Two discrete events of crossing the Poincare section

The disadvantage of using the eigenvalues of Poincare return map, is that assumes periodicity and that is valid only for small deviation from the limit cycle. Assuming that the system of the biped is periodic (e.g. walking on a flat terrain with constant speed) any disturbance such as a push can not be analysed with this stability margin.

2.2.2 Zero Moment Point and Center of Pressure

The *Zero Moment Point*, ZMP, was introduced by M. Vukobratovic and D. Juricic around 1970 [36].

"ZMP is defined as that point on the ground at which the net moment of the internal forces and the gravity forces has no component along the horizontal axes".

Considering a static biped that is in the *single support* state and the foot is in full contact to a flat ground. To simplify the model, someone can replace the effect of the mechanism above the ankle of the foot, with a force $\mathbf{F_A}$ and a moment $\mathbf{M_A}$ as it is shown in Fig. 2.5. If the biped is in balance, the ground reaction must equilibrate the $\mathbf{F_A}$ and $\mathbf{M_A}$. The ground reaction is also composed from a force $\mathbf{R}$ and a moment $\mathbf{M}$. The ZMP is the point on the *support polygon* (on this case, the area of the foot) that ground reaction is applied and balances the biped. ZMP can also be regarded as a dynamic equivalent of the floor projection of the Center of Mass (CoM).

As it might already be clear, ZMP is well suited in cases where the biped stands or walk on a flat floor and the foot is at rest. This will be more clear in the following analysis, as this was given in [36].

The horizontal components of $\mathbf{R}$ (R_x and R_y) represent the friction force and is equal to the force that is represented by the horizontal component of $\mathbf{F_A}$ (F_{A_x} and F_{A_y}), where the vertical reaction moment M_z is equal to the vertical component of $\mathbf{M_A}$, M_{A_z}, plus the moment that $\mathbf{F_A}$ produces. The vertical reaction force R_z balances the vertical forces. What is not balanced yet is

the horizontal component of M_{A_x}. "Since the ground reaction force is always oriented upwards, horizontal components of all active moments can be compensated for, only by changing the position of the reaction force within the support polygon." *Therefore, M_{A_x}, will shift the point where the reaction force is applied, to balance the additional load.*

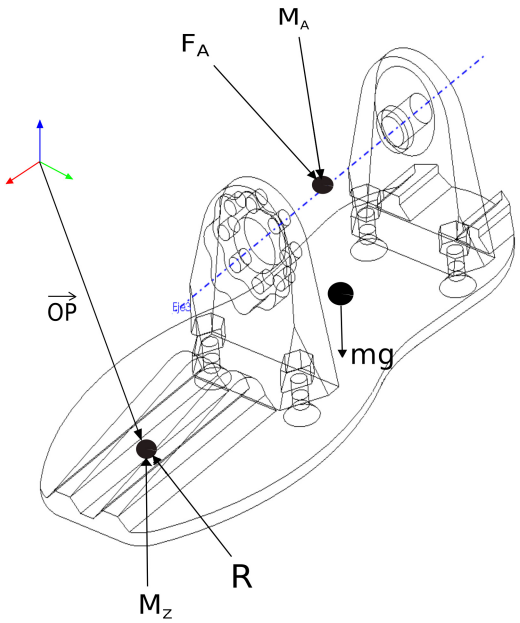


Fig. 2.5: Forces and Moments acting on the foot of the biped.

If the posture of the biped is such that the ZMP lies outside of the support polygon, then the actual ZMP will be at the edge of the foot and an uncompensated moment will exist that will produce a torque around the ZMP and if the biped is not able to take a step, it will fall. Therefore, the *ZMP stability margin* is defined as the distance from the ZMP to the nearest edge of the convex hull of the support polygon.

The Center of Pressure (CoP) is the point on the support polygon where the total sum of the reaction forces act, causing a force but no moment. If the biped is dynamically stable then the ZMP and the CoP is exactly the same point. The only difference between the two, is that the latter one cannot exist outside of the support polygon.

Derivation of the ZMP

There are various approaches for someone to calculate the ZMP. For most of them, the *forward kinematics* of the biped is needed. The following is a short overview of the derivation of ZMP as it was shown in [8].

The following assumption are made:

- The biped consists of only rigid links.

- The floor is rigid and motionless.
- The feet can not slide over the floor.
- All joints are actuated.

The first thing needed is the total center of mass, M_{CoM}, and the distance of it from the base-frame origin, $\mathbf{p}_{CoM}$:

$$M_{CoM} = \sum_{i=1}^{n} m_i \tag{2.2}$$

The total *linear momentum* and *angular momentum* expressed in the base-reference frame is:

$$P = \sum_{i=1}^{n} m_i \dot{\mathbf{p}}_i \tag{2.3}$$

$$H = \sum_{i=1}^{n} (\mathbf{p}_i \times m_i \dot{\mathbf{p}}_i + \mathbf{I}_i \boldsymbol{\omega}_i) \tag{2.4}$$

where $\mathbf{I}_i$ and $\boldsymbol{\omega}_i$ are respectively the inertia tensor and the angular velocity with respect to the base-frame.

The derivatives of the above quantities are:

$$\dot{\mathbf{P}} = \sum_{i=1}^{n} m_i \ddot{p}_i \tag{2.5}$$

$$\dot{\mathbf{H}} = \sum_{i=1}^{n} (\dot{\mathbf{p}}_i \times (m_i \dot{\mathbf{p}}_i) + \mathbf{p}_i \times (m_i \ddot{\mathbf{p}}_i) + \mathbf{I}_i \dot{\boldsymbol{\omega}}_i + \boldsymbol{\omega}_i \times (\mathbf{I}_i \boldsymbol{\omega}_i)) \tag{2.6}$$

and now the equations to calculate the zero moment point are:

$$x_{zmp} = \frac{M_{CoM}\, g_z\, p_{CoM_x} + z_{zmp} \dot{P}_x - \dot{H}_y}{M_{CoM}\, g_z + \dot{P}_z} \tag{2.7}$$

$$y_{zmp} = \frac{M_{CoM}\, g_z\, p_{CoM_y} + z_{zmp} \dot{P}_y - \dot{H}_x}{M_{CoM}\, g_z + \dot{P}_z} \tag{2.8}$$

where g_z is the z-component of the gravity vector.

Similar equation were derived by Huang [16] where it was assumed that $z_{zmp} = 0$,

$$x_{zmp} = \frac{\sum_{i=1}^{n} m_i \left(\ddot{z}_i + g\right) x_i - \sum_{i=1}^{n} m_i \ddot{x}_i z_i - \sum_{i=1}^{n} I_{iy} \ddot{\Omega}_{iy}}{\sum_{i=1}^{n} m_i \left(\ddot{z}_i + g\right)} \tag{2.9}$$

$$y_{zmp} = \frac{\sum_{i=1}^{n} m_i \left(\ddot{z}_i + g\right) y_i - \sum_{i=1}^{n} m_i \ddot{y}_i z_i - \sum_{i=1}^{n} I_{ix} \ddot{\Omega}_{ix}}{\sum_{i=1}^{n} m_i \left(\ddot{z}_i + g\right)} \tag{2.10}$$

2.2.3 Reservation of Angular Momentum

Based on the observation from researchers that humans appear to regulate angular momentum about the Center of Mass, CoM [1], also know as *spin angular momentum*, J. Pratt and Tedrake [13] indicated the following.

They believe that, the spin angular momentum by its own is not enough to indicate if the biped will fall or not. However, by minimizing angular momentum (which is limited because of the limits of the joints), the biped can reserve it. This reservation, can be utilised to help the biped recover for a disturbance such as a push.

2.3 Linear Inverted Pendulum

So far, the stability margins that were discussed, were relying strongly on the knowledge of the dynamic parameters, such as the inertia and the location of the center of mass of each link. This of course presumes accurate model of the robot and highly complex algorithms.

Another approach was proposed by Kajita [18]. By simplifying the model of the robot as much as to be defined only by its total center of mass, a *3D Linear Inverted Pendulum* model was developed. The model it was derived by a general three-dimensional inverted pendulum with an additional constraint of being allowed to move on an arbitrary defined plane. Kajita showed that this model allows someone to design separate controllers for the sagittal and the lateral planes, a property that simplifies drastically the walking algorithms.

When the biped is supporting its total CoM on one leg (single support state), a inverted pendulum from the foot to the CoM through a telescopic "leg", can describe the dominant dynamics of the robot. Such a pendulum is shown in Fig. 2.6. By selecting the state variables of the position of the mass as $\mathbf{q} = (\theta_r, \theta_p, r)$, where r is the -variable- length of the "leg" that connects the foot and the CoM. θ_r and θ_p are the angles between r and the projection of r to the xz-plane and xy-plane respectively, the position is described uniquely. To express these variables in Cartesian coordinates, (x, y, z), someone has to follow some basic trigonometry. For the x coordinate, it should be observed the orthogonal triangle formed by the projection of r on the yz-plane, the desired x value (translated in z-direction to the heigh of the mass) and the r itself. Similarly with the y-coordinate and the xz-plane.

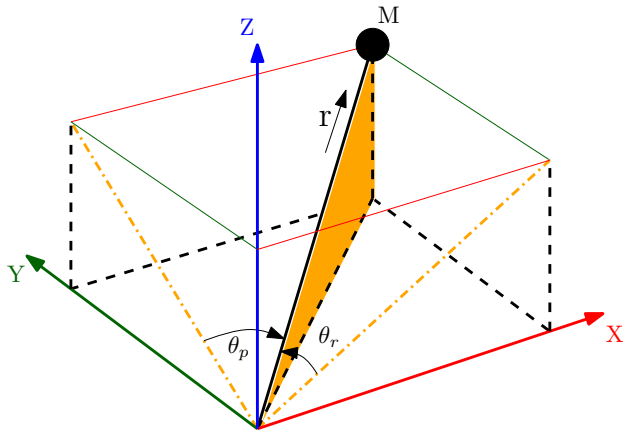


Fig. 2.6: 3D Inverted pendulum with telescopic leg.

$$x = r\ \sin\theta_p \tag{2.11}$$

$$y = -r\ \sin\theta_r \tag{2.12}$$

For the z-direction someone has to first express the projection of r in the x-y plane (let it be d) and then apply again the Pythagorean theorem.

$$d^2 = x^2 + y^2 = r^2\left(\sin^2\theta_p + \sin^2\theta_r\right) \tag{2.13}$$

$$\begin{aligned} z^2 = r^2 + d^2 = r^2 + r^2\left(\sin^2\theta_p + \sin^2\theta_r\right) \\ z = r\sqrt{1 - \sin^2\theta_p + \sin^2\theta_r} \\ z = rD \end{aligned}$$

where $D = \sqrt{1 - \sin^2\theta_p + \sin^2\theta_r}$

Let (τ_r, τ_p, f) are the torques and the force that are related with (θ_r, θ_p, r), then the equations of motion of the inverted pendulum in Cartesian coordinates, are:

$$m\begin{pmatrix}\ddot{x}\\ \ddot{y}\\ \ddot{z}\end{pmatrix} = \left(J^T\right)^{-1}\begin{pmatrix}\tau_r\\ \tau_p\\ f\end{pmatrix} + \begin{pmatrix}0\\ 0\\ -mg\end{pmatrix} \tag{2.14}$$

where the *Jacobian* can be found by partially differentiating $\mathbf{p}$ over $\mathbf{q}$ (let $C_r \equiv \cos\theta_r$, $C_p \equiv \cos\theta_p$, $S_r \equiv \sin\theta_r$ and $S_r \equiv \sin\theta_r$).

$$\mathbf{J} = \frac{\partial \mathbf{p}}{\partial \mathbf{q}} = \begin{pmatrix} 0 & rC_p & Sp \\ -rC_r & 0 & -S_r \\ -rC_r/D & -rC_pS_p/D & D \end{pmatrix} \tag{2.15}$$

Multiplying both sides of equation 2.14 with the transpose of J to avoid the inverse of it, the following is derived,

$$m \begin{pmatrix} 0 & -rC_r & -rC_rS_r/D \\ rC_p & 0 & -rCpSp/D \\ S_p & -S_r & D \end{pmatrix} \begin{pmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{pmatrix} = \begin{pmatrix} \tau_r \\ \tau_p \\ f \end{pmatrix} - mg \begin{pmatrix} -rC_rS_r \\ -rC_pS_p/D \\ D \end{pmatrix} \tag{2.16}$$

from which, using basic algebra, the dynamics along x-axis and y-axis, respectively are:

$$m\left(z\ddot{x} - x\ddot{z}\right) = \frac{D}{C_p}\tau_p + mgx \tag{2.17}$$

$$m\left(-z\ddot{y} + y\ddot{z}\right) = \frac{D}{C_r}\tau_r + mgy \tag{2.18}$$

Equations 2.17 and 2.18 are the dynamics of the pendulum without any constraints on its movement. For bipedal walking it is desired to limit its motion. The first constraint, limits the motion in a plane with given normal vectors $(k_x, k_y, -1)$ and z intersection z_c, where z_c is the heigh of the center of mass. By applying this constraint to equations 2.17 and 2.18, the dynamics become:

$$\ddot{x} = \frac{g}{z_c}x + \frac{k_2}{z_c}\left(x\ddot{y} - \ddot{x}y\right) + \frac{1}{mz_c}u_p \tag{2.19}$$

$$\ddot{y} = \frac{g}{z_c}y + \frac{k_1}{z_c}\left(x\ddot{y} - \ddot{x}y\right) - \frac{1}{mz_c}u_r \tag{2.20}$$

where u_p and u_r are virtual inputs that applied to compensate the input nonlinearity.

$$\tau_p = \frac{C_p}{D}u_p \tag{2.21}$$

$$\tau_r = \frac{C_r}{D}u_r \tag{2.22}$$

In the case the plane of constraint is parallel to the x-y plane (which means the biped walks on a flat terrain), $k_x = 0, k_y = 0$ the equations now become:

$$\ddot{x} = \frac{g}{z_c}x - \frac{1}{mz_c}u_p \tag{2.23}$$

$$\ddot{y} = \frac{g}{z_c} y - \frac{1}{m z_c} u_r \tag{2.24}$$

If someone wants to explore the natural dynamics of the 3D Linear Inverted Pendulum, all he has to do is to apply to the equations 2.23 and 2.24 the constraint of zero input torque and he gets:

$$\ddot{x} = \frac{g}{z_c} x \tag{2.25}$$

$$\ddot{y} = \frac{g}{z_c} y \tag{2.26}$$

Equations 2.25 and 2.26 are *independent linear equations.*

2.3.1 ZMP from Linear Inverted Pendulum Model

In Fig. 2.7 it is shown a model called *Cart-Table Model.* The cart with mass m, corresponds to the CoM of the biped and since it can move only along the table, the movement is linear, such as the 3D-LIPM with the constraint $k_x = k_y = 0$. In that case, the torque τ around point P is expressed as:

$$\tau = -mg\left(x_{CoM} - p\right) + m\ddot{x}_{CoM} z_{CoM} \tag{2.27}$$

where p is the position of the ZMP and by the definition of the ZMP, τ must be equal to zero. Therefore:

$$x_{zmp} = p = x_{CoM} - \frac{\ddot{x}_{CoM}}{g} z_{CoM} \tag{2.28}$$

$$y_{zmp} = y_{CoM} - \frac{\ddot{y}_{CoM}}{g} z_{CoM} \tag{2.29}$$

2.4 Capture Point

J. Pratt and Tedrake [13] introduced a new *velocity-based stability margin*, that they call *Capture Point.* Some definitions first:

Capture State

The state in which the kinetic energy of the biped is zero and can remain zero with suitable torques.

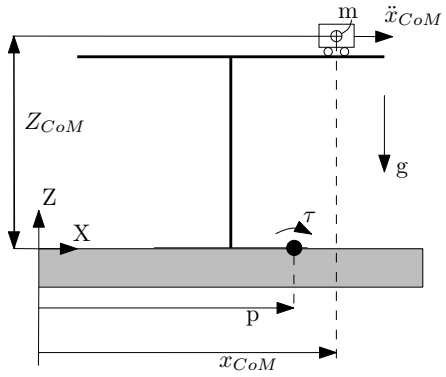


Fig. 2.7: Cart-table model

Capture Point

For a biped in state x, a *Capture Point*, p is a point on the ground where if the biped covers p, either with its stance foot or by stepping to p in a single step, and then maintains its *Center of Pressure* to lie on p then there exists a *Safe Feasible Trajectory* that ends in a Capture State.

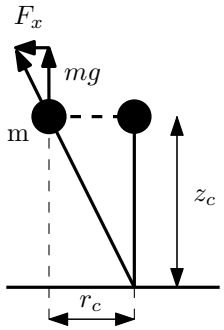


Fig. 2.8: Estimation of the location of the Capture Point based on the Linear Inverted Pendulum Model.

The derivation of estimates of the location of Capture Point is done using the LIPM that was discussed in 2.3. The ground reaction force can only act though the line that connects the Center of Pressure (CoP) and the Centero of Mass (CoM). The vertical component of this force is equal (in magnitude) to mg and F_x is the horizontal component. It is clear also from Fig. 2.8, that similar triangles are formed and therefore:

$$\frac{F_x}{mg} = \frac{x}{z_c} \rightarrow F_x = \frac{mg}{z_c} x \tag{2.30}$$

where x is the distance from the mass to the Capture Point. Recall here that the CoM can move with constant height ($z_c = const$), then energy absorbed while moving above the Capture Point will be the integral of the force times the displacement:

$$E = \int_0^{r_c} F\, dx = \frac{mg}{z_c} \int_0^{r_c} x\, dx = \frac{mg}{2z_c} r_c^2 \tag{2.31}$$

Equating initial and final energies,

$$\frac{1}{2}mv^2 = \frac{mg}{2z_c}r_c^2 \tag{2.32}$$

and solving for r_c

$$r_c = v\sqrt{\frac{z_c}{g}} \tag{2.33}$$

The above estimate assumed that the swing leg could instantaneously arrive at the Capture Point. If there is an estimate on the time remaining for swing, the Capture Point can be predicted using the *Linear Inverted Pendulum* equations 2.25 and 2.26. The closed form solution for the x-coordinate is then [13],[18],

$$x = \frac{1}{2}\left(x_0 + \frac{v_0}{w}\right)\ e^{wt} + \frac{1}{2}\left(x_0 - \frac{v_0}{w}\right)\ e^{-wt} \tag{2.34}$$

$$\dot{x} = \frac{1}{2}\left(wx_0 + v_0\right)\ e^{wt} + \frac{1}{2}\left(-wx_0 + v_0\right)\ e^{-wt} \tag{2.35}$$

where $w = \sqrt{\frac{g}{z_c}}$

The equations for y are identical given the proper substitutions. *Given the estimated swing, the position and velocity of the CoM at the end of the swing, can be estimated by Equations 2.34 and 2.35.*

Chapter 3

State of the Art

3.1 Honda's Asimo Humanoid Robot

Up to the date of this project, the most advanced humanoid robot, is *Asimo* from Honda. Asimo is a fully humanoid robot with 57 degrees of freedom (DoF). All the actuators are servomotors with harmonic gears. Its height is 130 cm and it weights 50 Kg. Asimo is able to walk with maximum speed of 1.7 kph and it can also run with up to 7 kph. It is able to change the walking cycle as well as the stride size. It uses rechargeable 51.8 V Lithium Ion Battery and is able to operate for maximum 1 hour.

The most challenging part of controlling a biped is it that the system is underactuated. What that means, is that the robot has more degrees of freedom than actuators. This has as a consequence, that the robot can not produce arbitrary accelerations in order to follow arbitrary trajectories. As an example, consider the planar robot of Fig. 3.1. This robot, cannot produce any torque to the indicated point (no actuators on the toes).

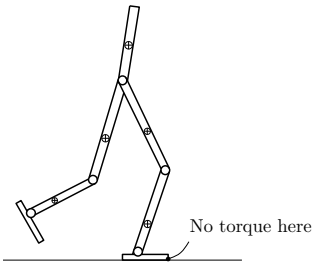


Fig. 3.1: Planar robot with 6 actuators and -at least- 7 DOF

This scenario is clearly pointing to the ZMP. In fact, the robot in Fig. 3.1 is in the *single support* state, and if the calculated ZMP lies outside of the support polygon, which is the area of the foot that is in contact with the floor, then the actual ZMP will be on the edge of the foot. As explained in [36] this will result to uncompensated moment from the ground reaction force and moment, and the robot will experience a torque on that point.

The basic idea of ASIMO's algorithm is based on that fact. If someone assumes that the foot

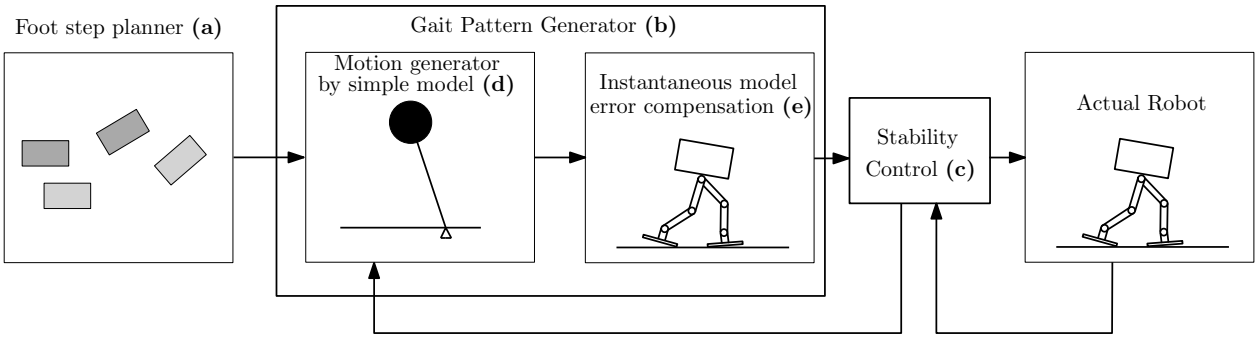


Fig. 3.2: Honda's ASIMO walking system overview

is not able to slip or move from the floor, in other words, its bolted to the floor, then the system becomes fully actuated. In order to fulfil this assumption, they continuously estimate the danger of foot roll by measuring the ground reaction forces. This is achieved with a "6-axis Foot Area Sensor"[1] that is installed on each ankle. This is a very accurate (and expensive) sensor that can measure the exact forces and moments that were discussed in Section 2.2.2. With the biped as a fully actuated system, they carefully design the desired trajectories (for example with the knees always bended to avoid singularities) and with use of adaptive trajectory tracking control (high feedback gains) they manage to design an overall impressive bipedal walking gait.

In Fig. 3.2 the system overview is shown as this was presented in [32]. By defining the *gait pattern* as a set of trajectories for the desired ZMP, the feet and the upper body, the system is then described in the following steps:

1. A step position and duration is given from the *Foot step Planner*, Fig. 3.2(a).
2. Given the parameters above, the desired ZMP and feet trajectories are designed. Then they design the upper body trajectory which satisfies the desired ZMP trajectory without causing the upper body to diverge 3.2(b).
   (a) Generate a gait pattern from a approximate dynamics model using estimate of the future model state 3.2(d).
   (b) Compensate for the dynamics error due to the approximate dynamics model 3.2(e).
   (c) Feed the gait pattern into the real robot, and stabilize it while it is following the gait pattern 3.2(e).

Each one of the blocks of Fig. 3.2 are described in detail in [35][33][34].

There are more bipedal robots implemented based on the same concept. Two of them worth noticing.

REEM-C for PAL Robotics[2] is a humanoid robotics research platform. REEM-C is 1.65m tall and weights 89 Kg. It is a ROS[3] based platform that is able to walk stable and smooth with speed

[1] asimo.honda.com/asimo-specs/
[2] pal-robotics.com/en/products/reem-c/
[3] Robotics Operating System

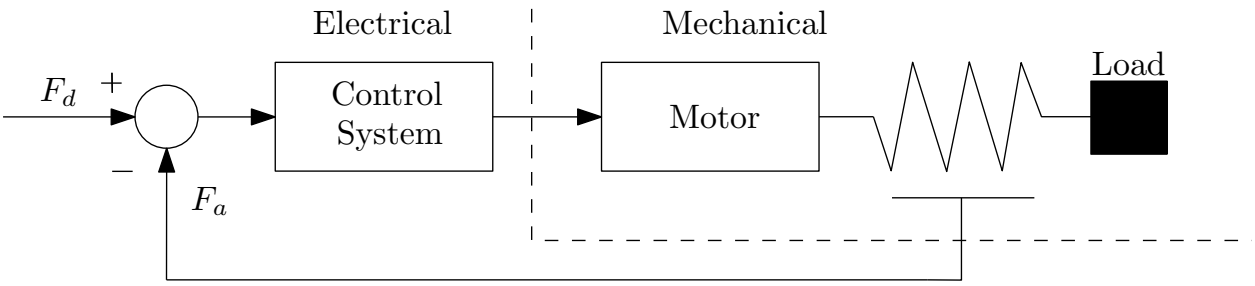


Fig. 3.3: Series-Elastic actuator block diagram

up to 1.5 kph and its battery can provide 3 hours of walking time. It runs with two *i5* computers running on *Ubuntu 12.04 LTS* and *Xenomai core* for real-time operations.

NAO is a robot by Aldebaran[1]. NAO is a 58cm robot that has already 7,000 sales. Its small size and mostly its cost, are probably the main reasons of making the NAO the first affordable choice for researchers and of course individuals. NAO is supported with its operating system, the *NAOqi OS*. It has 25 DoF and its inertial unit enables it to maintain its balance and to know whether it is standing up or lying down.

3.2 M2V2

There are several other bipedal robots that are State of the Art on dynamic balancing. Recently, *Boston Dynamics*[2] released a video of an impressive bipedal robot, the "*next generation of ATLAS*". This robot is able to walk on a rough terrain outdoors while its maintaining its balance even after large perturbations, to recover from pushes and to lift a box of 10 Kg of weight. Most likely, this robot is the State of he Art on dynamic balancing to date, but its recent release, don't allow to find any more information to support that statement.

What is important though, is that this biped is obviously not relying on the ZMP stability margins, as ASIMO. This is clear from the fact that the foot during the single support state, is not in full contact with the floor at all times.

As it was discussed on Section 2.4, another approach is the *Capture Point*. During the single support state, the swinging leg, is not interested on where "exactly" to place the foot at the end of the swing (unlike the carefully designed trajectories of the ZMP technique) but rather on placing the foot in an area, that will ensure the capturing of the kinetic energy either for the biped to stop or in order to perform another step.

This idea was introduced by J. Pratt and Drakunov [24] Pratt, as the leader of a research team consisted of several organizations (IHMC, MIT Leg Laboratory, and Delft University, among others)[3], developed a biped robot called **M2V2**[25].

[1] www.aldebaran.com/en/cool-robots/nao
[2] www.bostondynamics.com
[3] robots.ihmc.us/humanoid-robots/

M2V2 has 12 DOF, three at each hip, one at each knee and two at each angle (a typical arrangement for most of the bipeds). Since it was made to study bipedal walking, there is no DOF on the upper body. The biggest advantage of this robot is its actuators. The majority of biped robots today, are using stiff position controlled actuators. This actuators require a set of desired positions that follow a given trajectory. This leads to less robustness to disturbances. Force-controllable actuators on the other hand, allow low impedance algorithms that result in efficient and graceful walking that is robust to disturbances and rough terrain. M2V2 uses *Series Elastic Actuators* [23] to achieve force control. Fig. 3.3 shows a diagram of such an actuator.

"In Series Elastic Actuators, a spring is placed in series with the output of a motor and gear train. The output force of the actuator is then dependent on the compression of the spring, governed by Hooke's Law (F=kx). By servoing the compression of the spring via feedback control, the output force is thereby controlled. Hence the spring turns a position controllable device, such as a motor and gear train, into a force controllable device"[25].

M2V2 has various sensors that are read from a PC104 computer system at a rate of 1000Hz. These sensors are:

- 12 actuator sensors measuring actuator force, from which joint torque is calculated.
- 12 actuator sensors measuring actuator position, from which joint position and velocity is calculated.
- 1 Inertial Measurement Unit (IMU) to measure the body orientation.
- 4 foot switches (On or Off) monitoring the condition of the foot ("On" the ground or "Off" the ground).

" The PC104 system runs Solaris 10 and Real Time Java, both available from Sun Microsystems. The control loop executes at 1000 Hz, reading the sensors of the robot and determining joint torques on each control cycle. The PC104 system consists of a number of stacked boards including the main processor, quadrature encoder-decoder boards for reading the encoders at the joints, and an analog Input-Output board for reading the inertial sensor, and a PWM output board for sending current commands to the motor amplifiers. "

Chapter 4

Poppy Humanoid Robot - UPC version.

On this chapter, an introduction to the original humanoid robot will be given and afterwards, the modification that were made to meet the designs to this project's requirements.

4.1 Poppy platform.

"Poppy is an open-source platform for the creation, use and sharing of interactive 3D printed robots" [1]. One of their "creatures" is the *Poppy Humanoid Robot*[20] and is shown in Fig. 4.1

Poppy is a small size (84cm) humanoid robotic platform designed for scientific experiments on biped locomotion and human-robot interaction. The design was conceived to follow these goals:

- Anatomical proportion are bio-inspired.
- Large sensorimotor-space.
- Articulated spine.
- Light-weight.
- Small feet with compliant toes.
- Semi-passive knees.
- Bio-inspired bended thigh.

Two of the most interesting features of the Poppy Humanoid design (from the biped locomotion point of view) are the *semi-passive knee* and the *bended thigh*.

[1] www.poppy-projet.org

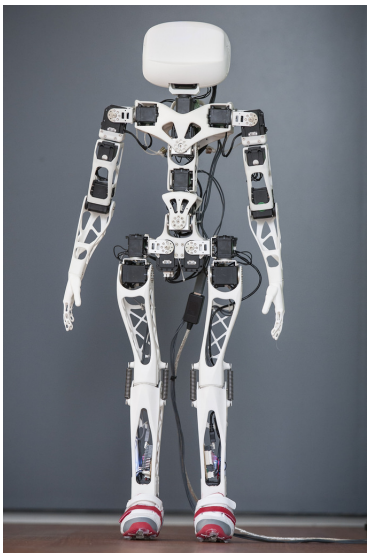


Fig. 4.1: The Poppy Humanoid robot.

The bended by 6 degrees thigh increases the stability of the biped. Recall the discussion in Section 2.3 where a biped is in balance if the projection of the *Center of Mass* (CoM) on the X-Y plane lies on the same point as the *Center of Pressure*. Being the biped in the early stage of the *Double Support* state (the swinging leg landed), then the biped must transfer the CoM from the last stance foot to the next stance foot, in order to perform a new step. During this lateral movement of the CoM at the *Double Support* state the effort is reduced because of the design of the thigh, as the feet are closer to each other and the CoM will track a smaller trajectory, Fig. 4.2[1].

The *semi-passive knee* is based on on the use of additional springs in parallel of the joint actuation. These springs have been design to participate in the leg dynamic during two main phases:

- They help to keep the leg straight during the support phase without any motor control.
- During the swing phase, they participate to the flexion of the leg.

These two modes are passively switched by the actual knee angle.

As it was also mentioned in the *Motivation* part of this project, even though Poppy Humanoid is much more affordable than most of the biped platforms, the cost is still considerable high. The main reason for that, is the actuators that are used. Poppy, as a fully humanoid robot, has 25 actuators. 19 of them are the *MX-28AT*, 4 of them the *MX-64AT* and 2 of them the *AX-12A*, all from *Dynamixel*[2] company.

The team behind Poppy, they also developed a *Python* library, called *Pypot*, that helps the users to control the platform. This library provides low-level access to motors and sensors and allows of creation of complex behaviours by combination of independent primitives.

[1] Picture was taken from [20]
[2] www.robotis.com/xe/dynamixel_en

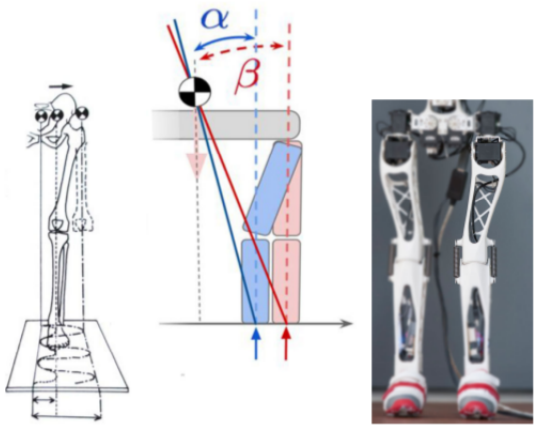


Fig. 4.2: Effect of the humans' bended femur on the biped locomotion.

They also have an active community [1] where users contribute with their ideas and their implementations, all in an open source environment. As e result of that, Poppy is integrated in the *v-rep* simulator, as well as to the *Gazebo* simulator.

4.2 Poppy UPC

The Poppy Humanoid robot was found to be very interesting to study bipedal walking algorithms but the use of these expensive servo motors is making it quite expensive. Therefore, the first objective was to reduce the cost of making a biped platform, by choosing cheaper motors and modify them to maintain the quality and the performance to the levels of the original ones (if not better).

4.2.1 Motors

The main characteristic that was desired to maintain as close as possible to the Dynamixel motors, was the *stall torque*. It was not made any study about how powerful the motors need to be (dynamics of the biped, static forces etc), either by Poppy team or on this project. Therefore the safest choice would be to go close to the "working example". All the comparisons were made with the *MX-28* motor, as these motors are the majority on the Poppy configuration. The stall torque of this motor, running at 12V (the operating voltage of Poppy) is at 2.5 N.m . The motor that is using is one from the *Maxon RE-MAX* series (without specifying which one). The original idea was to buy a single motors and gears, and design a 3D-printed case to host this mechanical configuration and the electronics needed to drive the motor, but as the project was already quite ambitious, this idea was abandoned to avoid over-complicating the project. The only available choice then, was to use one of the typical RC-servo motors. The one chosen based on the comparison with its characteristics and the Dynamixel one, was the *Hitec, HS-7954SH*. Table 5.1 shows the characteristics of both motors.

[1] forum.poppy-project.org

	Dynamixel MX-28AT	Hitec HS-7954SH
Operating Voltage	12V	7.4 V
Stall Torque	2.5 N.m	2.84 N.m
No-load Speed	55 rpm	83 rpm
Weight	72g	65.20g
Resolution	0.088°	0.7°
Reduction Rate	193:1	284:1
Operating Angle	0°-360°	-90°- 90°
Max Current	1.4A	2.6A
Material	Metal Gears and Plastic Body	Steel Gears and Metallic body
Price	$240	$100

Table 4.1: Comparison between the Dynamixel and the Hitec motor.

First thing that can be observed, is that the Hitec motor, can produce more torque and can run faster than the Dynamixel motor. That is probably because the Hitec servo is using a core-less motor which has less friction. Also, the fact that Dynamixel is operating at 12V while Hitec at 7.4V can explain the big difference on the *max current*. This high current of the Hitec motor is probably the reason of the metalic body[1] in order to act as a heat-sink. Surprisingly though, the Hitec motor is lighter than the Dynamixel one.

The parts that Dynamixel is superior over the Hitec motor, are the electronics and the provided controller. Of course this was expected as the one was made for robotics application while the other one for use on RC-airplanes. Dynamixel is using a contact-less magnetic encoder, to read the position of the shaft, hence the very high accuracy of 0.088°. It also has an embedded micro-controller that is running the motor controller. The controller is probably a typical *PID* controller, as the user has the option to re-set these gains. Apart of these gains, the user can program the limits on the torque, the rotating angle, the velocity and others. Someone can also set the reference of the controller, either it is a position-velocity reference or a torque reference. Finally the Dynamixel motor has also an embedded temperature sensor in order to protect the motor from over-heating.

Since the Hitec motor has barely any of this features the decision was made to modify the Hitec servo. To do that, all the electronic parts were took apart, along with the potentiometer that was working as a position sensor. In other words, what was left, was only the motors with the gears and the case. Instead a new *PCB* was designed which is hosting the driver of the motor and an embedded micro-controller to control the motor and to handle any desired communication with the motor.

This part of the design, was covered from previous work related to this project. It will not be presented here in detail but rather as a resume of the final result, in order to compare it with the Dynamixel motor. Fig. 4.3a and 4.3b show the resultant design of the main board and Fig 4.4a and 4.4b shows the daughter board of the sensor. Finally Fig. 4.5 shows the real printed board assembled with the components and the cables.

[1]Only the part of the case that is in touch with the motor is metallic.

(a) Top view **(b)** Bottom view

Fig. 4.3: The main board that host the driver and the micro-controller

(a) Top view **(b)** Bottom view

Fig. 4.4: The sensor board

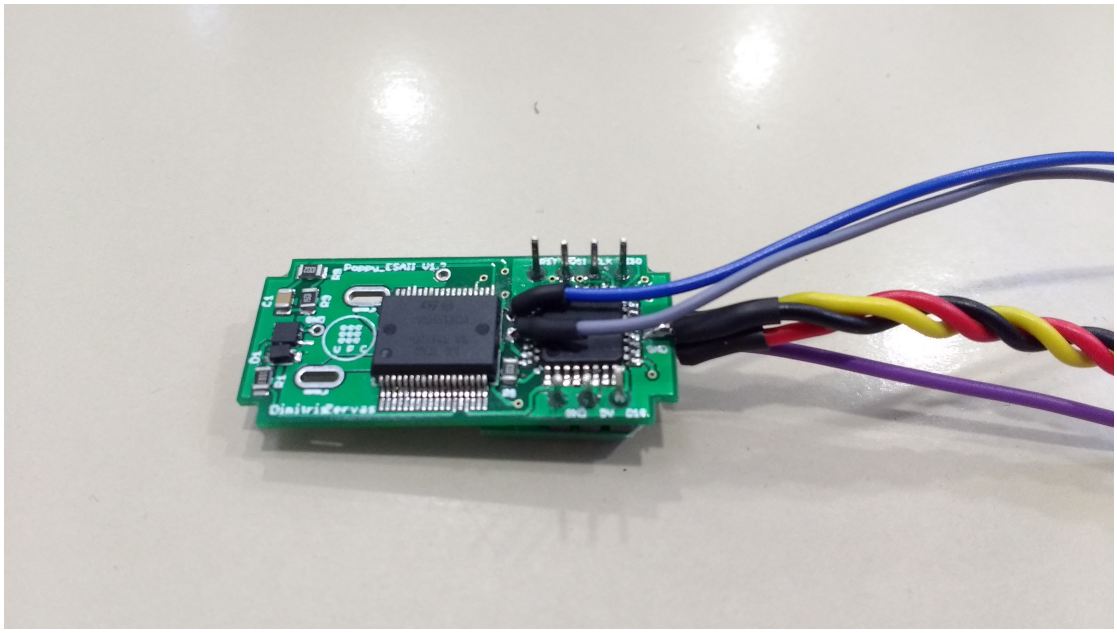


Fig. 4.5: The final result of the design og the board.

Driver

The driving of a dc-motor is implemented with an H-bridge. It was selected to use a fully integrated H-bridge on a chip instead of creating one with separate mosfets. The selected chip is the **VNH5180A-E**. Some of the reasons for this choice had to do with its characteristics,

- Output current: 8A
- 3V CMOS compatible inputs
- Undervoltage shutdown
- Overvoltage clamp
- Thermal shutdown
- Cross-conduction protection
- Current and power limitation
- Very low standby power consumption
- PWM operation up to 20 KHz
- Protection against loss of ground and loss of V_{CC}
- Current sense output proportional to motor current
- Output protected against short to ground and short to V_{CC}

with the main reason to be the high output current (8A is more than enough for the specific motor) and the integrated current sensor. It does not have a temperature sensor like the Dynamixel motor, but it does have a thermal shutdown, which is important to protect the motor from overheating. Finally, it also has two *diagnostic* pins that can be used to detect the reason of the failure (if occurred).

Position Sensor

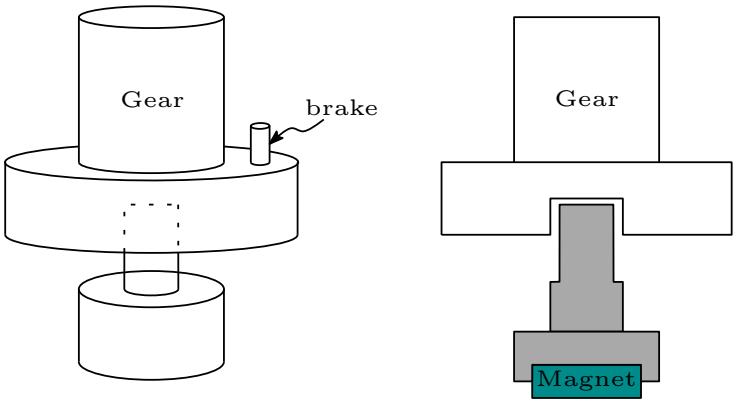


Fig. 4.6: The part of the potentiometer that was used to place the magnet.

The *Hitec* motor is using as a position sensor a potentiometer. Since the accuracy of such a sensor is far from satisfactory this sensor was removed. Instead a contact-less magnetic encoder was added to the system. The sensor used is the **AS5145** from *AMS*[1]. This sensor has a 12-bit

[1] ams.com/eng/Products/Magnetic-Position-Sensors/Angle-Position-On-Axis/AS5145H

resolution and therefore has the same accuracy as the Dynamixel motor. It transfers the data through a **S**ynchronous **S**erial **I**nterface (SSI) port. The *SPI* port of the micro-controller is used to read these data (practical SSI and SPI are the same).

This sensor, needs to have a rotating magnet close to its proximity, in order to measure the angular displacement of the magnet. It is crucial then, the placement of the magnet inside the motor. Fig. 4.6 on the left, it shows the original setup, where the tip of the potentiometer was attached in a *pocket* inside the gear that is attached to the axis of rotation of the motor (it also shows the mechanical brake that was removed). Originally, this tip of the potentiometer was removed in order to glue on it the magnet. As this was not a very stable solution, later on, a similar part was printed with the difference that it has a pocket for the magnet on the other side of the tip, as it is show in Fig. 4.6. When the designed PCB is mounted on the board, the daughter board (the sensor) fits right above the magnet.

Micro-controller

The micro-controller that was chosen to be embedded on the designed PCB is the *ATmega328P-AU*[1] from *ATMEL*. It is the same micro-controller used by the Arduino UNO[2] board. The reason of this choice is obvious. Is the popularity of the Arduino platforms and most important its community, that would make the programming of the micro-controller more attractive. Of course, the requirements of a motor control were taken under consideration before this decision and it was concluded, that the specific chip can cover them.

Arduino Uno is using this chip with a 16Mhz crystal but the maximum crystal that can be safely used is at 20Mhz (reports are made of running up to 24Mhz). A 20Mhz crystal will increase the speed of the micro-controller by 25% and that can be utilised in order to reduce the interval of the *time sampling*. At the first glance this looks optimal for this project but in reality it comes with one drawback. This has to do with the Serial communication baud rate. With a 16 Mhz crystal, the maximum baud rate (with 0% error) is at 1.000.000 bps, while with a 20Mhz crystal the maximum is at 500.000 bps. Therefore the trade off was between faster calculation and slower communication or slower calculations and faster communication. The decision factor was the PWM frequency that the 8-bit timers can produce in each case.

The PWM signal is used as an input to the driver and its duty cycle controls the voltage applied to the motor. Recall that the driver can accept PWM frequencies up to 20 Khz. With a 16 Mhz crystal, the maximum PWM frequency from an 8-bit timer, is at 7.812 Khz. There is no problem for the driver to operate with that signal. The problem is that, when the motor is applying torque and the load is big enough to stop any movement of the shaft, there was observed a 'high frequency' buzzing. It was annoying enough just from one motor so it is easy to imagine the noise from 12 motors. To solve this problem, the frequency of the PWM must be increased (so that it cannot be heard from the human ear). With the 20Mhz crystal, the maximum PWM frequency is around 10Khz. With that frequency, the buzzing is still there, but barely someone can hear it. That is the reason why the ATmega328P with a 20Mhz crystal was implemented.

NOTE: When the maximum PWM frequency is referred, it is meant the maximum frequency

[1] www.atmel.com/devices/atmega328p.aspx
[2] www.arduino.cc/en/Main/ArduinoBoardUno

without changing the resolution of the PWM from 0-255. 8-bit timers can be configured to achieve PWM signals with higher frequency than 10Khz (with both crystals) if the resolution of the PWM is lower than the 0-255.

One of the goals of the project is that the user can upload new sketches (code) to the micro-controller that is embedded on each motor. With the implemented design, this can be done with the Arduino IDE. In general, there are two ways to re-program an Arduino. This is done either with **I**n **S**ystem **P**rogramming (ISP) or through Serial. On both cases, a *programmer* is needed. The chosen way decided to be the Serial one for reasons that are explained in Section 5.1. In order for the micro-controller to accept the new program through its serial port, a new boot loader was needed as now it runs at 20Mhz. Someone can use the Arduino IDE to "make" the new boot loader. All it has to be done, is to modify the *make* file in the *bootloader* file of the Arduino IDE and to create a new board entry. Listing 4.1 shows the added lines in the make file and Listing 4.2 the added lines in the *boards.txt file*.

Listing 4.1 The lines added in the MakeFile

```
# File path: /home/'user-name'/arduino-1.6.7/hardware/arduino/avr/bootloaders/atmega

uno20: TARGET = uno_20Mhz
uno20: MCU_TARGET = atmega328p
uno20: CFLAGS += '-DMAX_TIME_COUNT=F_CPU>>4' '-DNUM_LED_FLASHES=1' -DBAUD_RATE=57600
uno20: AVR_FREQ = 20000000L
uno20: LDSECTION  = --section-start=.text=0x7800
uno20: $(PROGRAM)_uno20.hex

uno20_isp: uno20
uno20_isp: TARGET = uno_20Mhz
uno20_isp: MCU_TARGET = atmega328p
uno20_isp: HFUSE = DA
uno20_isp: LFUSE = FF
uno20_isp: EFUSE = 05
uno20_isp: isp
```

Listing 4.2 The lines added in the boards.txt file

```
File path: /home/'user-name'/arduino-1.6.7/hardware/arduino/avr

##############################################################

atmega328_20.name=Arduino Uno @ 20MHz

atmega328_20.upload.tool=avrdude
atmega328_20.upload.protocol=arduino
atmega328_20.upload.maximum_size=32265
atmega328_20.upload.speed=57600

atmega328_20.bootloader.tool=avrdude
atmega328_20.bootloader.low_fuses=0xFF
atmega328_20.bootloader.high_fuses=0xDA
atmega328_20.bootloader.extended_fuses=0x05
atmega328_20.bootloader.unlock_bits=0x3F
atmega328_20.bootloader.lock_bits=0x0F
```

```
atmega328_20.bootloader.file=atmega/ATmegaBOOT_168_uno20.hex

atmega328_20.build.mcu=atmega328p
atmega328_20.build.f_cpu=20000000L
atmega328_20.build.core=arduino
atmega328_20.build.board=AVR_UNO
atmega328_20.build.variant=standard
```

In a terminal, after navigating to the folder of the MakeFile, running the command:

```
make uno20 AVR_FREQ = 20000000L
```

will generate the *ATmegaBOOT_168_uno20* hex file, which is the required boot loader.

It is **important** to remember, that in order to burn the boot loader to the chip, an ISP programmer is needed. After the boot loader is burned to the chip, then the user can upload his sketches using the Serial programmer. Also, when the motors are mounted to the biped, the ISP pins are not exposed (no ISP cables go out of the motor). For the unlike scenario of someone want to burn a boot loader, he has to un-mount the motor and open the back case to access the ISP pins.

Programming the micro-controller

As it will be explained in Section 5.2 the motors are communicating through the Serial pins. In order someone to program the micro-controller he needs two things. A serial programmer and the *RX, TX* and *Reset* pins. From the motor case, three cables are coming out that are connected to those exactly pins. The serial programmer used on this project, was one based on the FTDI chip (see Fig. 5.1a), bought from the local electronics store for $5. The Tx pin of the programmer must be connected with the Rx pin of the micro-controller and the Rx pin of the programmer must be connected with the Tx pin of the micro-controller. The DTR pin of the programmer must be connected to the RST pin of the micro-controller, through a simple *auto-reset* circuit, similar to the one on the Arduino Uno board. With this configuration, the user can use the Arduino IDE to upload sketches as he would do with a normal Arduino Uno board (simply by clicking on the "Upload" button).

4.2.2 Comparison between Dynamixel motors and modified Hitec motors

At this point, the comparison between the two motors can be done. The electrical characteristics are already compared on Table 5.1.

As far as the electronics part, both motors are using the same type of sensor (if not the exactly same sensor) and therefore both of them have the same accuracy. Since the Dynamixel is providing a torque feedback it is assumed that it has an implemented sensor (a shunt resistor probably). The Poppy-UPC motors have a current sensor as part of the driver of the motor. The driver it self it provides various protections and diagnostic pins to inform the user what went wrong. The

Dynamixel motor, again based on the data-sheet, it has an embedded temperature sensor but the rest of the protections seem to be by software.

Both motors have an embedded micro-controller. The difference is that the Poppy-UPC motor is giving the opportunity to the user to program his own controller. A feature that gives great versatility to any servo-motor related project. To achieve this feature though, 3 cables are exported from the case of the motor (plus 3 for the power supply). Dynamixel motor on the other hand, it uses only 3 cables for both power supply and the communication.

As for the controller, a comparison is based solely on the user of the Poppy-UPC motors, as he can upload any controller he desires.

The mechanical part of the Poppy-UPC motors are presented on the next subsection, but a comparison with the Dynamixel motors is obvious. Dynamixel have a commercial quality outer case specifically designed to be easy to mount with other rigid parts. Hitec motors on the other hand do not provide such a feature and in order to fit them on the platform, 3D parts were designed.

Fianlly, since the author did not had any MX-28 motor on his possession the quality of the two motors can not be compared. Over all thought it would be fair to state that so far, the cheaper and modified motor is offering at least the same features as the Dynamixel motor. Another feature that is up to comparison, is the communication part, but this will discussed at the end of Chapter 5.

4.2.3 Mechanical Parts

On this Section, the mechanical parts of the Poppy-UPC will be presented.

Rigid bodies designs and modifications.

On this Section, the modifications and the designs on the rigid parts of Poppy made by Joan Guasch will be shown. Details about the designs will not be given as this is not work from this author.

Joan designed 3D printed parts in order the new motor to be able to be mounted to the rigid parts (it has smaller size than the Dynamixel motors). These parts are shown in Fig. 4.7a and in Fig. 4.7b are shown mounted on the motor.

In Fig. 4.8 the two motors of the ankle mounted on the leg are shown. It can also be seen the designed board that handles the power supply connections (Section 4.2.3).

This design had as a result to make the size of the motor larger than the original. For that reason, the rigid parts were also modified to be wider, in order to fit the new motors. There are two designs for the femur. One is bended like the original Poppy femur and the other is straight. The advantage of the bended femur was discussed in Section 4.1. The reason of the two designs is that the straight version is possible to be printed with the low-cost 3D printer while for the bended version the piece was printed in an external service. One bended femur was printed from a web service, and the cost was around $90 (the cheapest that was found). Under the scope of this project, it was decided to continue with the straight version, even though the advantage of the bended femur

(a) The pieces to house the motor and to connect with other parts.

(b) The pieces mounted to the motor. That way the mounting of the motor is achieved.

Fig. 4.7: The two types of sensors that are attached to the biped.

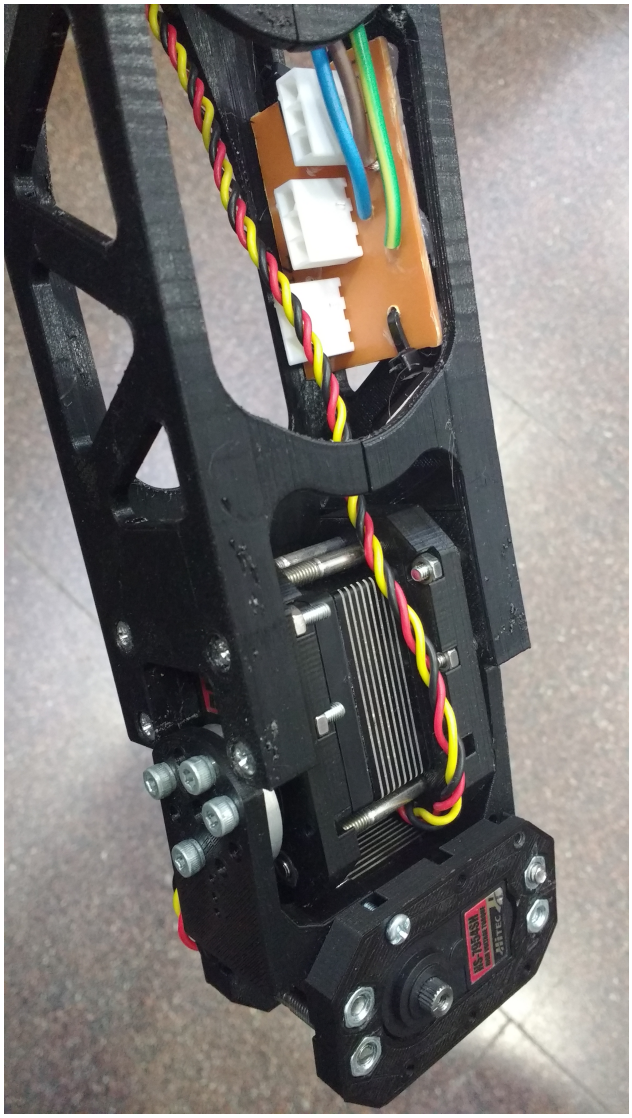


Fig. 4.8: Ankle motor mounted on the leg.

Fig. 4.9: The bended versus straight version of the femur.

is quite important. Both designs are available therefore its up to the user which one to print. Fig. 4.9 shows the two different printed versions while Fig. 4.10 shows the difference of the original Poppy's femur and Joan's modification.

Some pictures from the drawings of the complete parts that Joan designed/modified are following. The purpose of the existence of the rod as an upper body, is in order to "push" the total CoM of the biped closer to the hip area in order for the dynamics to be more realistic (human's body CoM is close to the the waist).

Power Supply Unit

To supply the motors and the sensors, a power supply unit was bought, Fig. 4.12. The maximum -continuous- current that the motor can draw was measured to be 2.6A at 7.5V. The PSU that is used, can supply up to 30A which means that if all the motors will draw the maximum current that they can ($2.6 \times 12 = 31.2A$), the PSU's protection systems will engage to protect it. However, the scenario of all the 12 motors be on full power is highly unlike, therefore it was decided not to buy a new, more powerful unit. The cost of this unit was around $60.

After the design of the boards of the motors, there was not space left to include a voltage regulator to translate the 7.5V to the *recommended* 5V of operation of the chip. For that reason, a DC/DC Step-Down (Buck) converter was used, to supply all the micro-controllers, like the one in Fig. 4.13. The drawback of this decision is that each motor needs to output 3 power source cables. One for the driver (7.5V), one for the micro-controller (5V) and of course one for the -common- ground.

Fig. 4.10: The original VS Poppy-UPC version of bended femur.

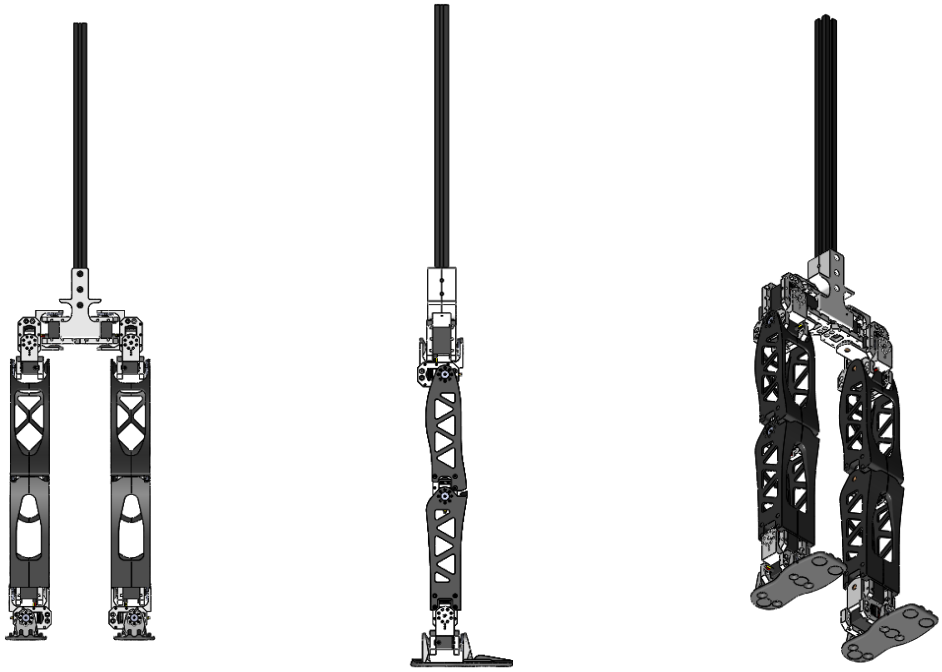


Fig. 4.11: Poppy-UPC designs.

Sensors

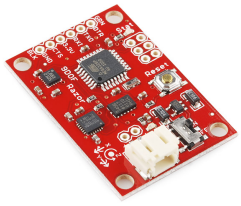
On this part of this section, the sensors used for this biped will be introduced. The magnetic encoders that read the position of each motor are *not* included, as those are part of the local loop of each motors. Basically, there are two types of sensors used, **F**oce **S**ensing **R**esistors (FSR) and and an **I**nertial **M**easurement **U**nit (IMU).

The FSR sensors are used in order to sense which of the legs are in touch with the ground. This is a useful information for any biped algorithm as it define on each state is at every moment,

Fig. 4.12: The Power Supply Unit that is used (7.5V - 30A).

Fig. 4.13: The DC-DC step down converter (7.5V to 5V).

single or *double* support. Originally it was hoped to be used also as a "noisy" measurement of the Center of Pressure but this idea was abandoned as the results were far from satisfying. To attach the sensor under the foot, two pieces were printed to ensure that the ground reaction forces will pass through the vertical axis of the sensors. Fig. 4.15a shows the frontal view of the configuration and Fig. 4.15b shows the sensors attached to the foot.

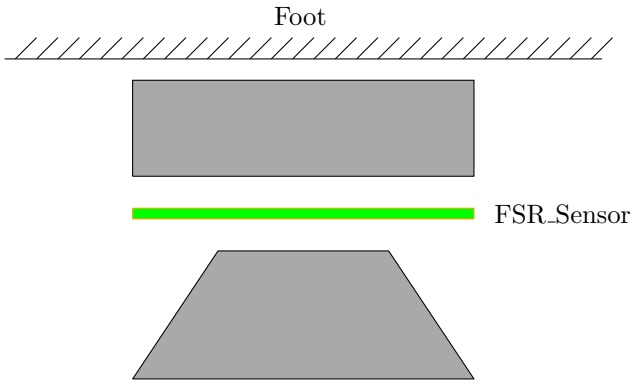


(a) The IMU sensor.

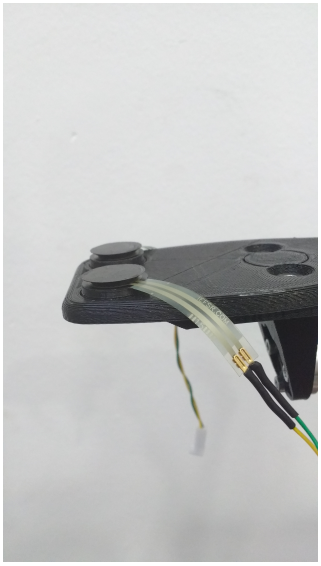
(b) Two of the FSR sensors.

Fig. 4.14: The two types of sensors that are attached to the biped.

The IMU is placed above the hip of the biped. It is used to "sense" the orientation of the robot. It is also an important measurement as it the basis for any forward kinematics algorithm. The



(a) The schematic drawing of the FSR attachment.



(b) The sensors attached on the front of the foot.

Fig. 4.15: The two types of sensors that are attached to the biped.

IMU that was used is the *Razor*[1]. Razor incorporates three sensors, an accelerometer, a gyroscope and a magnetometer. The outputs of all sensors are processed by an on-board ATmega328 (another Arduino here) and output over a serial interface. There is also an implemented algorithm *Razor-AHRS*[2]. This is a Direction Cosine Matrix (DCM) based Attitude Heading Reference System (AHRS) with gyro drift correction based on accelerometer (gravity) vector and magnetometer (compass) vector. This result to a simple reading the serial output of the Razor in order to acquire the Euler angles.

Design of PCB boards to handle the connections

As it was described in the *Power Supply Unit* subsection, the are two power sources. One at 7.5V to supply the motors and one at 5V to supply all the micro-controllers. In order to avoid the power supply cables of each motor to travel all along the legs, 3 cables starting from PSU are passing by along each leg. That way each motor is connected to the *power supply lines* through some connectors. Two boards were designed to host the 'female' part of the connectors. These boards are placed between the femur ans the knee and between the knee and the ankle. Fig. 4.16 shows the design of this board.

The advantage of this configuration is that if a one of the motors will fail, the power supply lines will remain active, as the motors are connected, in a way, in *parallel*.

Similar boards (Fig. 4.17) where also designed for the communication cables. With the only difference, that the RST cable has a standalone connector in order to be able to connect/disconnect separately of the Tx/Rx pins.

[1] www.sparkfun.com/products/10736
[2] github.com/ptrbrtz/razor-9dof-ahrs

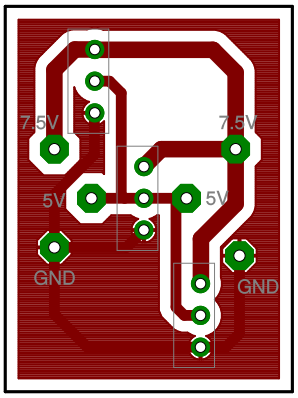


Fig. 4.16: The board where the motor connectors are attached.

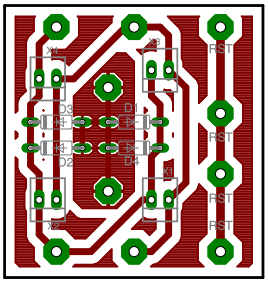
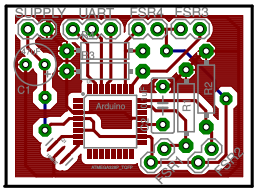


Fig. 4.17: The board of the RS-485 network.

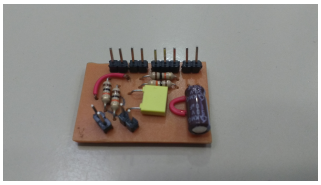
Design of PCB boards for the FSR sensors

For the reading of the FSR sensors, a small Arduino board was designed. Since this board was not complicated (from the point of view of space for the components), it was etched in the laboratory. To read the FSR sensor, an *analogRead()* call is needed for each one of them. The board is having 4 inputs for sensors, even though only three per foot are used, to give more flexibility to the user, if he decides to use 4 FSRs (for example, in case he want to try to compute the Center of Pressure with these sensors).

This Arduino board is configured to be part of the RS-485 network, as this was described in Section 5.2. Fig. 4.18a shows the design of this board and Fig. 4.18b the board populated with the components. This micro-controller is also re-programmable.



(a) The design of the FSR board



(b) The populated board

Fig. 4.18: The FSR board that reads the sensors of the foot.

4.2.4 Computer Unit

The main Computer unit was chosen to be the Odroid C1 platform[1], and is shown in Fig. 4.19. Odroid is practically a Quad Core Linux computer that can run Ubuntu[2] and cost $35. Some of its features:

CPU	Amlogic ARM Cortex-A5(ARMv7) 1.5Ghz quad core
GPU	Mali-450 MP2 GPU (OpenGL ES 2.0/1.1 enabled for Linux and Android)
Memory	1Gbyte DDR3 SDRAM
PSU	5V2A DC input
GPIO	GPIO/I2C/SPI/UART/ADC

The peripherals that the board os offering and are of interest for this project are:

- Two UART ports. The first one is mapped to /dev/ttyS2 which is connected to 40-pin header Pin #8 and #10. The second one is mapped to /dev/ttyS0 which is connected to Serial Console Port (see Fig. 4.19).
- Two I^2C ports, both of them on the 40-pin header. The first one is mapped to /dev/i2c-1 at the pins #3 and #5. The second one is mapped to /dev/i2c-2 at the pins #27 and #28.

The serial communication that was chosen for this project (Section 5.1) can be implemented either using directly the Tx/Rx pins of the two available ports or, since the FTDI programmer is used to program the micro-controllers, it could also be used for the communication, simply by connecting it to one of the USB ports (mapped to /dev/USB0 if available). In fact, since the Arduino IDE can be installed in a Linux machine, someone can upload the sketches to the micro-controllers, directly from the Odroid, by using the *avrdude*[3] command.

Odroid, as a Linux computer, comes with all the advantages of that OS. One of them that was found extremely useful was the *ssh* and *sshfs* remote connection as well as, the capability of running ROS[4] as easy as on a normal PC.

[1] www.hardkernel.com/main/products/prdt_info.php?_code=G141578608433
[2] www.ubuntu.com/
[3] A program for download/uploading AVR microcontroller flash and eeprom. Is installed along with Arduino IDE if it was not already pre-installed
[4] Robotics Operating System

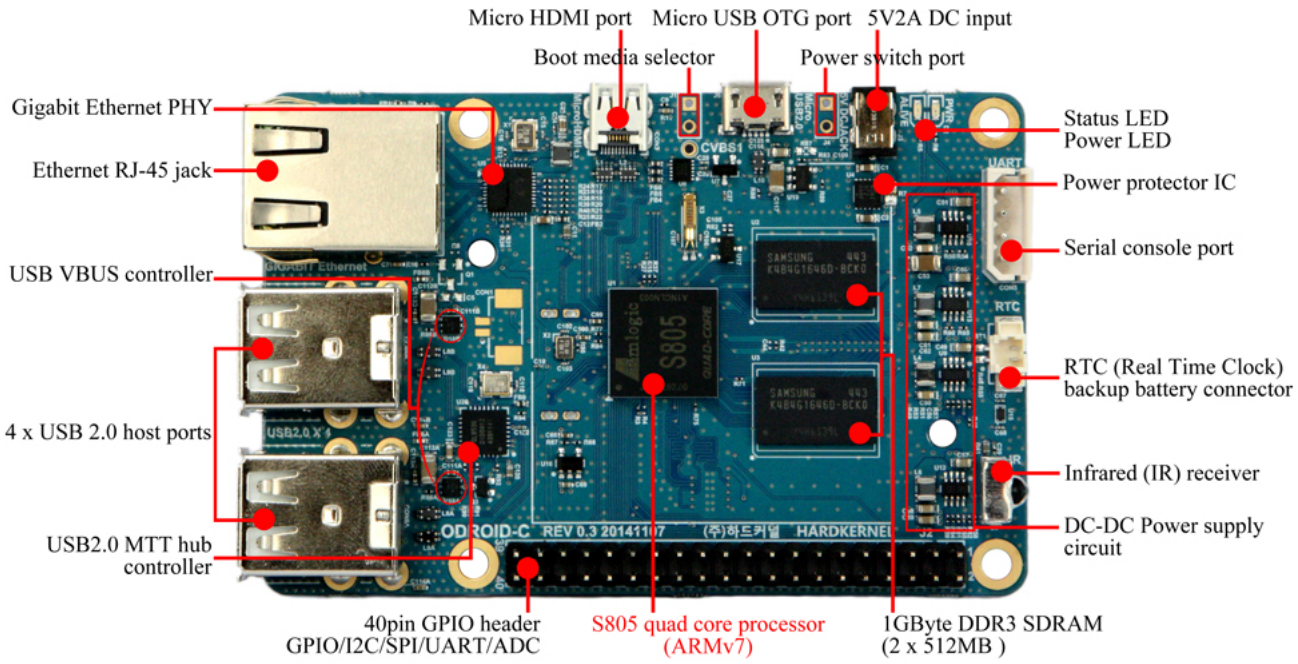


Fig. 4.19: The Odroid C1 platform with the board details.

Chapter 5

Communication

This chapter will describe the communication between the main computer unit, either this is an Odroid or a PC, and the micro-controllers of each motor. The hardware was implemented in such way that with the same configuration and minimum effort from the user, to be possible to upload new code (sketches) to the micro-controllers.

5.1 Evaluating the available options

The main idea was that the platform's main computer unit to be an Odroid board. The options that someone has to communicate with Odroid are three, I^2C, UART and SPI. Since every micro-controller on a board already uses the SPI port to read the embedded position sensor, it was decided not to overload this port with extra data transfer. Therefore the choice of the communication protocol was between the I^2C and the UART port. Even though at the beginning the communication part was implemented through the I^2C channel, eventually, the UART was chosen for reasons that are going to be explained on this section. However, a comparison of these two option shows that both of them fit equally for the needs of this project.

First of all, Odroid offers for both cases two separate ports. This is an interesting feature as instead of having 12 motors plus any sensors sending data through one port, someone can split this data flow to two ports. Using a *thread* per port to communicate with the nodes, the reading of these data an be achieved in "parallel". Therefore, Odroid offers for both I^2C and UART option, two ports, which can be utilized to decrease the load on the ports and increase the rate of communication.

As for the speed of communication, I^2C can be used (after some tweaking on the Odroid device tree) up to 400Khz, which translates to 400000 bits per second (bps). This number is known as *baud rate*. UART's baud rate on the other hand, is more flexible and speeds -at least- up to 1000000 bps can be achieved. Since an Arduino with 20 Mhz is used though, according to its data sheet, the maximum baud rate with 0% error is at 500000 bps, slightly better than I^2C.

I^2C is working in a *master-slave* manner. Even though an I^2C network is able to handle multiple

masters the Odroid's kernel, can only function as a master. This means that any node (e.g. motor) can not initiate a communication with Odroid. On the other hand, on a UART network any node can initiate a communication with any other node, but of course, care must be taken (by software) to avoid data collision.

Another feature under comparison, is the *addressing*. I^2C port has implemented hardware to issue the addressing from the master and vice versa. Another advantage of this feature, is that it provides also the information of the end of a data package [1] by hardware. On a UART network on the other hand, addressing must be checked through software. This of course implies that the data package contains one byte with the receiver's ID. Here it is worth mentioning, that the chosen micro-controller, can operate in a mode called *Multi-processor communication mode* or in short, MPC mode. On a UART network, data are transferred by data bits placed inside a frame. The frame can be consisted of a *start* bit, one or two *stop* bits, data bits (5,6,7,8 or 9 bits) and optionally a parity bit for error check. A typical frame on a UART network is one called *8N1*, which means 8 data bits, no parity, one stop bit. The micro-controller can use a filtering function that can decide if the incoming frame contains addressing information or data information. To achieve that, the network's frame must be set either with 9 data bits and 1 stop bit or, with 8 data bits and two stop bits.

" If the Receiver is set up to receive frames that contain 5 to 8 data bits, then the first stop bit indicates if the frame contains data or address information. If the Receiver is set up for frames with nine data bits, then the ninth bit is used for identifying address and data frames. When the frame type bit (the first stop or the ninth bit) is one, the frame contains an address. When the frame type bit is zero the frame is a data frame."

This is an extremely useful feature for a network such the one desired for this project. If the receiving node is not addressed, then all the data frames on the network are not processed from the CPU of this micro-controller. The receiving node also has to figure out when is the end of the data package. MPC mode can also solve this problem, by simply using a "dummy" address, common to all the nodes. A communication from a master to a slave, would consist of an addressing frame, the desired data frames and in the end the dummy-addressing frame to indicate the end of the communication.

Unfortunately, any Linux Serial API and hence Odroid, is not implemented to work with 9 data bits. There is a known round about, that uses a frame with 8 bits and two stop bits and with a specific use of some registers to manipulate the value of the first stop bit (the ninth bit that the micro-controller uses), but it was observed that the process to switch this bit was time consuming (around 10ms) and hence unacceptable for the speed of the communication that is desired. Therefore this feature of the micro-controllers can not be used.

Finally, the last feature under comparison, had to do with one of the goals of this project, *to be able the user to re-program each of the Arduinos (micro-controller) embedded in the motors.* This is mainly connected with number of cables that are going out of the case of the motor. There are two options for someone to upload new code to the micro-controller. Either through *In-System Programming* (ISP) or through *Serial Programming*, taking advantage of an installed bootloader. For both cases external hardware is needed. For ISP the simplest (and cheapest) one is the *USBasp*[14]. For UART, most of the available choices are based on the *FTDI*[6] chip which

[1] A term used to describe a set of bytes sent from the master through the network. Also known as "Serial Word".

(a) FTDI programmer **(b)** USBasp programmer

Fig. 5.1: Two options for programming the micro-controller

translates the signal coming from a USB port to UART (rx/tx) signals (also known with the term USB-to-UART). Fig. 5.1 shows these two programmers. The main difference (based on this project) is the number of cables that are using. The ISP option, uses 6 cables (MISO, MOSI, SCK, GND, VCC, RST) while for the USB-to-UART option, 5 cables (Rx, Tx, VCC, GND, RST). It is clear now the reason of comparison. A system implemented with I^2C network needs 8 cables (ISP + SCL, SDA) and a system implemented with a UART network needs 5 cables (Rx and Tx used for both programming and communicating).

Overall, it would be a fair statement that both I^2C and UART, based only on their network features and always according to the need of this project, are equal choices. Combined though with the goal of this project to give the opportunity to the user to upload his own code to the micro-controllers, the UART option seems more appropriate, only because of the number of cables that come out of the motor. Table 5.1 sums up the comparison that was analysed on this section.

	I^2C	UART
Number of ports available on Odroid	2	2
Data transmission speed	400000bps	500000 bps
Addressing handling	Hardware	Software
Number of cables used	2	2
Number of cables needed in combination with programmers	8	5

Table 5.1: Comparison between I^2C and UART network based on the project specifications.

5.2 RS-485

As mentioned in the previous section, the communication between the motors, the sensors and the Odroid was implemented through a *serial* network, in particular, implemented based on the RS-485 standard. This can be achieved either with the FTDI programmer or with the exposed Rx/Tx pins on the Odroid board. Since to reprogram the micro-controller the FTDI chip is needed, the implementation was done based on it. No modification to any code (for the communication) is

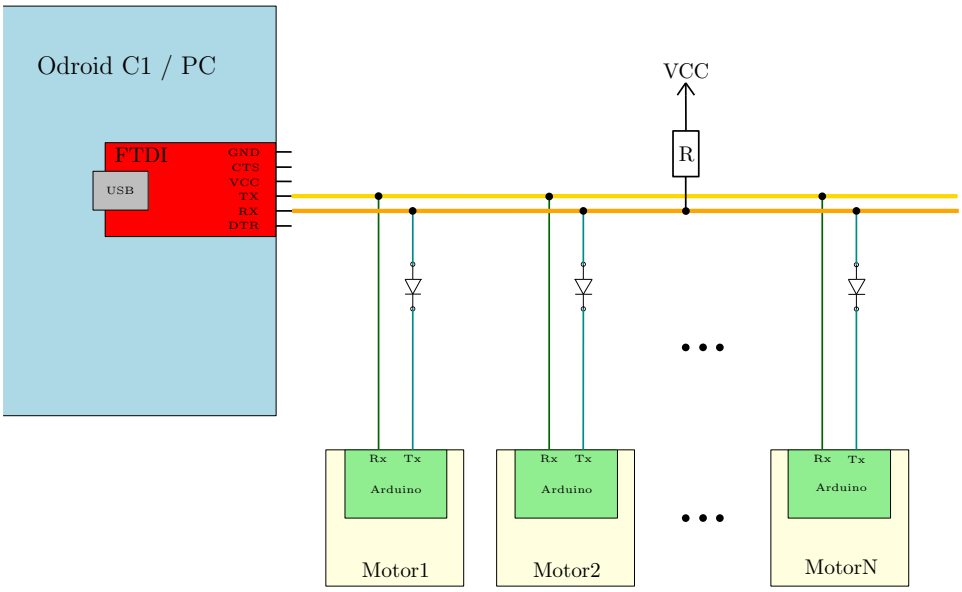


Fig. 5.2: RS-485 network topology

needed if someone wants to switch to the Odroid pins.

RS-485 enables the configuration of inexpensive local networks and multidrop communications links. It offers data transmission speeds of 35 Mbit/s up to 10 m and 100 kbit/s at 1200 m. It consists of two lines, and a connected series of point-to-point (multidropped) nodes. Fig. 5.2 shows the connection of the motors to the network.

The two lines (Rx and Tx) in idle are set to *high*. The Rx pin of each micro-controller is always functioning as in input pin. Therefore when the Odroid (master) transmitting data, this data are "*broadcasted*" to every motor (slave) connected to the network. The Tx pin of each micro-controller is always functioning as an output. As it is mentioned, RS-485 is a standard for a series of point-to-point nodes. That means that only one micro-controller can (and should) send back data to Odroid. That is the role of the diode and the pull-up resistor in Fig. 5.2, to isolate the Tx pin of each micro-controller. This is achieved by software, and is explained in detail in Section 5.4. Details on how to configure the UART port of each micro-controller as well as to issue the addressing and data-collision problems are also given in section 5.4.

Programming a micro-controller connected on an RS-485 network

In order for someone to program one of the micro-controllers connected to the network, all he has to do is to connect the RST cable to the DTR line of the programmer (using the Arduino IDE), upload the sketch and then disconnect the RST cable again.

5.3 Data Packet

Before start explaining how to configure each of the entities connected on the network, it is better to explain first the type of messages they are exchanging. This messages contains information such as the ID of the receiver instruction and data (e.g. feedback variables). This message will be referred as a *Data Packet*.

Odroid will always act as a master, and as such, it always instructs the nodes (motors or sensors). Most of the instructions will be followed by a response from the other "point". There are also instruction that are broadcasted and of course no response then is expected. The frame of the data packets is similar to this of the *Dynamixel* motors[1]. The reason for that is the popularity of these motors in the academic (and not only) area. Therefore, for someone to transfer his application to the motors of this project, it would require minimum modifications on his code.

Fig. 5.3 shows the form of the data packet transmitted by the master (Odroid) and the form of the data packet from the response of the specified node. The implemented data packets consist of:

- **FF**,**FF** Two "dummy" bytes to indicate the beginning of the packet.
- **ID** The ID of the receiver.
- **LEN** The *length* of the packet indicates the number of bytes to be checked for error (used in CRC algorithm). The *length* is defined as the number of data ($D_0 ... D_N$) plus two (the LEN byte and the INSTR or ERROR byte).
- **INSTR** This byte is used by the master to *instruct* the receiver according to its needs.
- **ERROR** This is a byte in the packet of the response, indicating the type of error occurred during transmission.
- $\mathbf{D_0, D_1, ... D_N}$ The actual data that any entity wants to transmit to the other side.
- $\mathbf{CRC_H, CRC_L}$ The high and low byte of the integer value of the CRC algorithm.

CRC stands for *Cyclic Redundancy Check*. It is an error-detecting code that used to check if the transmitted data were corrupted or not. Before any node transmit data, it calculates the CRC of the data and appends the resultant integer (2 bytes) to the end of the data packet. The receiver, after it identifies the end of the packet, it also calculates the CRC of the incoming data and compares the result with the CRC bytes of the received data packet. If there is a match it proceeds otherwise, on its response it will indicate that there was a CRC error with the *ERROR* byte.

To indicate the end of the data packet, two options are given on this project. The first one and the most simple, is that the master, sends always a data packet with fixed number of bytes. As it will be explained in the next section, the receiver (micro-controller) triggers an interrupt when a byte is received. During this interrupt someone can increase a counter to check if this byte is the last expected. The other option is to add two extra bytes at the end of the data packet representing the *end of line*. Then the receive,r with every incoming byte, is checking along with the previous byte, if there is a match.

[1] www.robotis.com/xe/dynamixel_en

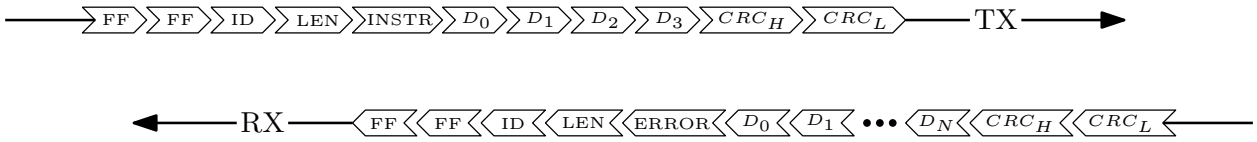


Fig. 5.3: Data packet that Odroid sends and the data packet of the response of the node.

5.4 Motors configuration

As it was already explained, each one of the motors has an embedded micro-controller, the same as the Arduino platform uses, the *ATmega328P-AU*. On this Section, a detailed description of the communication part of the micro-controller's main code will be given.

Enabling/Disabling the UART

The UART port on the micro-controller consists of (among other) the *Receiver* and the *Transmitter*. What is important to notice, is that these modules can be activated and de-activated separately. This property allows the implementation of the RS-485 standard. Looking at the Fig. 5.2, the existence of a diode on the Tx line of a micro-controller and the pull-up resistor, along with the ability to disable the Transmitter allows the in series point-to-point communication.

Each node (motors or sensors) can be in two modes, the receiving mode or better the *RX-mode* and accordingly the transmitting mode or *TX-mode*. A node, cannot be in both modes on the same time. To avoid data collision and to synchronise, in a way, the communication, all nodes are in RX-mode when are idle. Only when a node is addressed by the master it switches mode in order to transmit the response. After it sends the response data packet, it switches back to RX-mode.

To enable or disable the Receiver/Transmitter UART is providing a register that needs to be modified. This register is the **UCSR0B** and it is shown in Fig. 5.4 along with its corresponding bits.

7	6	5	4	3	2	1	0
RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	UCSZ02	RXB80	TXB80

Fig. 5.4: The **UCSR0B** register that controls the enabling/disabling of Rx and Tx modules.

The bits of interest are the *RXEN0*, *TXEN0* and the *RXCIE0*. The first two, if set, they enable the Receiver and the Transmitter of the UART port, respectively. The third one is to enable the trigger of an interrupt, whenever a byte is received from the Receiver. This procedure will be explained in more details, shortly. To easily switch modes along the micro-controller's code, two functions were created where using typical bit math in C-language are configuring the UCSR0B register. These function are shown in Listing 5.1. The function that enables Rx, on the same time disables Tx and vice versa.

Listing 5.1 Function used to switch rx/tx mode

```
void enableRx() {
  // Check the Transmit complete flag
  while (!( UCSR0A & (1<<UDRE0)));
  for(uint8_t i=0; i<25; i++){
    asm("nop");
  }

  // Enable receiver and RX Complete Interrupt
  UCSR0B |= (1 << RXEN0) |(1 << RXCIE0);
  // Disable TX. UART no longer override the TxDn port.
  // Disabling will be immediate as we already checked if there are still ...
    bytes to transmit
  UCSR0B &= ~(1 << TXEN0);
}

void enableTx() {
  // Disable Rx and the RX Complete Interrupt
  UCSR0B &= ~(1 << RXEN0) | (1 << RXCIE0);
  //Enable Tx (overides port condition)
  UCSR0B |= (1 << TXEN0);
}
```

Interrupts Priority

The main code of the micro-controller is using three interrupts. When an interrupt is triggered, the corresponding *Interrupt Service Routine*, ISR, is executed. If there are more than one pending interrupts then the ISR with the highest priority will execute first. During the ISR execution, the interrupts are disabled by default. It is possible though, for an interrupt with higher priority to interrupt an ISR if that is desired. Such phenomenon is called "*nested interrupts*". Nested interrupts must be used with caution and in general should be avoided. Table 5.2 lists, in order of highest priority to the lowest, the interrupt vectors that are used in the main code.

Source	Interrupt Definition
TIMER2 COMPA	Timer/Counter2 Compare Match A
TIMER1 COMPA	Timer/Counter1 Compare Match A
USART, RX	USART Rx Complete

Table 5.2: Interrupt vectors priorities

Timer 2 is used to implement a fixed, accurate *control* loop. Whenever the incremental counter matches a pre-defined value, an interrupt is triggered. The UART *Rx Complete interrupt*, occurs whenever a byte is successfully received in the UART's Rx pin. This interrupt is useful both for storing the incoming bytes to a buffer and to check if the incoming byte is the last one of the receiving data packet, as it was explained in Section 5.3. *Timer1* use will be explained in the *Code Structure* part of this Section.

Code Structure

On this part of the section, the strategy followed to issue the communication between a microcontroller (or else, a motor) and Odroid will be explained.

Any control of a dc-motor consists of at least two tasks, the calculations needed to apply the proper input signal and reading the feedback, to close the control loop. These two tasks take place in a fixed time loop, also called *control loop*. The time of this repeated loop is called *sampling time* as at every loop a sample of the output (sensor read) is taken. This control loop is called for this project the *local control loop* as it is implemented at each of the motor separately.

Odroid on the other hand, it also have a control loop, but this loop controls all the biped platform. What is worth noticing is that the two control loops do not have the same sampling time, with Odroid's sampling time to be much bigger. Therefore, if Odroid communicates once per loop with each one of the motors (for example, to read the position of the motors), from the perspective of each motor, this task happens in an aperiodic manner.

The idea of handling this aperiodic task without "braking" the accurate sampling time of the motor's control loop is shown in Fig. 5.5. Timer 2 with its ISR is setting the sampling time. The reading of the sensor is indicated by 'S' and since the function to read the sensor is very fast (around 60 microseconds) it is moved inside Timer's 2 ISR. The control calculations task is indicated with 'C'.

The key point is to consider the receiving of the instruction *and* the transmitting of the response as one task. This is where the Timer's 1 interrupt comes in. Since the *Rx Complete interrupt* ISR detect the last byte of the incoming data packet, it sets Timer 1 to start count for a (user defined) small amount of time. When the counter match the predefined value, it triggers an interrupt. Inside this interrupt routine, the transmitting of the response takes place.

By treating the communication with Odroid as explained, two "extreme" scenarios must be considered. One is when the Timer 2 interrupt is triggered during the data receiving, and the other when Timer 2 interrupt is triggered during the data transmission.

The ISR of the *Rx Complete interrupt* is executed for as long time as needed to store a byte to a buffer, which means very fast. Therefore, for the first scenario where Timer's 2 interrupt occurs while receiving data, it was decided that there is no need to create a nested interrupt and that the Timer's 2 ISR will be delayed at most, the time of reading a byte and one cycle command. In theory, this time with 500000 bps and 20Mhz clock equals to 1.605 ns. With sampling time at rates of milliseconds, this delay will barely affect the control algorithm.

For the second extreme scenario though, a nested interrupt is used. And that is because Odroid might request for example the full feedback of the motor (position, velocity, current) and the transmission of these data is long enough to delay the new sampling. So in that case, when ever the Timer's 1 ISR start to execute, the global interrupts flag is set again to allow the Timer 2 to interrupt this ISR. The only scenario where this could create a problem, is if new incoming data would come but first, the UART Rx has lower interrupt priority than Timer 1 and also, the *read()* call from Odroid's side, is a *blocking call*, which means that will block until it will receive the response or a timeout will occur. Therefore, the only interrupt that can interrupt Timer's 1 ISR

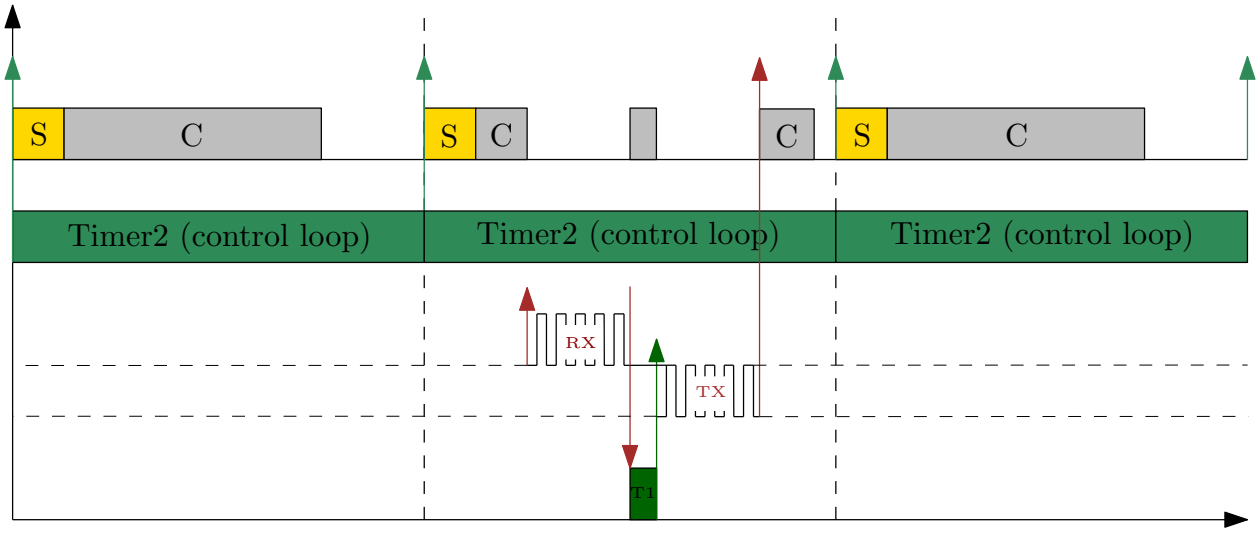


Fig. 5.5: Task schedule of a motor's controller.

is Timer's 2, which as was mentioned, its execution time is quite fast. Fig. 5.6a shows the task schedule of the first scenario while Fig. 5.6b the task schedule with the nested interrupt.

For any of the above task schedules to be feasible, the following relation must be hold:

$$t_S + t_C + t_{rx/tx} \leq T_s \tag{5.1}$$

where t_S is the time of the 'S' task, t_C is the time of the 'C' task, $t_{rx/tx}$ is the time of the worst case scenario[1] of receiving and transmitting data and T_s is the sampling time as this is defined by Timer 2.

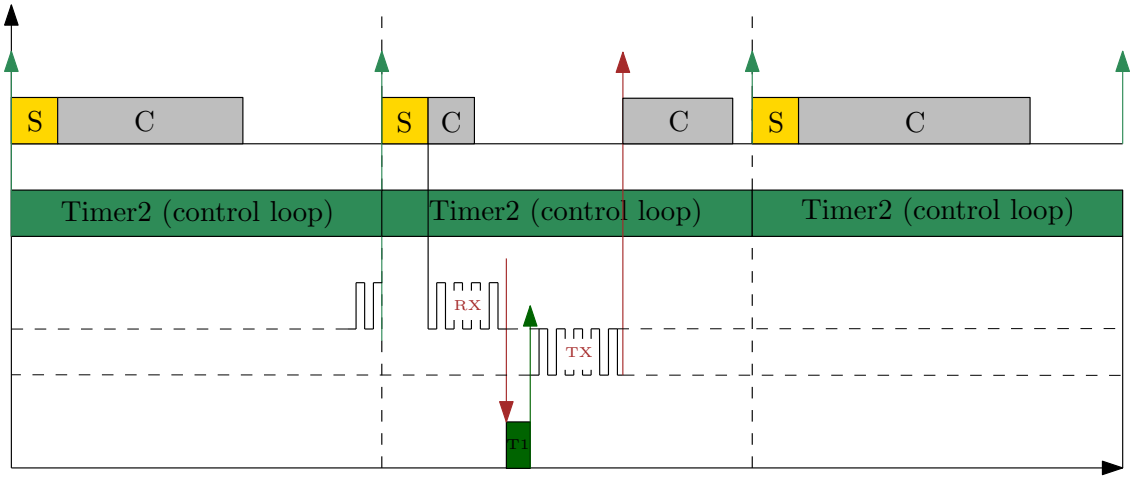
5.5 Odroid configuration

From the side of Odroid, the communication is more simple. Either using the FTDI chip or directly with the exposed Rx/Tx pin on the board, the communication is achieved using the classic Unix C API for Serial Communication.

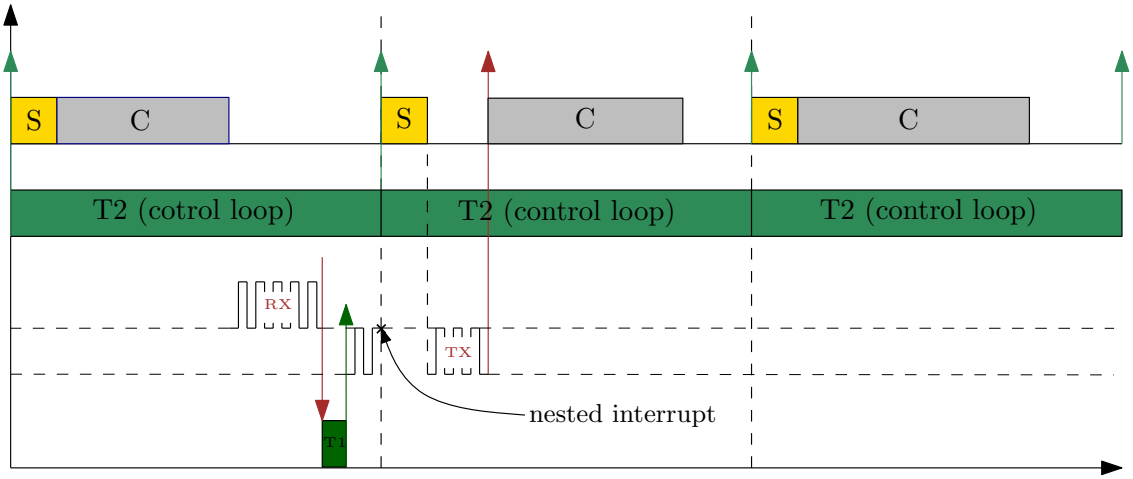
The baud rate is set to 500000 bps as this was determined from the micro-controllers on the other side of the line. The data frame is also set the same as in the micro-controller case, which is 8 data bits, no parity check, 1 stop bit (8N1). The hardware flow control (CTS and RTS pins) is disabled. The input processing is set to *non-canonical* and with raw data (no processing by the API to the incoming data). The output data also are sent in raw form.

As it was also mentioned in Section 5.4 the *read()* function is set to be in blocking mode with timeout set to 10 decisecond (100 milliseconds). There was one issue with the *read()* function. There were times when it was returning 0 bytes. The reason was, probably, that the function was

[1]The maximum number of bytes as a combination of receiving and transmitting



(a) Timer's 2 interrupt occur while receiving data



(b) Timer's 2 interrupt occur while transmitting data and a nested interrupt occurs.

executing before the addressed micro-controller. Was said probably because if that was the case then it should wait 100 milliseconds before it returned which was not the case. This strange behaviour was overpassed by simply calling the function again until the required bytes were received or the timeout time was passed. Fig. 5.7 shows an example of a communication between the Odroid and one of the motors, where the Odroid is handshaking (ping) the motor, as this was shown from a *Logic Analyser*.

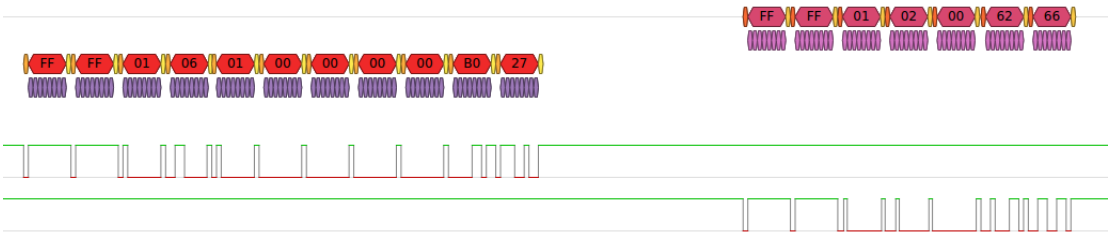


Fig. 5.7: Using a Logic Analyser to scope the data transfer through the network. This is the *PING* of the motor with ID equal to 1.

5.6 ROS

"The Robot Operating System (ROS) is a set of software libraries and tools that help you build robot applications. From drivers to state-of-the-art algorithms, and with powerful developer tools, ROS has what you need for your next robotics project. And it's all open source."[1] The Robot Operating System (ROS) is a set of software libraries and tools that help you build robot applications. From drivers to state-of-the-art algorithms, and with powerful developer tools, ROS has what you need for your next robotics project. And it's all open source.

One of the objectives of this project, is to create the *connection link* between the control of the motors and the reading of the sensors, with ROS. This section will not describe how to use known ROS *packages* related to robotics, as this is not part of the scope of the project, but rather how to extract the informations taken from the robot and "*publish*" then into the ROS world. It is up to the user how these information will be used.

serialProxy library

For the communication between Odroid and the nodes connected to the RS-485 network a C++ library was created, named *serialProxy*. This library consists of one class with various functions that allow the user to interact with the motors. Some examples of these functions are shown in Table 5.3.

• *ping(id)*	Looks if a motor with $ID = id$ is connected to the network.
• *read_position(id)*	Reads the position of the motor with the specified ID.
• *read_feedback3(id)*	Reads the full feedback of the motor (position, velocity, current) of the motor with the specified ID.
• *goal_position(id, pos)*	Sets the goal position to be equal to val, of the motor with the specified ID.
• *set_gain(id, 'P', val)*	Sets the *P-Gain* of the motor with the specified ID (assuming that the motor has a code with an implemented PID controller).
• *enable_torque(id)*	Makes the motor with the specified ID to be stiff (sets the current position as the goal position).

Table 5.3: Some of the implemented function of the *serialProxy* library.

There are more implemented functions in the library but since the user is able to upload his own code to each of the motors, this functions might need to be modified or new to be created. For example, if the control of a motor is designed as a *torque controller*, the *set_gain(id, 'P', val)* function on Table 5.3 is of no use any more.

What is worth noticing though, is the form of the *constructor* of the class. Recall from 4.2.4 that Odroid-C1 has two of its serial ports (of 4 in total) exposed on its pins on the board. The first is mapped on */dev/ttyS2* and the second on */dev/ttyS0*. With the use of the constructor, two instances of the class can be created to associate each of the to one of the ports. That way, the nodes connected to the RS-485 network can be split to two, releasing the load of the Tx/Rx lines.

[1] www.ros.org

serialProxy and ROS

With the library implemented, the connection with ROS is quite straight forward. Assuming that a package is created (e.g. *motor_driver*, the header file (.h) is placed in the *include* folder of the package while the source (.cpp) file is placed in the *src* folder. With the appropriate changed in the *CMakeList* file the library is compiled with the *catkin_make* command and can be *included* in any -ROS- *node*.

"The use of nodes in ROS provides several benefits to the overall system. There is additional fault tolerance as crashes are isolated to individual nodes. Code complexity is reduced in comparison to monolithic systems. Implementation details are also well hidden as the nodes expose a minimal API to the rest of the graph and alternate implementations, even in other programming languages, can easily be substituted."[1].

To take advantage of the ROS nodes and the two serial ports of the Odroid board, the motors of one leg were connected to one of the port and the motors of the other leg to the second port. Two nodes are used, one for each leg, and each one is responsible to communicate with the motors (and the sensors) using an instance of the serialProxy library. Both of them are *publishing* information through the *topics* that were created. That way, a connection with the biped platform and the "world" of ROS was achieved. The user now, can connect his ROS packages with the biped platform by *subscribing* to these topics.

[1] wiki.ros.org/Nodes

Chapter 6

DC Motor Control

On this chapter the modelling of a dc servo motor will be discussed. Afterwords, a parameter estimation will be performed of a linearised system. Finally, a simple position-velocity controller will be presented, as well as, a state observer. The scope of this section, it is not to offer a state of the art controller of a dc-servo motor but rather to demonstrate the advantage of the ability to upload new codes to the motors of a robotic platform and to offer a baseline to the design of more complicated controller designs.

6.1 DC-Motor Mathematical Model

The schematic diagram of an electric dc-motor is shown in Fig. 6.1. A voltage source v is applied across the coil of the armature. The coil can be described by an inductance L_a in series with a resistance R_a in series with the *bemf*[1] which opposes the voltage source. The BEMF is generated when the armature is rotating and is proportional to the angular velocity.

$$e(t) = K_b \dot{\theta}_m(t) \tag{6.1}$$

[1]Back electromotive force

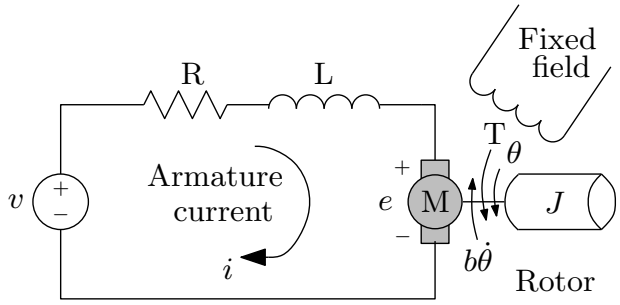


Fig. 6.1: DC-Motor electric diagram.

where K_b is the back EMF constant. The differential equation for the armature circuit is

$$L_a \frac{di(t)}{dt} + R_a i(t) + e(t) = v(t) \tag{6.2}$$

The armature current produces the torque that is applied to the inertia and friction, therefore

$$J\ddot{\theta}_m(t) + B\dot{\theta}_m(t) = \tau_m = K_i i_a \tag{6.3}$$

An interesting property about the torque and back-emf constants. Although functionally K_i and K_b are two separate parameters, for a given motor, their values are,

$$K_b(V/rad/sec) = K_i(N \cdot m/A) = K_m \tag{6.4}$$

Taking the Laplace transform of Equations 6.1,6.2, 6.3 (assuming initial conditions are zero)

$$K_b(s)\Theta(s) = E(s) \tag{6.5}$$

$$(L_a s + R_a)I_a(s) + E(s) = V(s) \tag{6.6}$$

$$(Js^2 + Bs)\Theta_m(s) = T_m(s) = K_i I_a(s) \tag{6.7}$$

Considering $V(s)$ as the input and $Theta(s)$ as the output, the *transfer function* of the system is,

$$\frac{\Theta(s)}{V(s)} = \frac{K_i}{s\left[L_a J s^2 + (L_a B + R_a J)\, s + R_a B + K_i K_b\right]} \tag{6.8}$$

Considering as state variables the armature current i_a and the angular velocity of the shaft $\dot{theta}_m$ the state representation is,

$$\frac{d}{dt}\begin{bmatrix} \dot{\theta} \\ i \end{bmatrix} = \begin{bmatrix} -\frac{B}{J} & \frac{K_m}{J} \\ -\frac{K_m}{L_a} & -\frac{R_a}{L_a} \end{bmatrix} \begin{bmatrix} \dot{\theta} \\ i \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{L} \end{bmatrix} V \tag{6.9}$$

$$y = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} \dot{\theta} \\ i \end{bmatrix} \tag{6.10}$$

while the state space representation of the system with the angular displacement of the shaft also as a state (and the only measurement) is,

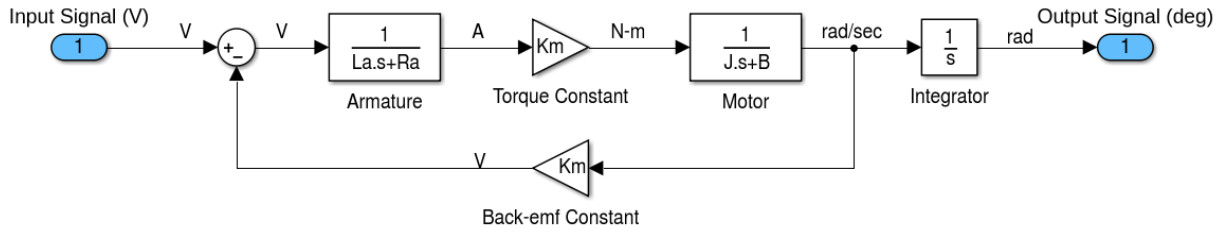


Fig. 6.2: Simulnk Model of a DC-servo motor.

$$\frac{d}{dt}\begin{bmatrix}\theta\\ \dot{\theta}\\ i\end{bmatrix} = \begin{bmatrix}0 & 1 & 0\\ 0 & -\frac{B}{J} & \frac{K_m}{J}\\ 0 & -\frac{K_m}{L_a} & -\frac{R_a}{L_a}\end{bmatrix}\begin{bmatrix}\theta\\ \dot{\theta}\\ i\end{bmatrix} + \begin{bmatrix}0\\ 0\\ \frac{1}{L}\end{bmatrix} V \tag{6.11}$$

$$y = \begin{bmatrix}1 & 0 & 0\end{bmatrix}\begin{bmatrix}\theta\\ \dot{\theta}\\ i\end{bmatrix} \tag{6.12}$$

In Fig. 6.2 the *Simulink* model of the DC-Motor is shown.

6.2 Collecting Data

The data collection was achieved using the new implemented board of the motor. Since this board has embedded the driver of the motor, the micro-controller and the position sensor, all it had to be done, is to transfer the data through the serial port. In all the experiments that are described on this Section, a single motor was used unmounted from the biped platform, without any load attached to it. With that configuration, the motor can send the sensor data to the Arduino IDE's *serial monitor*, as long as one of the acceptable baud rate is selected (e.g. 115200). From the serial terminal it is easy to copy the printed values to a *text* file. This file then, can be imported to *Matlab* for plotting or further process of the data.

6.3 Measuring the position

The process to measure the position is straight forward since there is the available sensor. The reading is achieved through the *SPI* port. The sensor is sending the data through a *SSI* protocol but there is no compatibility problem with the *SPI*. On Fig. 6.3 the *SSI interface* of the *AS5145* is shown.

This sensor is a *systen-on-a-chip*. It runs an algorithm that linearises its output. The *Status Bits* of the SSI interface contain flags that are informing if this linearisation was failed and therefore

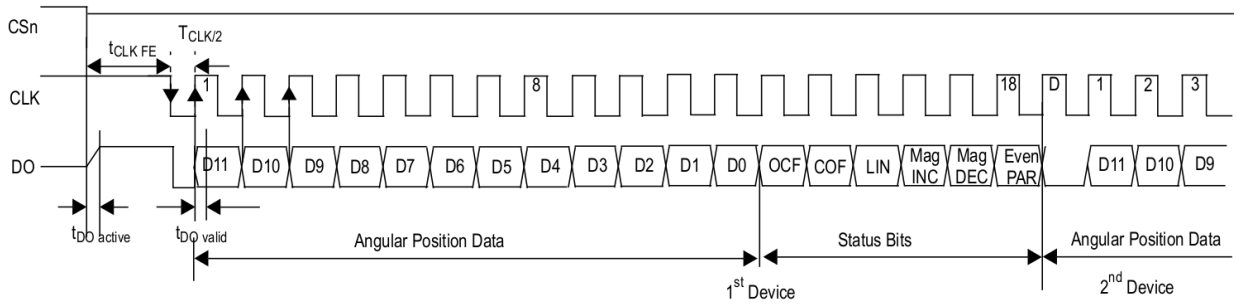


Fig. 6.3: SSI interface of AS5145.

OCF	COF	LIN	MagINC	MagDEC	Parity
1	0	1	0	0	Even check sum of bits 1:15
			0	1	
			1	0	
			1	1	

Table 6.1: Data validity flags.

the reading is not valid. The user should repeat the reading in such a case. Data D11:D0 are valid, when the status bits have one of the configurations of Table 6.1.

Listing 6.1 is showing the configuration of the SPI port as well as the function used to read the position. In Fig. 6.4 the output of the sensor is plotted, while the motor was rotating with constant speed where it can be seen that the output is linear.

Listing 6.1 The setup of the SPI port and the function to read the sensor.

```
/* ------------------  SPI configuration  ------------------- */
  DDRB = 0;  //Check if needed
  // configure SCK(PB5) and Slave Select(PB2) as output, MISO(PB4) as input
  DDRB = (1 << PB5) | (1 << PB2) | (0 << PB4);
  // configure SPI as master, SPR0=1 -> fosc/16 CHANGE: SPR0 = 0 -> fosc/4
  // 16 Mhz XTAL:
//  SPCR = (1 << SPE) | (1 << MSTR) | (0 << CPOL) | (1 << SPR0) | (CPHA << 1);
  // 20 Mhz XTAL:
  SPCR = (1 << SPE) | (1 << MSTR) | (0 << CPOL) | (1 << SPR1) | (CPHA << 1);
/* ---------------------------------------------------------- */

void position_update(int &position) {
  uint8_t u8data;  uint32_t u32result;
  // Pulse to initiate new transfer
  digitalWrite(10, HIGH);
  digitalWrite(10, LOW);
  //Receive the 3 bytes (AS5145 sends 18bit word)
```

```
    for (uint8_t byteCount=0; byteCount<3; byteCount++) {
       u32result <<= 8;  // left shift the result so far - first time shifts ...
      0's-no change
       SPDR = 0xFF;  // send 0xFF as dummy (triggers the transfer)
       while ( (SPSR & (1 << SPIF)) == 0);  // wait until transfer complete
       u8data = SPDR;  // read data from SPI register
       u32result |= u8data;  //store the byte
    }

    u32result >>= 12; // * no check of the flags!
    int *ssi_pnt16 = (int *)&u32result;
    position = *ssi_pnt16;

}
```

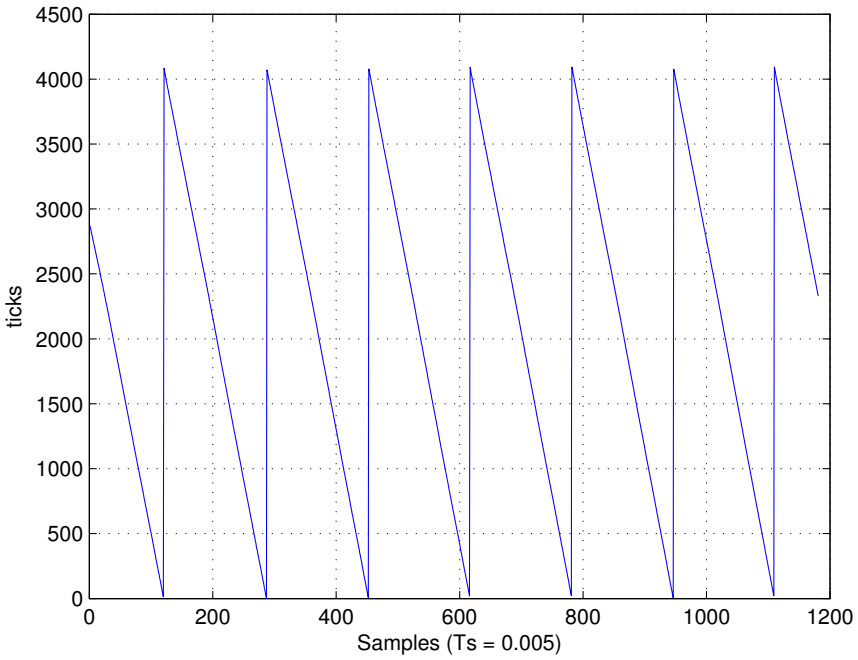


Fig. 6.4: The absolute position output of the AS5145 sensor.

Position data trough the RS-485 network

Since the resolution of the sensor is 12-bits then the data of the position can fit in a 16-bit integer (*uint_16t* in C++). Therefore the position is *travelling* through the network in units of "ticks" with its value to range from $[0 - 4095]$ in *dec* representation or from $[0 - 0x0FFF]$ in the *hex* representation. It is up to the program on Odroid side to interpret this value to rad/sec or rpm.

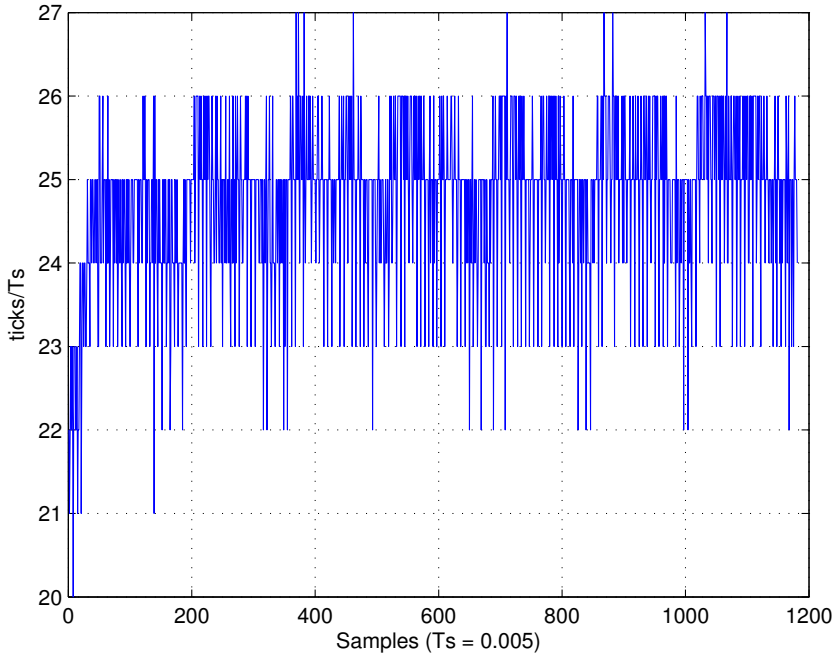


Fig. 6.5: The noisy outcome of the derivative of the position.

6.4 Calculating the velocity

The output of the system is the position of the shaft (Equation 6.12), therefore, the derivative of the position must be calculated to acquire the velocity. A simple numerical differentiation was performed based on the *backward differencing* formula,

$$vel[k] \approx \frac{pos[k] - pos[k-1]}{T_s} \tag{6.13}$$

where $v[k]$ is the approximation of the velocity at the current instance k, $pos[k]$ is the position that was just read and $pos[k-1]$ is the position that was read on the previous period, T_s. Fig. 6.5 shows the derivative of the position signal that was plotted in Fig. 6.4 where the duty cycle of the PWM was constant.

Before start analysing Fig. 6.5 it must be clarified the resolution of the speed calculations. According to the -few- informations given with *Hitec* motor, the *maximum speed* of the motor is $0.12\frac{sec}{60°}$. To translate this to rad/sec,

$$0.12\frac{sec}{60°} = 0.72\frac{sec}{360°} = 1.389\frac{rev}{sec} = 8.7273\frac{rad}{sec} \;\; or \;\; 83.333rpm \tag{6.14}$$

and to $ticks/T_s$ (with $T_s = 0.005$),

$$1.389\frac{rev}{sec} = 5689.34\frac{ticks}{sec} = 28.45\frac{ticls}{5ms} \tag{6.15}$$

From Eq. 6.14 and Eq. 6.15 it can be observed that the resolution of the speed expressed in rad/sec is,

$$\frac{8.7273}{28.45} = 0.3067\frac{rad}{tick} \tag{6.16}$$

Looking back to Fig. 6.5 it can be seen that the speed is fluctuating roughly between 23 and 26 $ticks/Ts$ and this is translated to $\approx \pm 0.9 rad/sec$ which is far from satisfactory. For that purpose, a first order Low Pass Filter (LPF) was used in order to smooth this noisy signal. The transfer function of a fisrt order LPF is,

$$H(s) = \frac{\omega_c}{s + \omega_c} = \frac{1}{1 + \frac{s}{\omega_c}} \tag{6.17}$$

where $\omega_c = 2\pi f$ with f the cut-off frequency.

Using the Fast Fourier Transform analysis of the signal and with some experimentation, the cut-off frequency was chosen to be $f = 2.556$. To implement the LPF, first the transfer function was discretized and from it the difference equation was taken:

$$H(s) = \frac{16.0598}{s + 16.0598} \leftrightarrow H(z) = \frac{v_f[k]}{v[k]} = \frac{0.1484}{z - 0.8516} \tag{6.18}$$

where $v_f[k]$ is the output of the filter (filtered velocity) and $v[k]$ the input of the filter (the noisy derivative of the position). Then the difference equation is:

$$v_f[k] = 0.1484\, v[k-1] + 0.8516\, v_f[k-1] \tag{6.19}$$

Fig. 6.6 shows the "raw" velocity versus the filtered velocity.

It is more obvious now, that there is a periodic pattern. In fact this periodic change of the speed can be heard while the motor is turning. This suggest that there is a part in the gears that creates more friction. The motor case was opened to check the gear and it was observed that the last gear, the one that produced the output torque, is not rotating co-axial with the vertical axis of the shaft. Originally it was thought that after cutting the mechanical brake, it left behind a non flat surface but after a close inspection it was realised that this is not the case. It was decided that no more action should be taken about this issue and to continue with the analysis of the motor.

Two more observation can be made from Fig. 6.6. The first one has to do with the response of the filter which seems that follows the velocity of the motor quite fast. The second has to do with the variation of the filtered velocity which is a little bit more than 1 tick difference per T_s (roughly

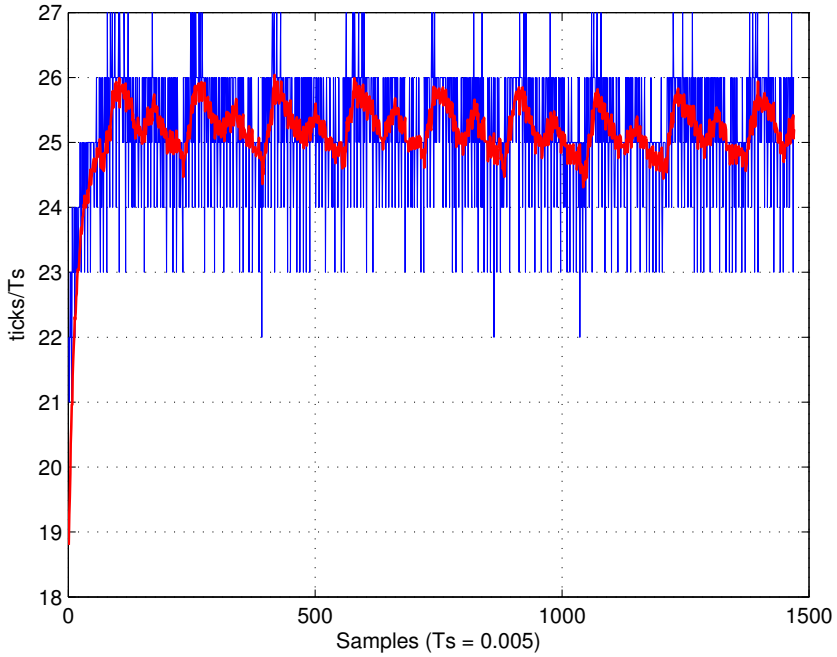


Fig. 6.6: The result of the implementation of the LPF filter.

speaking). In order to smooth the output more, another filter was applied, this time a medium (or average filter), which practically is another LPF. The implementation of the filter is as simple as summing the current filtered velocity with the three previous filtered velocities and divide the sum by four. The result of applying also the second filter is shown in Fig.6.7.

Finally, the last observation has to do with the fact that the last output of the velocity is (in that case) between the $25 ticks/Ts$ and $26 ticks/Ts$. If an error on the velocity 'calculation' of $\pm 0.3 rad/sec$ is acceptable, someone could keep only the integer part of the output and that has a result like the one on Fig.6.8.

Velocity data trough the RS-485 network

As it is already mentioned, the maximum speed in rad/sec is 8.7273. That gives a resolution of $0.3067 rad/sec$. If it is assumed that only the integer part of the filtered velocity is kept, then the range of the velocity is $8.7273/0.3067 \approx 28$ or else, $[0 - 28]$. This information can be encoded in one byte, therefore only one byte is used to transfer the data of the velocity.

6.5 Parameter Estimation

Before continue with the design of a controller, a parameter estimation process will be discussed. Having a relatively accurate model can be proved useful during the process of tuning the gains of

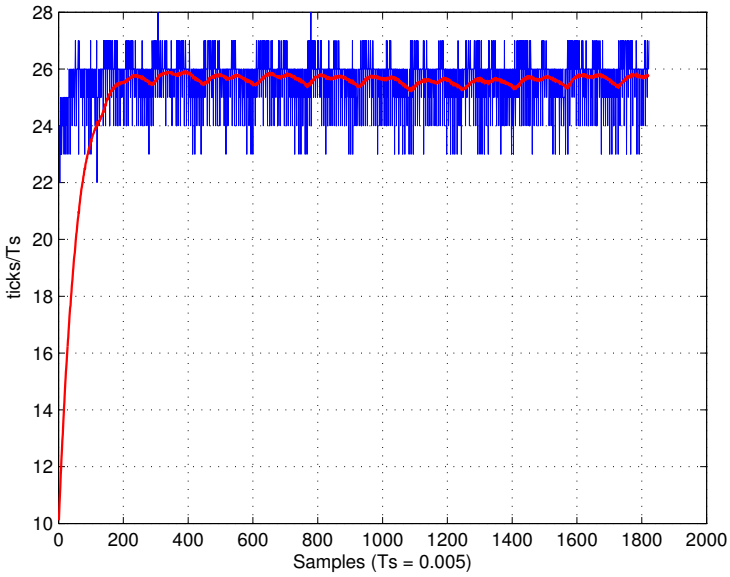


Fig. 6.7: The velocity of the motor filtered twice.

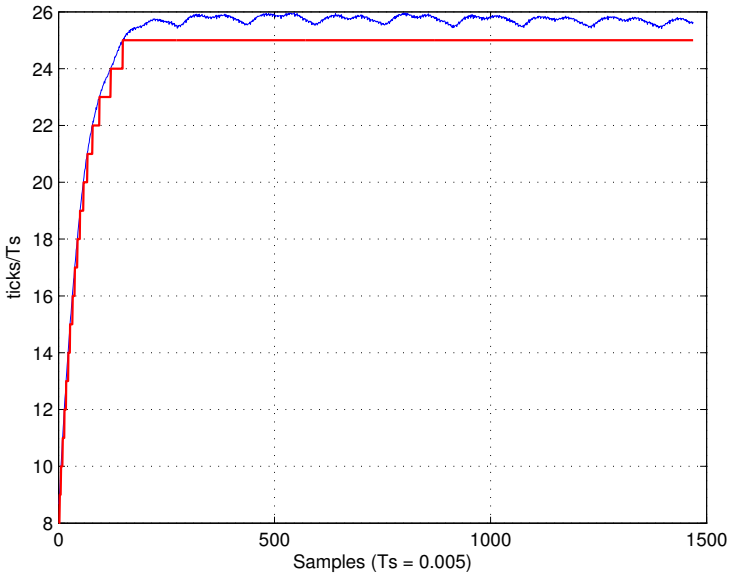


Fig. 6.8: The final outcome of filtering the derivative of the position.

the controller.

The non linearity that is counteracted is between the relationship of the input PWM signal and the output velocity. In a linear system, these two values would increase proportionally. As it is shown in Fig. 6.9 that is not the case. On that figure, the PWM duty cycle was increasing every 3 second in order to give time to the motor to reach it 'steady state' speed.

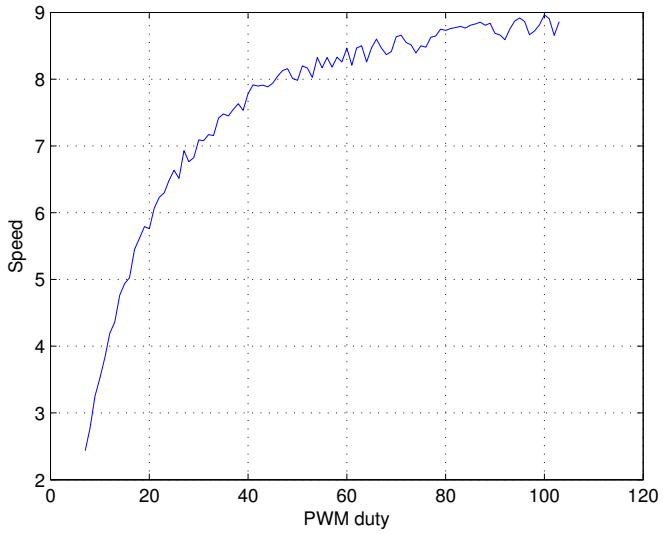


Fig. 6.9: The non linear relationship between PWM duty cycle and output speed.

To counter act this non linearity, a function f is fitted on the plot of Fig. 6.9. The inversion of this function, f^{-1}, is used then as part of the -unknown- system. Fig. 6.10 shows the fitted function f and its inverse.

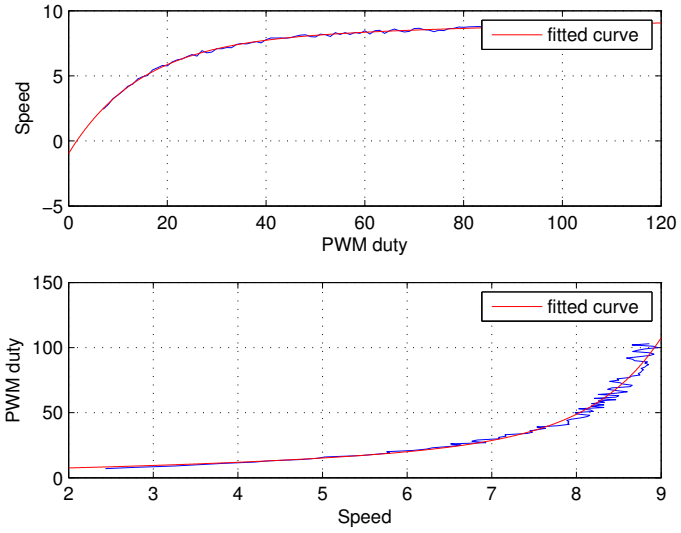


Fig. 6.10: The fitted function and its inverse.

The new system now, has as input the desired speed and as output the actual speed. Repeating

the same experiment, but now with input the desired speed, the affect of the f^{-1} is shown in Fig. 6.11.

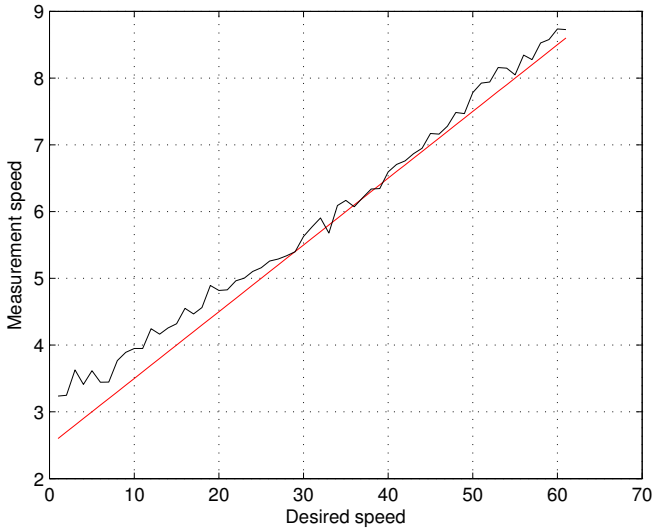


Fig. 6.11: The linearising effect of the f^{-1}

Now that the model is 'linear' the parameter estimation can be performed. The parameter estimation was implemented using the *fminsearch* command of MATLAB. *fminsearh* finds the minimum of a problem specified by

$$\min_x = f(x)$$

where $f(x)$ is a function that returns a scalar.

Basically the idea is to use the model of Equation 6.9 and its unknown parameters. Input-output data were gathered, where input is the a reference of desired speed and the output is the actual speed of the motor. Then with random initial values, the model is simulated. The *fminsearch* function will try to minimise at every step, the *norm* distance between the actual output and the simulated one. At the end of every step, it updates the values of the parameters. In Fig. 6.12 it shows the result of this process.

The estimated parameters are,

J_m	0.0024
B	0.0092
K_m	0.0212
L_a	0.0242

Table 6.2: The estimated parameters

The R_a parameter, the resistance of the motor it was simple to be measured and it kept fixed to 2.2 Ohms. The reason that the values of Table 6.2 do not have any units, is because it does

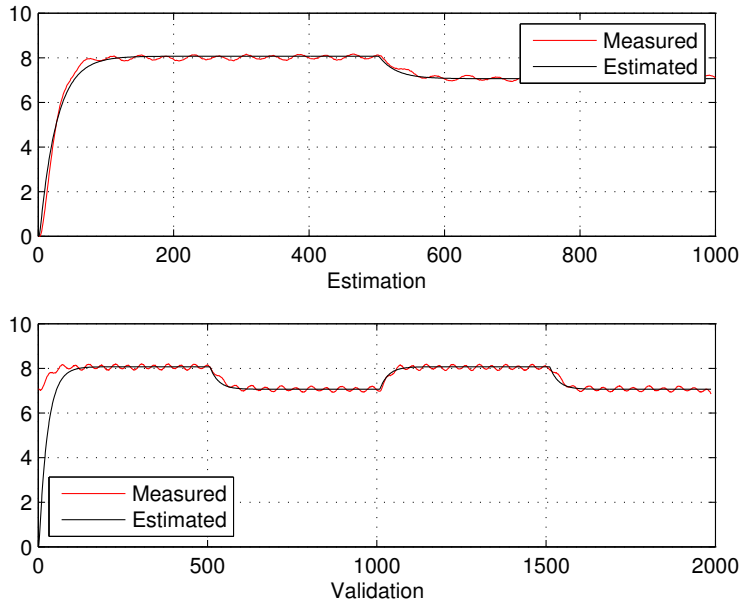


Fig. 6.12: The simulation of the model with the estimated parameters follows the actual output of the system.

not mean that these values represent the real quantities. For the optimisation process there are just parameters of a function without any physical meaning. As long as the dynamics of the model with these parameters follow the dynamics of the input-output data though, this model can be used during the tuning of the gains of a possible PID controller.

6.6 PID Controller

PID controllers have survived many changes in technology, from mechanics and pneumatics to microprocessors via electronic tubes, transistors, integrated circuits. The microprocessors has had a dramatic influence on the PID controller. Practically all PID controllers made today are based on microprocessors. This has given opportunities to provide additional features like automatic tuning, gain scheduling, and continuous adaptation.

6.6.1 The Algorithm

The basic algorithm will be summarised here as it is described in [11] and [4].

The "textbook" version of the PID algorithm is described by:

$$u(t) = K\left(e(t) + \frac{1}{T_i}\int_0^t e(\tau)d\tau + T_d\frac{de(t)}{dt}\right) \tag{6.20}$$

where y is the measured process variable, r the reference variable, u is the control signal and e is the control error ($e = y_{sp} - y$). The control signal is thus a sum of three terms: the P-term (which is proportional to the error), the I-term (which is proportional to the integral of the error), and the D-term (which is proportional to the derivative of the error). The controller parameters are the proportional gain K, integral time T_i, and derivative time T_d.

The transfer function of a PID controller in the s-domain is expressed as:

$$\frac{U(s)}{X(s)} = G_c(s) = K_P + \frac{K_I}{s} + K_D s \tag{6.21}$$

A digital implementation of this controller can be determined by using a ***discrete approximation*** for the *derivative* and *integration*.

Numerical Differentiation

There are commonly three simple discretization methods of the derivative, the *Forward* difference, the *Backward* difference and the *Trapezoidal* methods. The backward difference method is,

$$u(k) = \frac{1}{T_s}\left(x[k] - x[k-1]\right) \tag{6.22}$$

The z-transform of Equation 6.22 is

$$U(z) = \frac{1 - z^{-1}}{T_s} X(z) \cdot \frac{z}{z} = \frac{z-1}{T_s z} X(z) \tag{6.23}$$

Similarly, the z-transform for the forward and trapezoidal method, respectfully,

$$U(z) = \frac{z-1}{T_s} X(z) \tag{6.24}$$

$$U(z) = \frac{2}{T_s}\frac{z-1}{z+1} X(z) \tag{6.25}$$

Numerical Integration

The same methods are applied to the numerical integration as well.

The integration of $x(t)$ can be represented by the **forward-rectangular integration**

$$u[k] = u[k-1] + T_s x[k] \tag{6.26}$$

The z-transform of Equation 6.26 is

$$U(z) = z^{-1}U(z) + T_s X(z) \Leftrightarrow \frac{U(z)}{X(z)} = \frac{T_s z}{z-1} \tag{6.27}$$

As expected Equation 6.27 is the inverse of the Equation 6.24. Similarly, for the backwards and trapezoidal method respectively,

$$\frac{U(z)}{X(z)} = \frac{T_s}{z-1} \tag{6.28}$$

$$\frac{U(z)}{X(z)} = \frac{T_s}{2}\frac{z+1}{z-1} \tag{6.29}$$

PID Algorithm Implementation

Putting all together, the z-domain transfer function of the **PID controller** is

$$G_c(z) = K_P + \frac{K_I T z}{z-1} + K_D \frac{z-1}{Tz} \tag{6.30}$$

The complete difference equation algorithm that provides the PID controller is obtained by adding the three terms to obtain

$$u(k) = \boldsymbol{K_P} x(k) + \boldsymbol{K_I}\left[u(k-1) + Tx(k)\right] + \left(\frac{\boldsymbol{K_D}}{T}\left[x(k) - x(k-1)\right]\right) \tag{6.31}$$

Equation 6.31 can be easily implemented in a microcontroller using the tools provided in this report. Of course, someone can obtain a *PI* or *PD* controller by setting the appropriate gain equal to zero.

Chapter 7

Kinematics

On this Chapter, a theoretical *Forward Kinematics* analysis is given. Although the forward kinematics are not part of the scope of the project, it might be useful starting point for someone, during the future improvements of the biped platform. Due to the limited time, this analysis was not tested in practice, even though, it would be an interesting demonstration of the ROS package and the platform in general.

7.1 Forward Kinematics

There are two option to be considered as the origin of the kinematic chain. The first one, would be to start the analysis from the *Center of Mass* of the biped. The other option, to start from the *Center of Pressure*. Assuming the *Center of Pressure* is known (calculated or measured) the latter option was chosen. In addition, only the frames of one leg are attached as the two legs are symmetrical.

The frames are attached to the joints in the sense of the *Denavit-Hartenberg* (DH) parameters as it is shown in Fig. 7.1.

The frame '0', is the one attached to the *Center of Pressure* and is orientation is according to the *anatomical position* (Chapter 2). An intermediate frame 'd' was attached between the frame '0' and frame '1' (d for "dummy") on projection of the frame '1' on the floor. The reason for that is because the z-axis of frames '0' and '1' intersect which is not suggested by the DH method. In Fig. 7.1 is shown an example where frame '0' and 'd' coincide.

Starting the analysis, we remind the definition of the DH parameters [7]:

- a_{i-1} The distance from $\hat{z}_{i-1}$ to $\hat{z}_i$ measured along $\hat{x}_{i-1}$.
- α_{i-1} The angle from $\hat{z}_{i-1}$ to $\hat{z}_i$ measured along $\hat{x}_{i-1}$.
- d_i The distance from $\hat{x}_{i-1}$ to $\hat{x}_i$ measured along $\hat{z}_i$.
- θ_i The angle from $\hat{x}_{i-1}$ to $\hat{x}_i$ measured along $\hat{z}_i$.

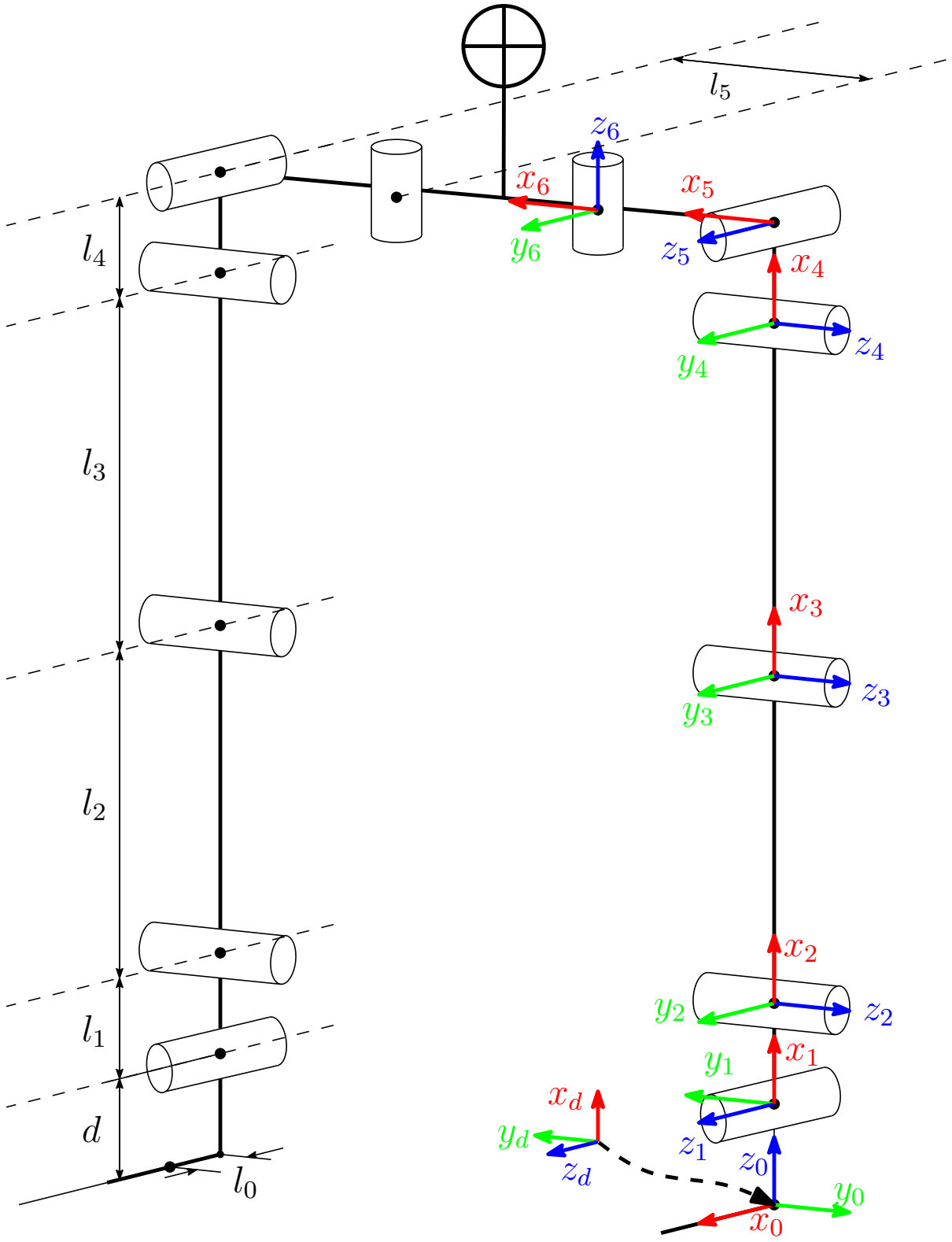


Fig. 7.1: The structure of the joints of Poppy-UPC and the attached frames.

The transformation matrix from frame 'a' to frame 'b' is denoted as ${}^a_bT$ and the relation ${}^ap = {}^a_bR^2p$ holds. Since both the frame '0' and the frame 'd' are fixed, then $x_0 = z_d$, $y_0 = -y_d$ and $z_0 = x_d$ therefore,

$$ {}^0_dT = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{7.1} $$

Of course the someone could arrive to the same result if he would rotate the frame '0' first around x_0 for $-90\,\text{deg}$ ($Rot_Z(-\pi/2)$) then around y for $-90\,\text{deg}$ ($Rot_Y(-\pi/2)$) and finally around x for $-90\,\text{deg}$ ($Rot_X(-\pi/2)$).

$$ {}^0_dR = Rot_Z\left(-\frac{\pi}{2}\right)\ Rot_Y\left(-\frac{\pi}{2}\right)\ Rot_X\left(-\frac{\pi}{2}\right) \tag{7.2} $$

Frame '1' is also fixed from the frame 'd' aspect. Therefore is only translated in the z_0 direction to distance d_0.

$$ {}^d_1T = \begin{bmatrix} 0 & 0 & 1 & 0^1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & d \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{7.3} $$

The rest of the frames, follow the DH-parameters convention. For example the transformation matrix between frame '1' and frame '2',

- The distance from $\hat{z}_1$ to $\hat{z}_2$ measured along $\hat{x}_1$ is equal to d_1.
- The angle from $\hat{z}_1$ to $\hat{z}_2$ measured along $\hat{x}_1$ is equal to $\pi/2$.
- d_2 The distance from $\hat{x}_1$ to $\hat{x}_2$ measured along $\hat{z}_2$ is equal to 0.
- The angle from $\hat{x}_1$ to $\hat{x}_2$ measured along $\hat{z}_2$ is equal to θ_2.

[1] In case frame '0' does not coincide with frame 'd' then this entry would equal to $-l_0$.

Following the same procedure up to the frame '6' the DH-parameters *table* is derived:

	a_{i-1}	α_{i-1}	d_i	θ_i
$1 \rightarrow 2$	l_1	$\pi/2$	0	θ_2
$2 \rightarrow 3$	l_2	0	0	θ_3
$3 \rightarrow 4$	l_3	0	0	θ_4
$4 \rightarrow 5$	l_4	$-\pi/2$	0	$90 + \theta_5$
$5 \rightarrow 6$	l_5	$\pi/2$	0	θ_6

Table 7.1: The DH-parameters of one of the legs.

Table 7.1 it is to be verified, by simply setting all the θ angles equal to zero and checking the *rotation matrix* of each frame.

To acquire the transformation matrices someone has to follow the general form of ${}^{i-1}_{i}T$:

$$ {}^{i-1}_{i}T = \begin{bmatrix} c\theta_i & -s\theta_i & 0 & a_{i-1} \\ s\theta_i\, c\alpha_{i-1} & c\theta_i\, c\alpha_{i-1} & -s\alpha_{i-1} & -s\alpha_{i-1}\, d_i \\ s\theta_i\, s\alpha_{i-1} & c\theta_i\, s\alpha_{i-1} & c\alpha_{i-1} & c\alpha_{i-1}\, d_i \\ 0 & 0 & & 1 \end{bmatrix} \tag{7.4} $$

where $c\alpha_{i-1}$ and $s\alpha_{i-1}$ are equal to $cos(\alpha_{i-1})$ and $sin(\alpha_{i-1})$ respectively. Similarly, $c\theta_i$ and $s\theta_i$ stand for $cos(\theta_i)$ and $sin(\theta_i)$.

Chapter 8

Conclusions and further work

This chapter shows the conclusion of the development of this project. First an economical analysis is given in order to support the conclusion for the main goal of the project. Finally, some suggestions are made for possible lines of future work.

8.1 Economical Analysis

This section is divided to an analysis only for one motor in order to compare it with the Dynamixel one and to an analysis of the total cost of the project.

Cost of the motor modification

The cost of the modification of the motor consists of the Hitec motor that was bought, the cost of printing the PCBs, the components to populate the PCBs including the connectors of the cables. The summary of all these components is shown on Table 8.1

	Component	Price
Motor	HS-7954SH	€100
PCB	ITEAD studio	€24
Driver	VNH5180-A	€2.710
Sensor	AS5145	€9.26
Arduino	ATmega328P-AU	€2.89
Components	(R,C,etc)	€5
Connectors	Molex	€10
SUM	———	**129.86**

Table 8.1: List of components for the Poppy-UPC motors

The cost of the resistors, capacitors etc is a rough estimation. Probably the cost is much less. The cost of the PCBs is for 10 pieces and two designs.

Estimation of the total cost of Poppy-UPC

To the calculation of the total cost of the biped platform, the 3D printing parts are not included since it is difficult to calculate the cost of the material used. Nevertheless, the printed parts of the original Poppy can not be printed from the low cost 3D printers, unlike the designed parts of Joan. Therefore not including the cost of the 3D printing is in favour of the original Poppy. Table 8.2 sums up the total cost of the biped.

	Units	Price
Motors	12	€1200
PCB	2	€48
Driver	12	€32.55
AS5145	12	€111.20
Arduino	12	€34.68
Components	12	€60
Connectors	6	€60
FSR	6	€41.4
IMU	1	€83.5
Odroid	1	€35
SUM	———	**€1707**

Table 8.2: The list of components of the biped platform.

8.2 Conclusions

The main reason to implement this project, was the desire to study bipedal walking, but the high cost of acquiring a biped platform was a huge obstacle. That is why the main objective of this project, was to modify an open-source humanoid robot in order to reduce the cost but on the same time, to maintain its quality.

From the cost reduction point of view it is fair to state that the goal was achieved. Either looking at the Table 8.1 or at the Table 8.2 the conclusion is the same. On the first case, the cost to buy a cheaper motor and modify it, costs around 129.86 euros. The Dynamixel motor that was replaced, costs 270 euros. It is a cost reduction of 50%. Looking at the total cost, and lets assume that original Poppy legs are using only the MX-28AT model of Dynamixel (in reality, the hip motors are the even more expensive model, MX-64AT) the cost only for the motors would be $270 \times 12 = 3240$ euros. On the other hand, the Poppy-UPC biped platform, along with the motors, the sensors, and the Odroid, costs in total 1707 euros.

As far as the quality is concerned, component wise, both motors are more or less equal. Both of them use the same type of sensor with the same resolution. The driver of the Poppy-UPC might

have more protection but apart of that does not support something more than the Dynamixel. And also both of the motors using the same communication protocol.

The Dynamixel motor has the advantage of the more compact outer case without the need of printing new components to facilitate the mounting on rigid bodies. It also uses only 3 cables. On the other hand, Poppy-UPC is using 6 cables in total. The big advantage though over the Dynamixel motor, is the ability to re-program the embedded micro-controller which is "*tweeked*" to run 25% faster than the typical Arduino. One final aspect is the total weight. Hitec motor by itself might be lighter than the Dynamixel but it needs extra 3D-printed parts in order to be mounted to the main body. The 3D-printed parts might be light enough but the weight of the screws used to hold them together is what gives the disadvantage over the Dynamixel motor.

After this analysis, the main goal of the project to reduce the cost of the original Poppy while maintaining the same quality, is achieved.

Further more, pressure sensors were attached to each of the foot to indicate the state of the biped (single support or double support), as well as the always necessary IMU unit to 'sense' the orientation of the biped. The Odroid unit is responsible to communicate with the motors and the sensors and to transfer this data to the ROS world using the implemented package. Using this information, packages to calculate the CoM and the CoP are easily to implement. The basis for both of these calculation is the Forward Kinematics. A theoretical approach was proposed even though it was not part of scope of the project.

8.3 Future Implementation

Future steps for the Poppy-UPC can be endless. To keep this section short, the most important of them (according to the author) are listed:

- Calculate the CoM and CoP.
- Parse the model of the biped to the Gazebo simulator.
- Use a *Reat-Time* kernel on Odroid to implement a Real time communication.
- Implement the semi-passive knee as the original Poppy design.
- Use brush-less DC motors instead of dc-servo.
- Improve the controller (disturbance rejection observer, velocity profiles, torque control etc.)
- Attempt a walking algorithm using the Linear Inverted Pendulum and the Capture Point concept.

References

[1] M. Abdallah and A. Goswami. A biomechanically motivated two-phase strategy for biped upright balance control. *IEEE International Conference on Robotics and Automation (ICRA)*, 2005.

[2] Aldebaran. NAO robot. `https://www.aldebaran.com/en/cool-robots/nao`, 2016.

[3] AMS. AS5145H Rotary Sensor. `http://ams.com/eng/Products/Position-Sensors/Magnetic-Rotary-Position-Sensors/AS5145H`, 2008.

[4] Karl Johan Aström and Richard M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeston Univeristy Press, 2010.

[5] Atmel. Atmel microcontroller. `http://www.atmel.com/Images/doc8161.pdf`, 2009.

[6] FTDI chip. FTDI chip. `http://www.ftdichip.com/`, 2016.

[7] John J. Craig. *Itroduction to Robotics Mechanics and Control*. Prentice Hall, 2005.

[8] M.H.P. Dekker. Zero-moment point method for stable biped walking. Internship report, July 2009.

[9] docs.python. Multiprocessing. `https://docs.python.org/2/library/multiprocessing.html`.

[10] docs.python. Threading vs Multiprocessing. `https://docs.python.org/2/library/threading.html`.

[11] Richard C. Dorf and Robert H. Bishop. *Modern Control Systems*. Prentice Hall, 12 edition, 2011.

[12] Boston Dynamics. Atlas - The Agile Anthropomorphic Robot. `http://www.bostondynamics.com/robot_Atlas.html`.

[13] Pratt J. E. and Tedrake R. Velocity-based stability margins for fast bipedal walking. *Springer Berlin Heidelberg*, pages 299–324, 2006.

[14] Thomas Fischl. USBasp - USB programmer for Atmel AVR controllers. `http://www.fischl.de/usbasp/`, 2016.

[15] Honda. ASIMO. The world's most advanced Humanoid Robot. `http://asimo.honda.com/`, 2016.

[16] Q. Huang, K. Yokoi, S. Kajita, K. Kaneko, H. Arai, N. Koyachi, and K. Tanie. Planning walking patterns for a biped robot. *IEEE Trans. on Robotics and Automation*, pages 280–289, 2001.

[17] Flowers INRIA. Poppy Humanoid Robot. `https://www.poppy-project.org/creatures/poppy-humanoid/`, 2016.

[18] S. Kajita, F. Kanehiro, K. Kaneko, K. Yokoi, and H. Hirukawa. The 3D linear inverted pendulum mode: a simple modeling for a biped walking pattern generation. In *Intelligent Robots and Systems, 2001. Proceedings. 2001 IEEE/RSJ International Conference on*, volume 1, 2001.

[19] Hassan K. Khalil. *Nonlinear systems*. Prentice Hall, 1996.

[20] M. Lapeyre, P. Rouanet, and P.Y. Oudeyer. The poppy humanoid robot: Leg design for biped locomotion. *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2013.

[21] MOOG. MOOG agrrement with IIT. `http://www.moog.com/news/operating-group-news/2016/moog-announces-agreement-with-the-italian-institute-of-technology-iit-for-joint-devel` 2016.

[22] M. Popovic, A. Englehart, and H. Herr. Angular momentum primitives for for human walking: Biomechanics and control. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2004.

[23] G. Pratt and M. Williamson. Series elastic actuators. In *IEEE International Conference on Intelligent Robots and Systems*, pages 399–406, 1995.

[24] J. Pratt, J. Carff, S. Drakunov, and A. Goswami. Capture point: A step toward humanoid push recovery. In *2006 6th IEEE-RAS International Conference on Humanoid Robots*, pages 200–207, Dec 2006.

[25] Jerry Pratta and Ben Krupp. Design of a bipedal walking robot. Technical report, Yobotics, Inc, 2004.

[26] Wiki Python. PyQt. `https://wiki.python.org/moin/PyQt`.

[27] Qt. Qt. `http://www.qt.io/developers/`.

[28] PAL Robotics. REEM-C: Robotics Research. `http://pal-robotics.com/en/products/reem-c/`, 2016.

[29] sourceforge. PySerial. `http://pyserial.sourceforge.net/`.

[30] IEEE SPectrum. The Next Generation of Boston Dynamics' ATLAS Robot Is Quiet, Robust, and Tether Free. `http://spectrum.ieee.org/automaton/robotics/humanoids/next-generation-of-boston-dynamics-atlas-robot`, 2016.

[31] ST. VNH5180A-E:Automotive fully integrated H-bridge motor driver. `http://www.st.com/web/en/resource/technical/document/datasheet/CD00264619.pdf`, 2008.

[32] T. Takenaka, T. Matsumoto, and T. Yoshiike. Real time motion generation and control for biped robot -1st report: Walking gait pattern generation-. *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2009.

[33] T. Takenaka, T. Matsumoto, and T. Yoshiike. Real time motion generation and control for biped robot -3rd report: Dynamics error compensation-. *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2009.

[34] T. Takenaka, T. Matsumoto, T. Yoshiike, T. Hasegawa, S. Shirokura, H. Kaneko, and A. Orita. Real time motion generation and control for biped robot -4 h report: Integrated balance control-. *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2009.

[35] T. Takenaka, T. Matsumoto, T. Yoshiike, and S. Shirokura. Real time motion generation and control for biped robot -2nd report: Running gait pattern generation-. *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2009.

[36] M. Vukobratovic and B. Borovac. Zero-moment point - thirty five years of its life. *International Journal of Humanoid Robotics.*, 1(1):157–173, 2004.

[37] Yobotics. Spring Flamingo Robot. `http://www.ai.mit.edu/projects/leglab/robots/Spring_Flamingo/Spring_Flamingo.html`, 1996-2000.

Automatic Control and Robotics

Implementation of a robot platform to study bipedal walking

ANNEX A: Previous work
ANNEX B: Codes examples

Autor: Dimitris Zervas
Director/s: Dr. Manel Velasco and Dr. Cecilio Angulo

Convocatòria: April 2016

Escola Tècnica Superior
d'Enginyeria Industrial de Barcelona

Appendices

Appendix A

Previous work

At this part of the appendix, the previous work related to this project is going to be presented.

A.1 Driver

H-Bridge motor driver

To drive the motor, the ***VNH5180A-E*** fully integrated H-bridge motor driver from STMicroelectronics, was selected [31]. The two main reasons behind this choice, were the output current of 8 A, that covers the need of most of the hobby RC-servos in the market and, the integrated current sensor.

General Description

The *VNH5180A-E* is a full bridge motor driver intended for a wide range of automotive applications. The device incorporates a dual monolithic high-side driver and two low-side switches. Both switches are designed using STMicroelectronics' well known and proven proprietary $VIPower$ M0 technology that allows to efficiently integrate on the same die a true Power MOSFET with an intelligent signal/protection circuitry. The three dies are assembled in PowerSSO-36 TP package on electrically isolated leadframes. This package, specifically designed for the harsh automotive environment offers improved thermal performance thanks to exposed die pads. Moreover, its fully symmetrical mechanical design allows superior manufacturability at board level.

The input signals IN_A and IN_B can directly interface to the microcontroller to select the motor direction and the brake condition. The DIAG_A/EN_A or DIAG_B/EN_B , when connected to an external pull-up resistor, enables one leg of the bridge. Each DIAG_A/EN_A provides a feedback digital diagnostic signal as well. The normal operating condition is explained in the truth Table A.2. The CS pin allows to monitor the motor current by delivering a current proportional to its value when CS_DIS pin is driven low or left open. When CS_DIS is driven high, CS pin is in high

impedance condition. The PWM, up to 20 KHz, allows to control the speed of the motor in all possible conditions. In all cases, a low level state on the PWM pin turns off both the LS_A and LS_B switches.

Key Features

- Output current: 8 A
- 3 V CMOS compatible inputs
- Undervoltage shutdown
- Overvoltage clamp
- Thermal shutdown
- Cross-conduction protection
- Current and power limitation
- Very low standby power consumption
- PWM operation up to 20 KHz
- Protection against loss of ground and loss of V_{CC}
- Current sense output proportional to motor current
- Output protected against short to ground and short to V_{CC}

Interfacing with Arduino

They key points in order to connect with Arduino are:

- The pins *SOURCE_HSA/B* (high side mosfet's source) and *DRAIN_LSA/B* (low side mosfet's drain), must be connected to each other. Of course the junction between them is where the motor cables are also connected.
- In normal operating conditions the *EN/DIAG_A* and *EN/DIAG_B* pins are considered as inputs by the device. They must be connected to an external pull up resistor.

Apart of these points, the connection with the Arduino is straightforward.

Table A.1: Pin connection between Driver and Arduino

Driver		Arduino
IN_A	↔	D4
IN_B	↔	D7
IN_PWM	↔	D3

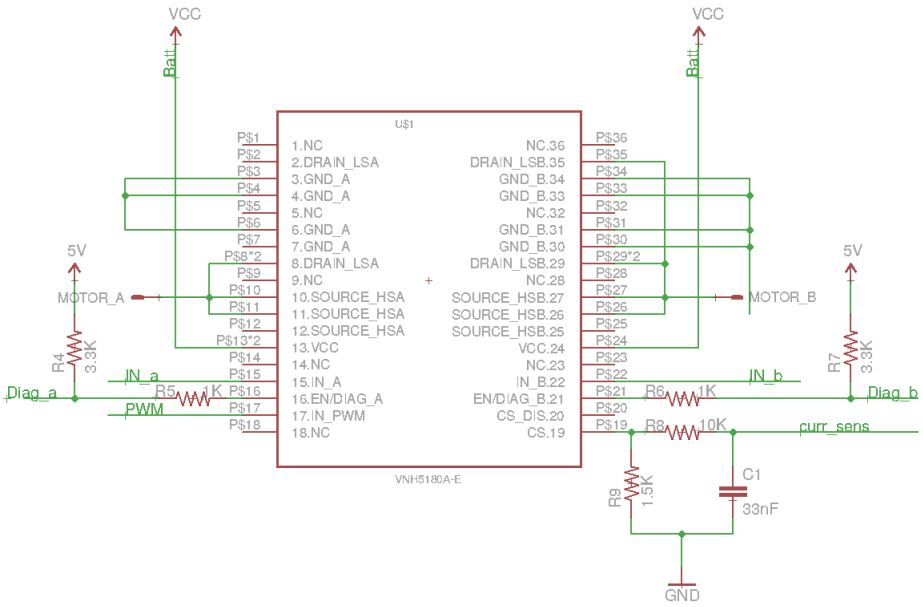


Fig. A.1: Schematic of the VNH5180A-E

where D4 and D7 are the digital pins we chose to control the direction of the motor. The possible combination of all the pins in *normal operation* conditions are shown in Table A.2

Table A.2: Truth table in normal operating conditions

IN_A	IN_B	$DIAG_A/EN_A$	$DIAG_B/EN_B$	OUT_A	OUT_B	Operating mode
1	1	1	1	H	H	Brake to V_{CC}
	0				L	Clockwise (CW)
0	1			L	H	Counterclockwise (CCW)
	0				L	Brake to GND

Run the motor

In order to run the motor all it needs to be done, is choose the direction (CW/CCW) and apply the PWM in the proper pin. An example code is show in Listing A.1.

Listing A.1 Arduino code to run the motor

```
// Make sure you don't hae cross - conduction (even though
// chip has protection about it)
digitalWrite(7,LOW);  digitalWrite(4,LOW);  //Brake
digitalWrite(4,HIGH);
OCR2B = 100;
```

where *OCR2B*, is the 8-bit register of the timer is used to create the PWM signal at pin D3, as explained in Section A.4.2

A.2 AS5145

On this chapter it will be discussed the implementation of the electronics of the servo motor that will allow the user to drive the motor and read the position of the rotor using a microcontroller. In first section the sensor for the reading of the position will be presented while in second section it will be presented the driver that was chosen to run the motor. Finally in section three it is presented the microcontroller that is used (Arduino) and the way to use it with the driver and the sensor.

Magnetic Encoder

In order to read the position of the motor, the rotary magnetic encoder AS5145 from *Austria Microsystems (AMS)* [3] was chosen. The AS5145H is a contactless magnetic rotary position sensor for accurate angular measurement over a full turn of 360° and over an extended ambient temperature range of -40°C to 150°C. The *absolute* angle measurement provides instant indication of the magnet's angular position with a resolution of 0.0879° = 4096 positions per revolution via a serial bit stream and as a PWM signal.

General Description

The *AS5145* is a contact-less magnetic encoder for accurate angular measurement over a full turn of 360 degrees. It is a system-on-chip, combining integrated Hall elements, analog front end and digital signal processing in a single device.

To measure the angle, only a simple two-pole magnet, rotating over the center of the chip, is required. The magnet can be placed above or below the IC. The *absolute angle measurement* provides instant indication of the magnet's angular position with resolution of 0.0879 deg = 4096 positions per revolution. This digital data is available as a serial bit stream and as a PWM signal.

Key Features

- Contact-less high resolution rotational position encoding over a full turn of 360 degrees.
- Two digital 12-bit absolute outputs:
  - Serial interface
  - Pulse width modulated (PWM) output
- Three incremental outputs
- User programmable zero position

- Failure detection mode for magnet placement, monitoring, and loss of power supply
- Red-Yellow-Green indicators display placement of magnet in Z-axis
- Serial read-out of multiple interconnected AS5145 devices using Daisy-Chain mode
- Tolerant to magnet misalignment and gap variations

Interfacing with Arduino

In order to read the position of the rotor, the **S**ynchronous **S**erial **I**nterface (SSI) was used. The schematic of the IC in Figure A.2, shows which pins need to be used.

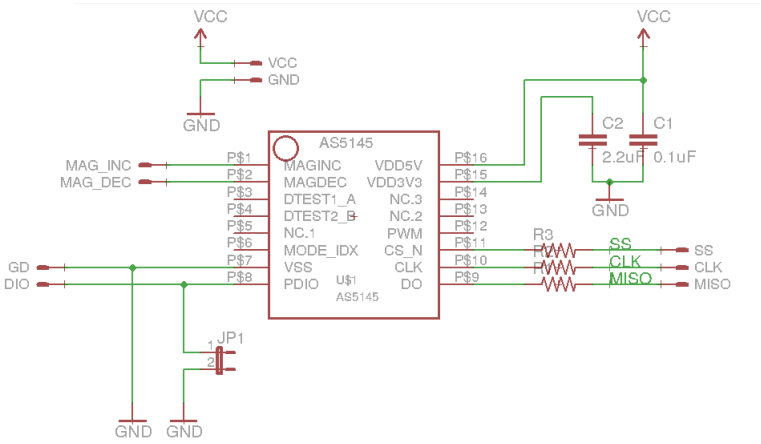


Fig. A.2: Schematic of AS5145

Where the pins of interest are,

VDD5V	5V power supply pin.
CSn	Chip Select. Active low.
CLK	Clock input of SSI.
DO	Data Output of SSI.
VSS	GND

And the connection between AS5145 pins and Arduino pins are,

AS5145		Arduino
CSn	↔	D10
DO	↔	D12 (MISO)
CLK	↔	D13

where, for example, D12 stands for Digital Pin 12.

Synchronous Serial Interface (SSI)

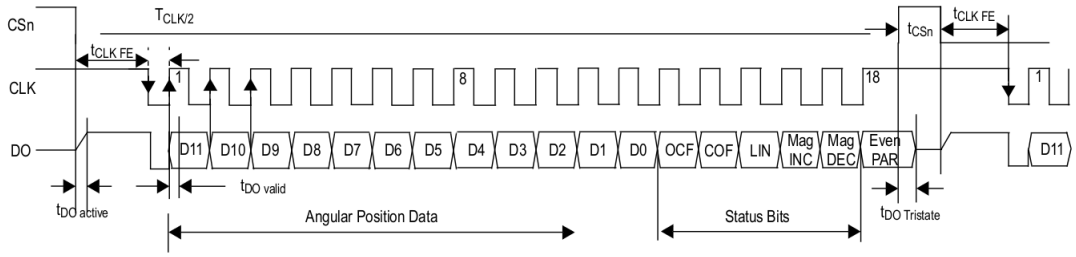


Fig. A.3: SSI Interface

If CS_n changes to logic low, Data Out (DO) will change from high impedance (tri-state) to logic high and the read-out will be initiated.

- After a minimum time t_{CLKFE}, data is latched into the output shift register with the first falling edge of CLK.
- Each subsequent rising CLK edge shifts out one bit of data.
- The serial word contains 18 bits, the first 12 bits are the angular information $D[11:0]$, the subsequent 6 bits contain system information, about the validity of data.
- A subsequent measurement is initiated by "high" pulse at CSn with a minimum duration of t_{CSn}.

Data Content

- **D11:D0** absolute position data (MSB is clocked out first
- **OCF** (Offset Compensation Finished), logic high indicates the finished Offset Compensation Algorithm.
- **COF** (Cordic Overflow), when the bit is set, the data D11:D0 is invalid. The absolute output maintains the last valid angular value. This alarm can be resolved by bringing the magnet within the X-Y-Z tolerance limits.
- **LIN** (Linearity Alarm), logic high indicates that the input field enerates a critical output linearity. When the bit is set, the data D11:D0 can still be used, but can contain invalid data. This alarm can be resolved by bringing the magnet within the X-Y-Z tolerance limits.
- **EVEN PARITY** bit for transmission error detection of bits 1...17 (D11...D0, OCF, COF, LIN, MagINC, MagDEC).

Data D11:D0 is valid, when the status bits have the following configurations

OCF	COF	LIN	MagINC	MagDEC	Parity
1	0	1	0	0	Even check sum of bits 1:15
			0	1	
			1	0	
			1	1	

Read the position

This is the function to read the sensor.

Listing A.2 Arduino function to read the position of the rotor

```
uint32_t readSSI () {
    uint32_t data;
    //Pulse to initiate new transfer
    digitalWrite(10,HIGH);
    digitalWrite(10,LOW);
    //Receive the 3 bytes (AS5145 sends 18bit word)
    for (u8byteCount=0; u8byteCount<3; u8byteCount++){
       u32result <<= 8;  // left shift the result so far - first time shifts ...
    0's-no change
       SPDR = 0xFF;  // send 0xFF as dummy (triggers the transfer)
       while ( (SPSR & (1 << SPIF)) == 0);  // wait until transfer complete
       u8data = SPDR;  // read data from SPI register
       u32result |= u8data;  //store the byte
    }
    // Print only the data no check of flags!
    u32result >>= 12;
    data = u32result;
    u32result = 0;
    return data;
}
```

And an example of calling it,

Listing A.3 readSSI() call example

```
uint_32t pos = readSSI();
```

Selecting Proper Magnet

Typically the magnet is $6mm$ in diameter and $2.5mm$ in height. Magnetic materials such as rare earth AINiCo/SmCo5 or NdFeB are recommended. The magnetic field strength perpendicular to the die surface has to be in the range of $\pm 45mT ... \pm 75mT$ (peak). The magnet's field is verified using a gauss-meter. The magnetic field Bv at the given distance, along a concentric circle with radius of $1.1mm$ (R1) is in the range of $\pm 45mT ... \pm 75mT$ (see Figure A.4)

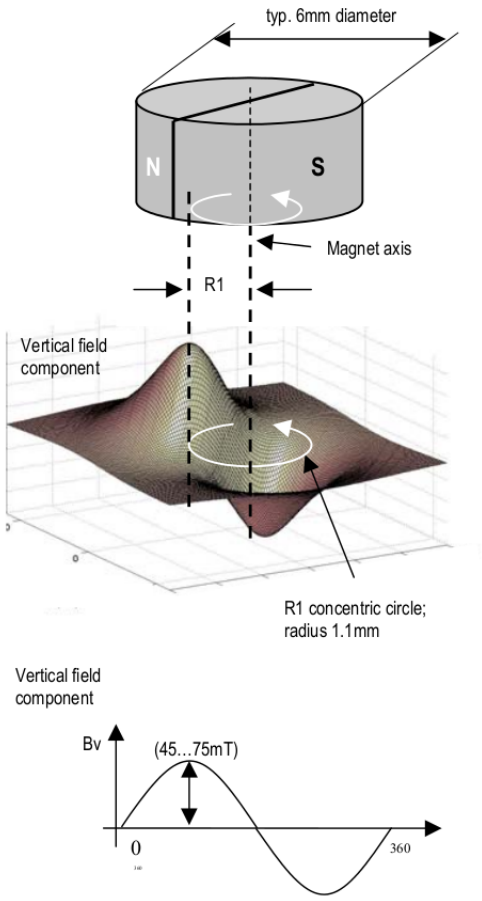


Fig. A.4: Typical magnet (6x3) and Magnetic Field Distribution

Physical Placement of the Magnet

The best linearity can be achieved by placing the center of the magnet exactly over the defined center of the chip as shown in the drawing below:

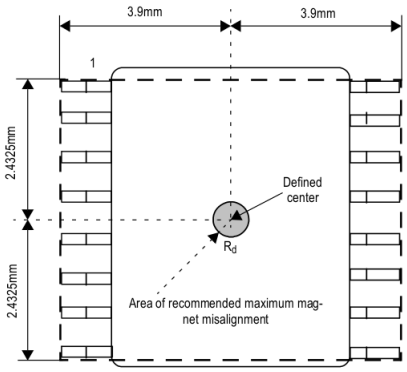


Fig. A.5: Defined Chip Center and Magnet Displacement Radius

The magnet's center axis must be aligned within a displacement radius R_d of $0.25mm$ from the defined center of IC. The magnet can be placed below or above the device. The distance can be chosen such that the magnetic field of the die surface is within specified limits. The typical distance "z" between the magnet and the package surface is $0.5mm$ to $1.5mm$, provided the use of the recommended magnet material and dimensions ($6mm \times 3mm$).

Alignment Mode

The alignment mode simplifies centering the magnet over the center of the chip to gain maximum accuracy.

Alignment mode can be enabled with the falling edge of *CSn* while $PDIO = logic\ high$. Afterwards, there are two ways to check if the magnet is proper placed.

- In alignment mode, the Data bits D11:D0 of the SSI change to a 12-bit displacement amplitude output. A high value indicates large X or Y displacement, but also higher absolute magnetic field strength. The magnet is properly aligned, when the difference between highest and lowest value over one full turn is at a minimum. Under normal conditions, a properly aligned magnet will result in a reading of less than 128 over a full turn. Stronger magnets or short gaps between magnet and IC will show values larger than 128. These magnets are still properly aligned as long as the difference between highest and lowest value over one full turn is at a minimum.
- Under normal conditions, the *MagINCn* and *MagDECn* indicators will be equal to 1 when the alignment mode reading is less than 128. At the same time, both hardware pins MagINCn

(pin 1) and MagDECn (pin 2) will be pulled to VSS. A properly aligned magnet will therefore produce a $MagINC = MagDEC = 1$ signal throughout a full 360 deg turn of the magnet.

The Alignment mode can be reset to normal operation by a power-on-reset (disconnect/reconnect power supply) or by a falling edge on CSn with PDIO = low.

A.3 Arduino

A.4 ATmega328 microcontroller - Arduino

For this application we chose the *Atmel* microcontroller *ATmega328P-AU*, which is also used to *Arduino UNO* boards [5]. That way the user can take advantage of all the functionalities of Arduino, such as the IDE and the libraries. The only difference with an Arduino board is that the code is uploaded through ISP instead of USB.

A.4.1 Peripheral Features

Some of the peripheral features of the micro-controller are,

- Two 8-bit Timer/Counters with Separate Prescaler and Compare Mode
- One 16-bit Timer/Counters with Separate Prescaler, Compare Mode and Capture Mode
- Real Time Counter with Separate Oscillator
- Six PWM Channels
- 8-channel 10-bit ADC in TQPF and QFN/MLF package Temperature Measurement
- 6-channel 10-bit ADC in PDIP package Temperature Measurement
- Programmable Serial USART
- Master/Slave SPI Serial Interface
- Byte-oriented 2-wire Serial Interface (Philips I^2C compatible)

A.4.2 Setup

The system clock is at 16Mhz. The connection with the sensor and the driver is already described in subsections A.2 and A.1 respectively.

Arduino IDE, is using a main loop which is repeating every time as soon as the code inside it is executed. For any typical digital control system, there is the need of a fixed sampling time. In

order to achieve that, *Timer0* was used, to trigger an interrupt in the desired sampling time. This time will be referred as ***control loop*** from now on.

Note! The use of *Timer0* interrupt interferes with the arduino library that uses the ***delay()*** function. If the interrupt routine is used the user shouldn't use the *delay()* function any more. Even though the compiler will not find any error, the accuracy of the timing of the *delay()* command is lost.

Timer 0 for Control Loop

Timer0 is used in the *"Clear Timer on Compare Match"* or **CTC** mode. The timer has an 8-bit register called *OCR0A*. It also has a counter, *TCNT0* that, if the timer is active, it increases its value every timer-clock cycle. Whenever $OCR0A = TCNT0$ the counter goes to 0 again (on the same clock) and an interrupt is triggered.

The timer-clock can be configured from the *TCCR0B* register. The configuration of these registers give the user the choice to choose the control loop frequency.

Listing A.4, shows the configuration of *Timer0* for this application,

Listing A.4 Setup of Timer0 registers

```
TCCR0A = 0;
TCCR0B = 0;

TCCR0A |= B01000010;
TCCR0B |= B00000101;
// to be able to use the interrupt
TIMSK0 |= B00000010;
loop_flag = 1;

OCR0A = 78;  //with CS00:2 = 101 -> period = 0.01
// Enable global interrupts
sei();
```

The interrupt routine must be as fast as possible. All it does is to raise a boolean flag. In the control loop, after the execution of the code, this flag is turned back to LOW.

Listing A.5 Timer0 interrupt routine.

```
/* Counter0 compare match interrupt - for control loop*/
ISR(TIMER0_COMPA_vect) {
  if (loop_flag == 0){
    loop_flag = 1;
  }
}
```

And an example of a control loop,

Listing A.6 Example of control loop

```
void loop() {
  // loop_flag was set 1 in setup
  if (loop_flag == 1) {

    /* Your code here */

    loop_flag = 0;
  }

}
```

After the execution of "*Your code*", the microcontroller will stay in the *loop()* doing nothing, until the *loop_flag* will be raised again from the interrupt routine, which happens in a fixed -sampling-time. For that reason "*Your code*" must be executed before the end of the *control loop*. If the user wants to check if the code exceeds this time, he can use an *else* statement in the interrupt routine.

Timer 2 for PWM generation

Apart of the use of *Timer0* for the control loop, *Timer2* is also used to generate the PWM signal that will be used to drive the motor.

Timer2 is used in the *"Fast PWM"* mode. This mode provides a high-frequency PWM waveform and the reason for that is its single-slope operation. The counter counts from BOTTOM to TOP and then restarts from BOTTOM. BOTOM is equal to 0 while TOP can be configured from the timer registers. This high frequency makes the fast PWM mode well suited for power regulation, rectification, and DAC applications. High frequency allows physically small sized external components (coils, capacitors), and therefore reduces total system cost.

Every time the timer overflows it toggles the state of OCOB which is the *Digital Pin 3*. Listing A.7, shows the setup of the timer,

Listing A.7 Timer2 setup for Fast PWM

```
/* ---------- Timer 2 - Configuration (FAST_PWM) ------------ */
  TCCR2A = 0;
  TCCR2B = 0;
  TCCR2A |= B00100011;
  TCCR2B |= B00000111;
/* ---------------------------------------------------------- */
```

The value of the 8-bit register ***OC2B*** corresponds to the PWM duty cycle. So if a PWM signal, with 50% duty cycle is required, someone could use the command of Listing A.8.

Listing A.8 Setup of PWM duty-cycle

```
OCR2B = 128;
```

By setting the *OCR2B* register the PWM signal generation starts.

A.5 System Identification and Parameter Estimation

A.6 System Identification

System Identification Toolbox provides MATLAB functions, Simulink blocks, and an app for constructing mathematical models of dynamic systems from measured input-output data. It lets you create and use models of dynamic systems not easily modeled from first principles or specifications.

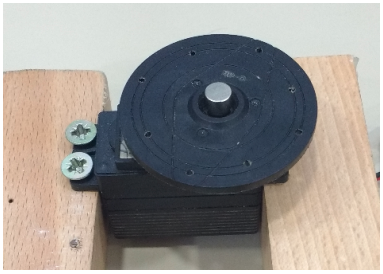
The toolbox provides identification techniques such as maximum likelihood, prediction-error minimization (PEM), and subspace system identification. To represent nonlinear system dynamics, you can estimate Hammerstein-Weiner models and nonlinear ARX models with wavelet network, tree-partition, and sigmoid network nonlinearities. The toolbox performs grey-box system identification for estimating parameters of a user-defined model. You can use the identified model for system response prediction and plant modeling in Simulink. The toolbox also supports time-series data modeling and time-series forecasting.

A.6.1 Data acquisition

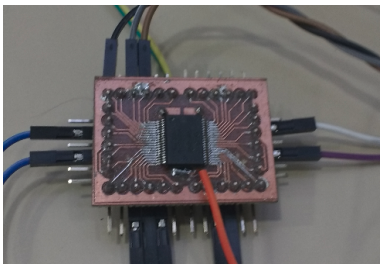
The most important thing in any system identification is the data someone has in order to use them in the estimation and validation process. In order to collect this data, various experiments were conducted. The input of the system (Voltage) as well as the output (Position) were measured. To help the process of collecting data, codes providing communication between MATLAB and Arduino are provided.

The setup is simple. Arduino and MATLAB are communicating through the *Serial* interface. The sensor used to measure the *position* is the *AS5145*. Its functionality is described with details in Section A.2. In short, it is a magnetic encoder that senses the rotation of a bipolar magnet on top of it. The magnet is attached on the shaft on the outer part of the servo motor as shown in Fig. A.6a. The sensor is positioned in a close distance to the magnet in a kind of random alignment as shown in Fig. A.6d. That way it is able to test also the misalignment limits of the sensor (Section A.2). The motor is driven using the *VNH5180A-E* full bridge, Fig. A.6b. The use of the driver is described in Section A.1. "*Dummy*" boards for both the driver and the sensor were designed in order to connect them easily with the Arduino board. In Fig. A.6 the whole setup is presented. The input of the system, the applied voltage, is calculated from the applied *PWM* signal.

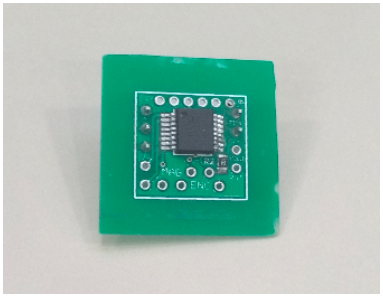
The Arduino code as well as the MATLAB code are provided with this report. The "concept" of the implementation is briefly explained.



(a) Driver test board



(b) Driver test board, populated



(c) Sensor test board

(d) Sensor test board

Fig. A.6: The setup of the experiment

The "Arduino side", is receiving the desired input (*PWM duty cycle* and *direction*), it applies it to the system, measures the current position and sends it back to the "MATLAB side". The communication is initiated every time from Arduino, as in a microcontroller environment it is possible to achieve accurate fixed sampling time. The way to achieve this accurate timing is described in Section A.4.2.

The "MATLAB side" is firstly loading a Simulink file, in which the user can create its own input signals. Then translates the desired signal (in every sampling time) to two bytes namely, the *PWM duty cycle* and the *direction* byte. Afterwards it open the Serial port and waits for "Arduino side" to initiate the communication.

The files that are needed to conduct someone an experiment are:

- collect_data.m
- collect_data_buffer.m
- simuling_signal_generator.slx
- plot_data.m
- collect_data_v1_0.ino

The user must first upload the *collect_data_v1_0.ino* file to the Arduino board and then (assuming signals were created in the *simuling_signal_generator.slx*) run the *collect_data.m* file in MATLAB.

After the end of the experiment he can run the *plot_data.m* to observe the results. The communication is not optimised as it was made only to serve for collecting data. After every experiment, the Arduino needs to be restarted and before re-run the *collect_data.m* file the user must "*clear all*" the MATLAB variables.

A.6.2 Data preparation

A set of data is also provided with this report, even though someone can perform its own experiments, as described in Section A.6.1. The experiments for the provided data were conducted at $7.4V$ and *sampling time* at $20ms$.

There were used three different input signals in order to test different -realistic- dynamics. Fig. A.7 shows the three input/output data sets.

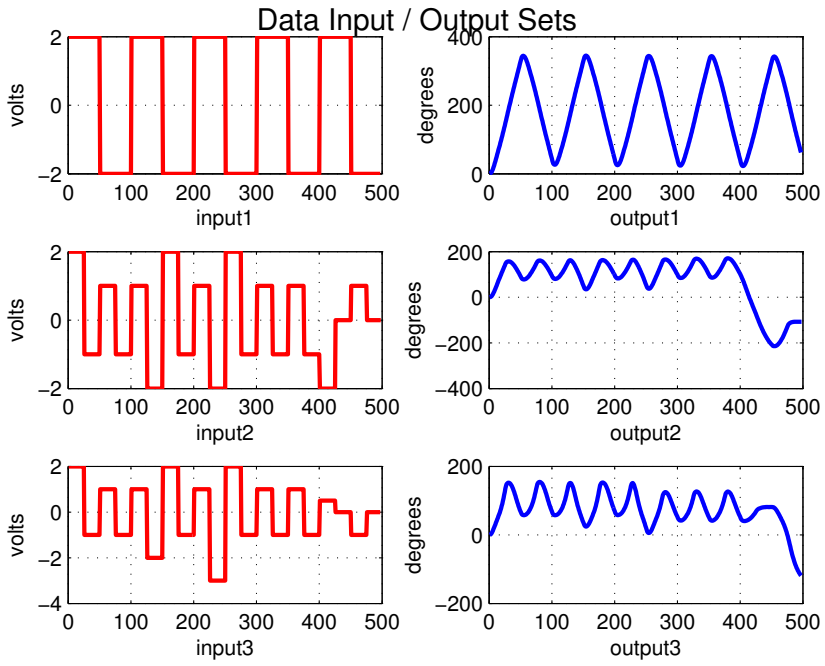


Fig. A.7: Input-Output Data set

The first data set was selected in order to estimate the transfer function. The data are in the *collect_data2.mat* file,

Listing A.9 Load the data

```
% Load the input/output data
load('estimation_data');
```

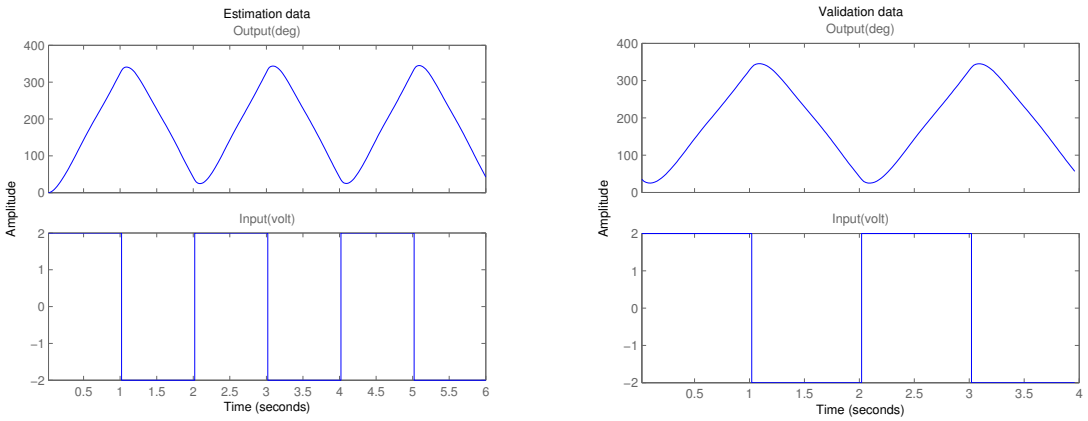
which contains the *input1* and *out_rel_deg* matrices.

The System Identification Toolbox data object iddata, encapsulate data values and data properties into a single entity. The System Identification Toolbox commands can be used, to conveniently manipulate these data objects as single entities. One part of the data will be used for the estimation and the rest for validation.

Listing A.10 Creating iddata

```
%Create identification iddata
ze = iddata(out_rel_deg(1:300,1),input1(1:300,1),Ts);
%Properties of ident. data
set(ze,'InputName','Input(volt)','OutputName','Output(deg)',...
    'InputUnit','Volt','OutputUnit','degrees','TimeUnit','seconds');
%Create validation iddata
zv = iddata(out_rel_deg(301:end,1),input1(301:end,1),Ts);
%Properties of validation data
set(zv,'InputName','Input(volt)','OutputName','Output(deg)',...
    'InputUnit','Volt','OutputUnit','degrees','TimeUnit','seconds');
%Plot
figure; plot(ze); title('Identification data');
figure; plot(zv); title('Validation data');
```

And the resulting plots,



(a) Estimation Data

(b) Validation Data

Fig. A.8: The data set seprated

A.6.3 Estimating the Empirical Step Response

Frequency-response and step-response are nonparametric models that can help someone understand the dynamic characteristics of the system. These models are not represented by a compact mathematical formula with adjustable parameters. Instead, they consist of data tables. To estimate the step response from the data, first estimate a non-parametric impulse response model (FIR filter) from data and then plot its step response.

```
%% Estimating the Empirical Step Response

% model estimation
Mimp = impulseest(ze);
% empirical step response
figure, step(Mimp);
```

As we can see from Fig. A.9 , the response of the model shows that it might be a first order system or an overdamped function.

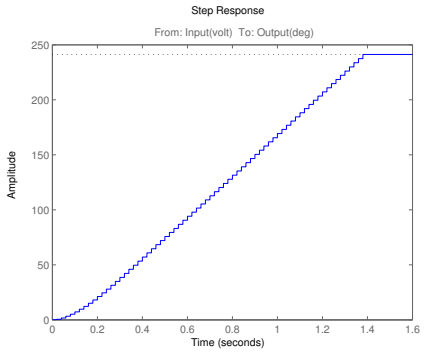


Fig. A.9: Empirical step response

A.6.4 Estimating Input/Output delays

To identify parametric black-box models, the input/output delay must be specified as part of the model order. If the input/output delays for the system are not known from the experiment, the System Identification Toolbox software can be used, to estimate the delay.

Listing A.11 Estimation of Input-Output delays

```
%Estimate delay
estimated_delay = delayest(ze) %ans=1 -> 1*Ts = 20ms delay
```

As it was expected, the result is $1 * Ts$ delay.

A.6.5 Estimate Transfer Function

At this point the data are prepared for the estimation of the transfer function. The only choice left, is the number of poles. For $np = 3$ the result is as shown in Fig. A.10.

Listing A.12 Transfer function estimation

```
%% ESTIMATE TRANSFER FUNCTION
Opt = tfestOptions('Display', 'on');
% # of poles
np = 3;
% delay
ioDelay = estimated_delay*Ts;
% Estimate the transfer function
mtf = tfest(ze, np, [], ioDelay, Opt);
figure, step(mtf);

figure, compare(zv, mtf)
```

The estimated transfer function is,

$$mtf = \frac{67.56s^2 + 1893s + 4.252e^4}{s^3 + 27.94s^2 + 231.4s + 4.586e^{-11}} \tag{A.1}$$

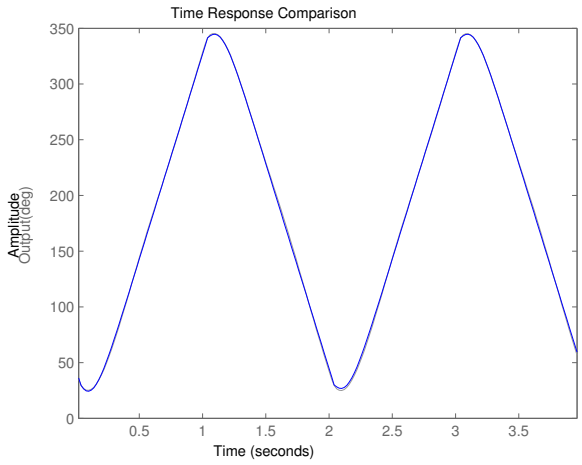


Fig. A.10: Validation Data fit to the transfer function

In Fig. A.10 and A.11 it is observed that the fit of the validation data reaches the 98.37%. The reason for this very good result is that the dynamics of the validation data are the same as the estimation data. In order to test the transfer function with different dynamics, a simple Simulink model was created, only this time, the second input data was selected as the input to the transfer function. Fig. A.12 shows the model.

In Fig. A.13 the comparison between the experiment's output and simulation's output is shown. The estimated transfer function it may not follow the dynamics as well as with the previous data set but the result is still good and suggests, that the estimated transfer function can be used for a controller design.

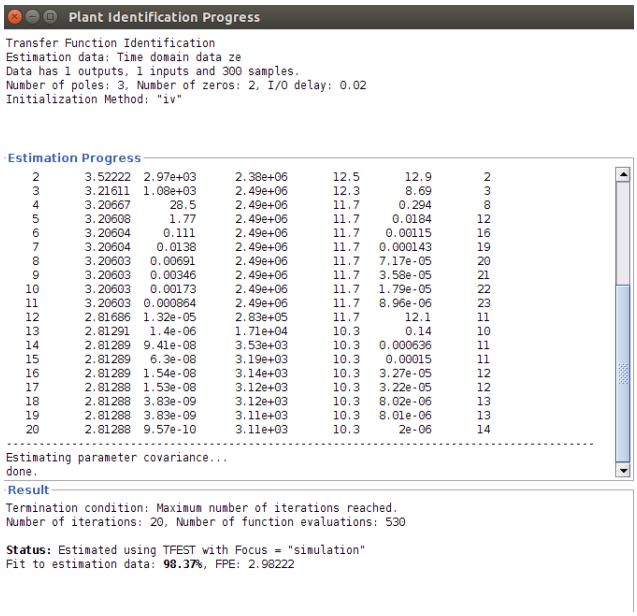


Fig. A.11: Estimation process

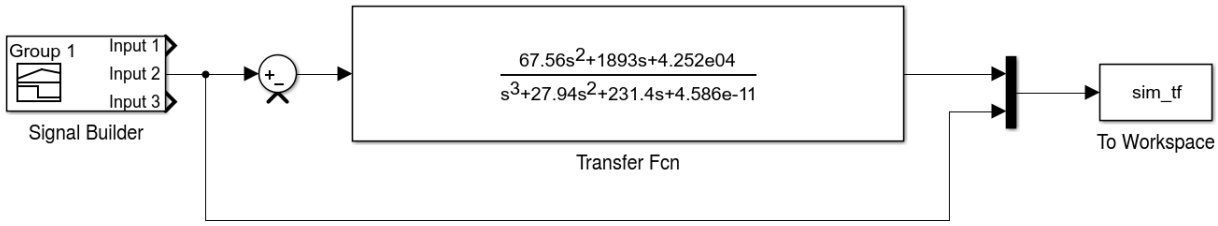


Fig. A.12: Simulink mode to validate the estimated transfer function

Parameter Estimation

As it was shown in Section A.6 it was possible to someone estimate a transfer function considering the system as a black box. If the mathematical model of the system is known, it is possible to estimate the parameters of the model, using again input-output data. On this section, this process will be described.

Simulink model

The mathematical model that was derived in Section **??** is going to be used. The implementation of this model in Simulink is pretty straightforward,

The parameters of the system that we want to estimate are:

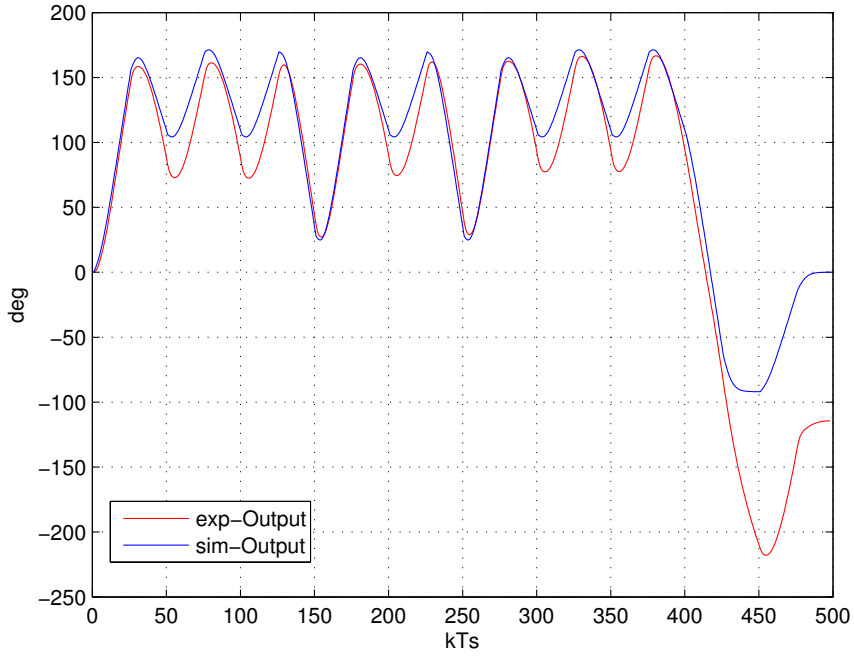


Fig. A.13: Validation of estimated transfer function

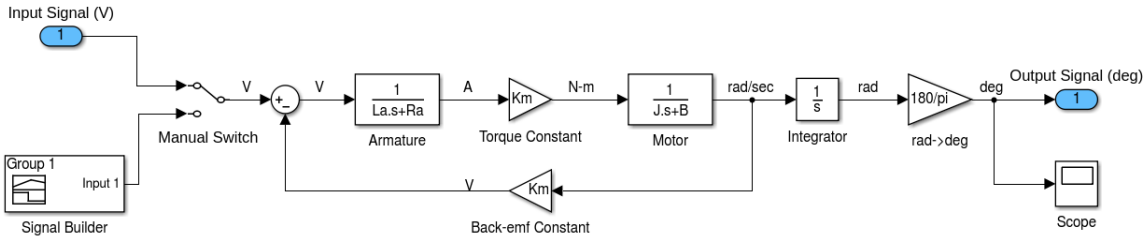


Fig. A.14: Mathematical model of the motor

B effective damping coefficient, $N{-}m\ s/rad$,
J effective inertia, $N{-}m\ s^2/A$,
K_m torque constant, $N{-}m/A$,
L_a armature inductance, H
R_a armature resistance, Ω.

In order to be able to continue we need to define initial values for these parameters in the workspace.

```
>> B = 0.008;
>> J = 5.7e-07;
>> Km = 0.0134;
>> La = 6.5e-05;
>> Ra = 1.9;
```

inp_duty	The duty cycle of the pwm signal.
inp_v_pwm	The duty cycle translated in $[0-5]$ volts.
inp_volt	The output voltage of the driver, the input voltage to the motor.
out_abs	The absolute output of the system expressed to ticks per revolution.
out_rel	The relative output of the system expressed to $[0-4098]$ ticks per revolution.
out_rel_deg	The relative output of the system expressed to $[0-360]$ degrees per revolution.

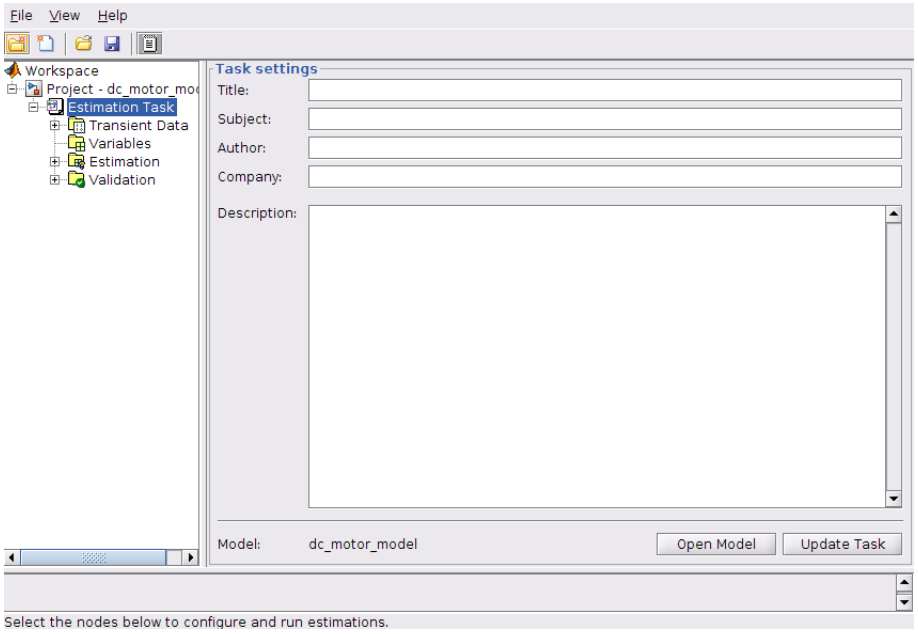
Table A.3: Data set matrices

The same data set will be used, as in Section A.6, namely *collect_data2*. The data set contains 6 matrices.

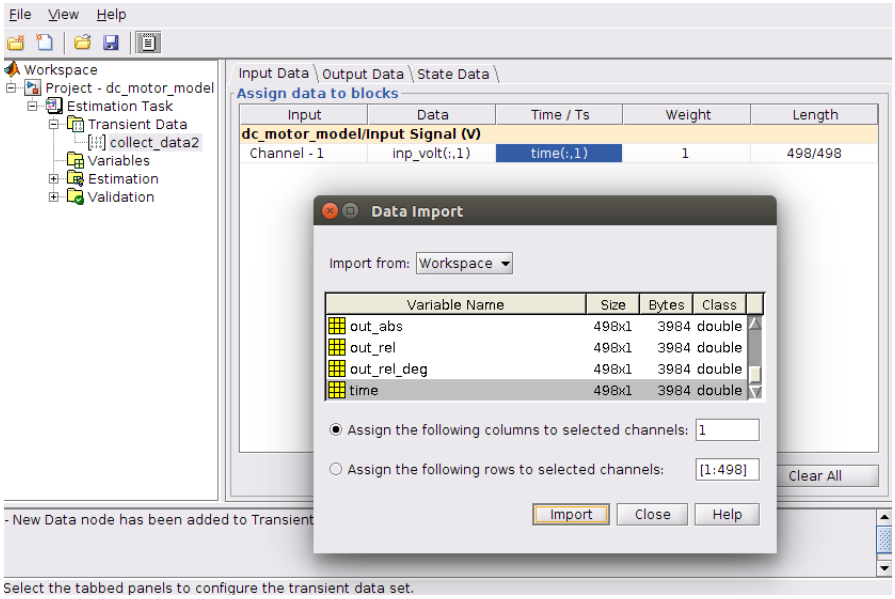
MATLAB Parameter Estimation Toolbox

The input of the model A.14 is in voltage and this voltage is the output of the driver of the motor. Therefore the *input data* that will be used will be the *inp_volt*. The chosen output is in degrees therefore the selected *output data* is *out_rel_deg*.

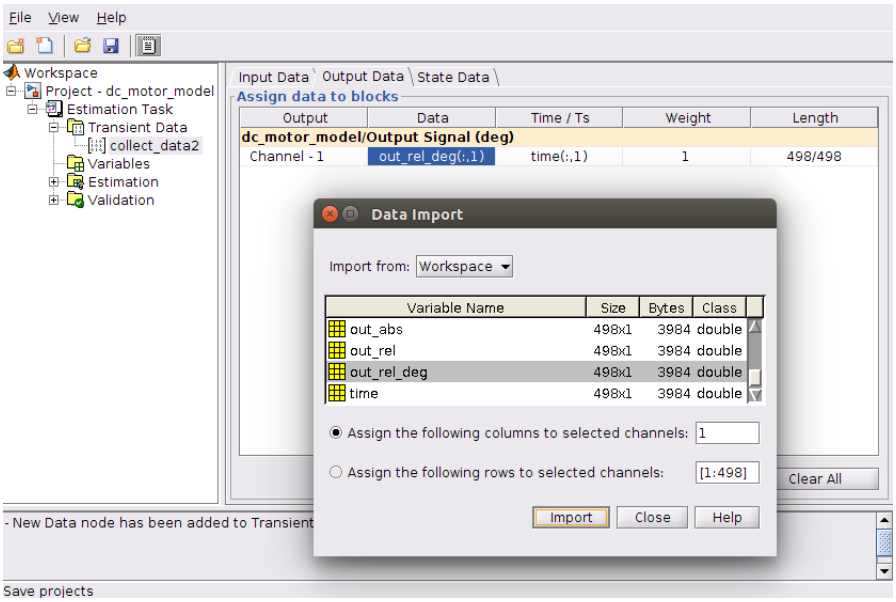
By selecting *Analysis* → *ParameterEstimation* it opens the *Parameter and Estimation Tools Manager* main window. In the first tab the user can add some information such as the title and author of the project.



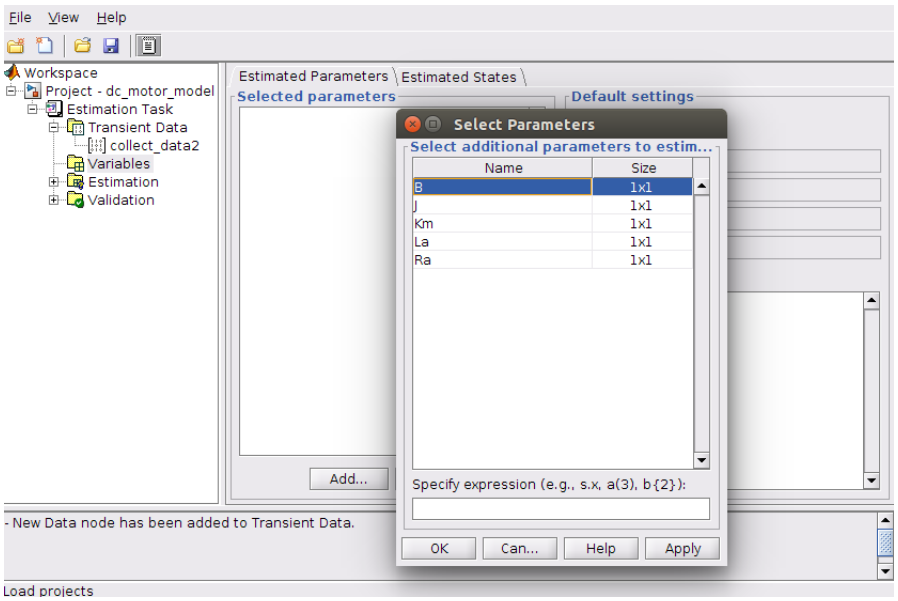
In the *Transient Data tab* a new *data* was created named *collect_data2*. And now the user is able to add the desired input/output data. First the input,

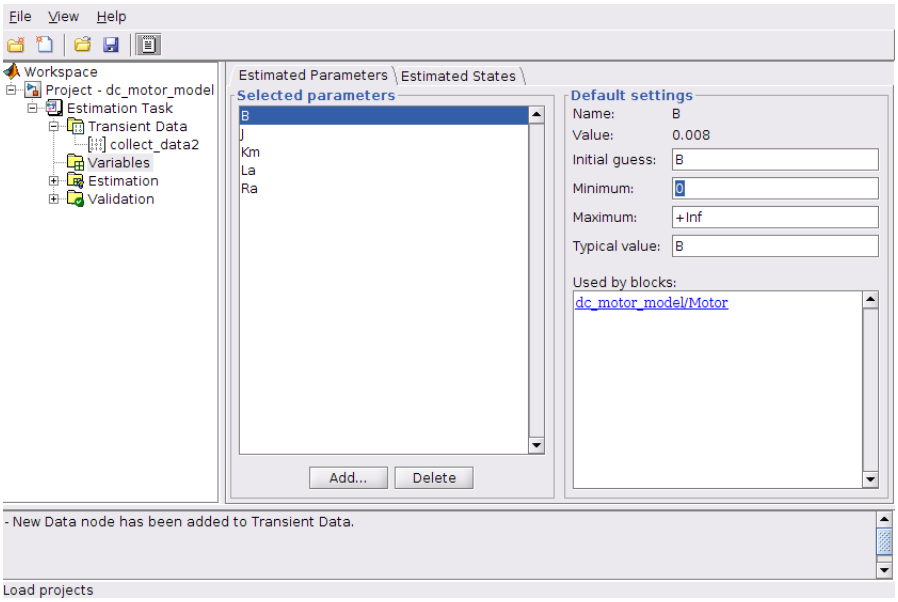


and then the output

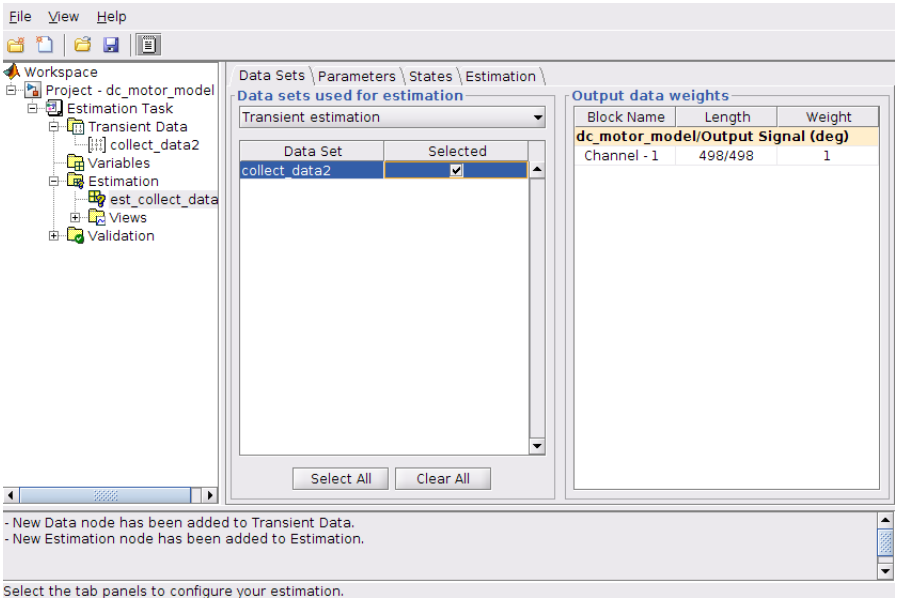


Next tab is *Variables*, where by selecting *Add* the user can choose which parameters wants to estimate. Since all these parameters are physical quantities, their minimum values are set to zero, as they cannot have negative values.

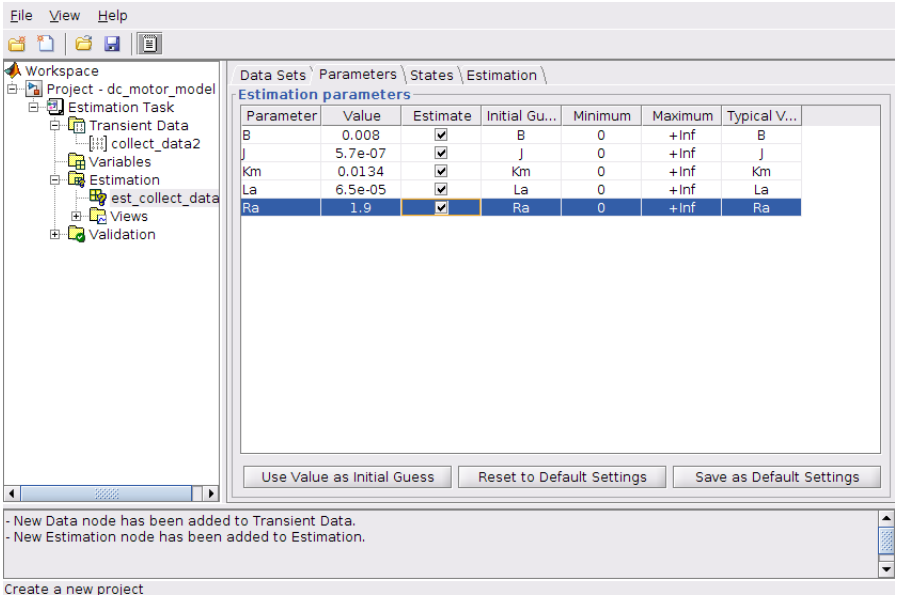




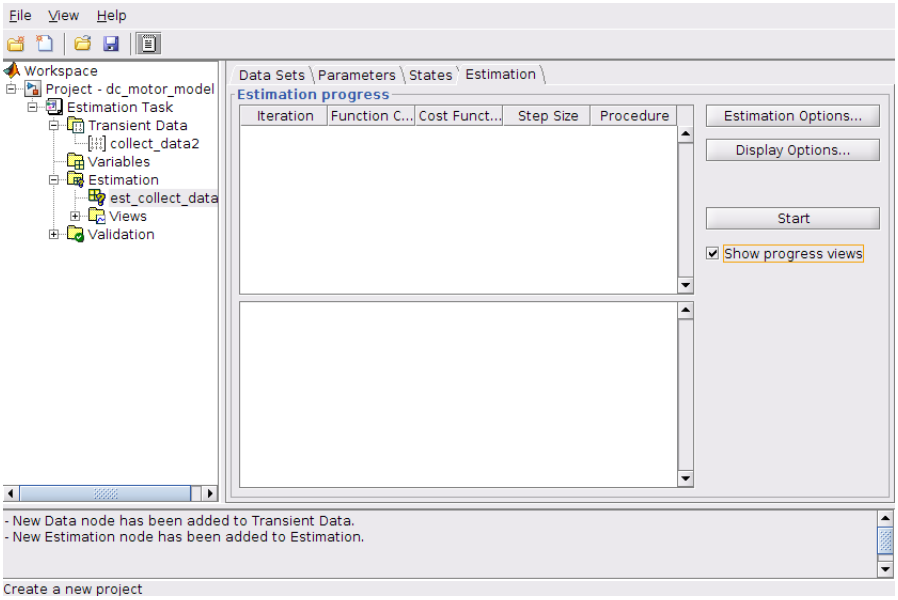
Now that the data are imported and the variables to estimate are chosen, in the *Estimation* tab a new entry created called *est_collect_data2*. There, the imported data are chosen,



as well as the parameters,



and then everything is ready for the estimation process to begin. Before of that, by selecting the *Show progress views*, the user can see the parameters trajectory and the result of the estimation during the process.



The result of the *Parameter Estimation* is shown in Fig. A.15 and Fig. A.16. It is shown that the trajectories converged relatively fast and the fit to the data is very good.

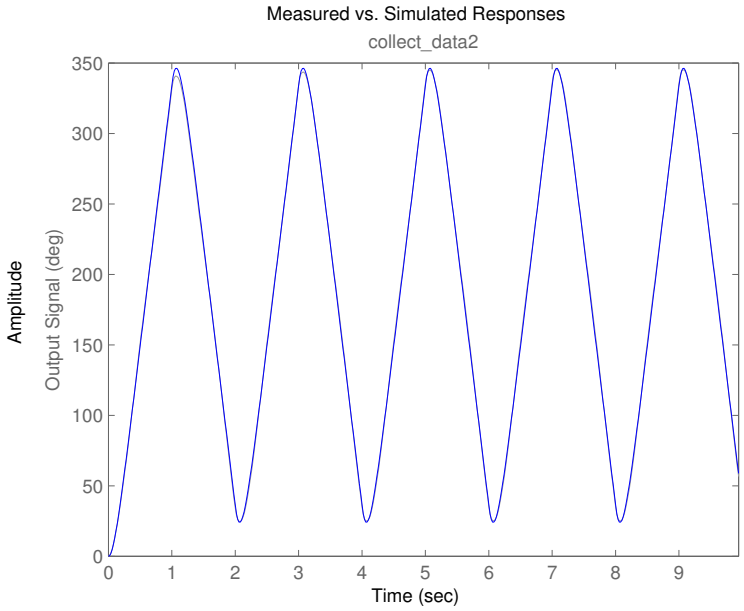


Fig. A.15: Simulation vs. Measured Responses

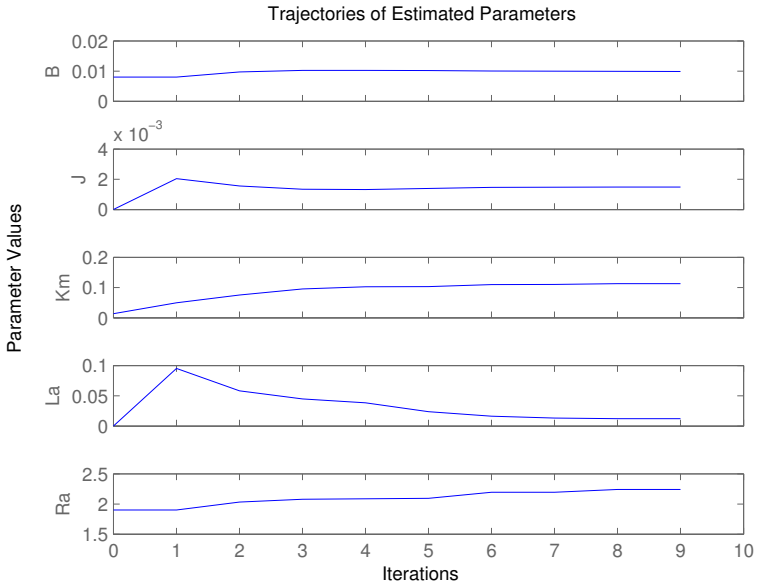


Fig. A.16: Trajectories of Estimated Parameters

The estimated parameters are,

B	0.0098949
J	0.0014829
Km	0.11262
La	0.012109
Ra	2.2397

Table A.4: Estimated parameters

Validation

In order to validate the estimated parameters, the user only has to connect the *Manual Switch* to the second input of the *Signal Builder* (which is the same one used for the data acquisition) and run the simulation, as shown in Fig. A.17.

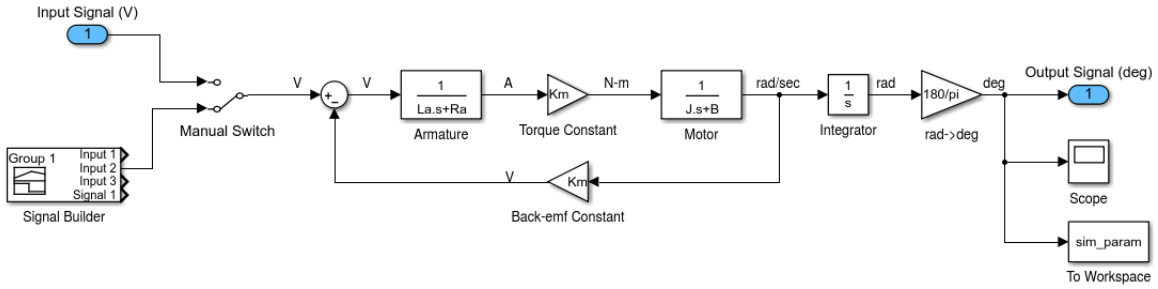


Fig. A.17: Validation model

The result is compared with the *out_rel_deg* of data set *collect_data_2_1.mat*. As it can be seen in the Fig. A.18 the result is not that good as in the case of transfer function identification. One of the reason for that, could be the unmodelled frictions in the motor. Also, the input of the system, is the *input_volt*. This voltage, that is the output of the driver of the motor, is not measured but calculated from the known PWM duty cycle. Finally, the simulink model does not contain the gear ratio. The estimated parameters are such as the effect of the gears is included in the model. All these result in not a such good fit on the measured dynamics but still good enough for someone to design a controller.

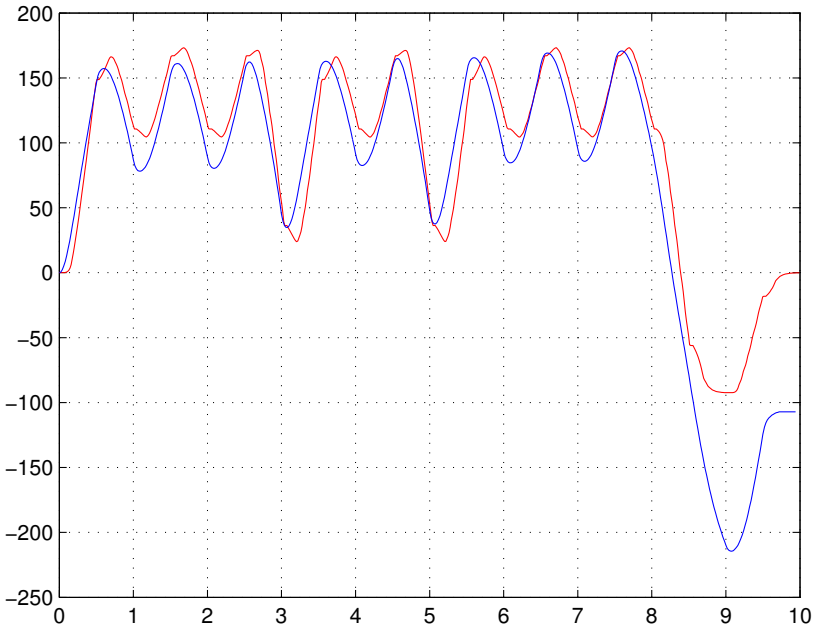


Fig. A.18: Validation of the estimated parameters

A.7 GUI

Introducing the GUI

The proper tune of a *PID* controller is most of the times a tedious process. Usually the designer, first creates the mathematical model, then designs the simulation where he is tuning the gains of the controller and finally he applies the controller to the real system. In most of the cases the result of the simulation don't coincide with the result of the real system. Some of the reasons for that could be, the uncertainty of the model or the difficulty to model some physical phenomena, such as friction, backlashes etc.

The solution for that, is that the designer must re-tune the gains empirically based on experimental results or using some other techniques. It is a process that can be time consuming, as the choice of the gains is based purely on the designer's intuition.

In order to help with this process and make it easier, an application was created, where the user can see the output data of the motor *live*, compared to the reference signal and tune the PID gains *"online"*. It also provides the option to create its own reference signals. Figure A.19 shows the main window.

The application was implemented using *Python*. More specific *PyQt4* was used. *PyQt4* is a set of *Python* bindings for *Qt*. *Qt* is a cross-platform application development framework for desktop, embedded and mobile. Someone can find more informations at [27] and [26]. The application is communicating through serial communication with an Arduino, on which the driver of the motor is connected, as well as the sensor. For this implementation was used the driver that was chosen and discussed in Section A.1 and the magnetic encoder as discussed in Section A.2. Through this section, only some of the *Python* code is presented because of its size. At the end of this section the full Arduino code is presented.

In Fig. A.19 is shown the main window of the application. It consists of 3 parts,

- The figure part, where the output data is compared with the reference signal
- The knobs part, to tune the P/I/D gains
- the configuration part, where you can choose the reference signal you want to apply, the serial port to connect to, and to start/stop the process.

Knobs section

The three knobs are to modify the P/I/D gains accordingly. There are three characteristics for each knob. The ***range***, the ***mid*** value and the ***step***. The range is expressed as $\pm val$ the *mid* value. The *step* refers to the minimum change on the knob value. For example, the original configuration of the knobs are,

Table A.5: Knobs original configuration

mid	10
range	±10
step	0.1

So with $mid = 10$ the range is $[0 - 20]$ and every movement of the knob will change the value by ± 0.1.

Under of each knob, there is a *"Line Edit"* with original value of 10. This corresponds to the mid value of the knob. For example, with the original configuration described in Table A.5, if the

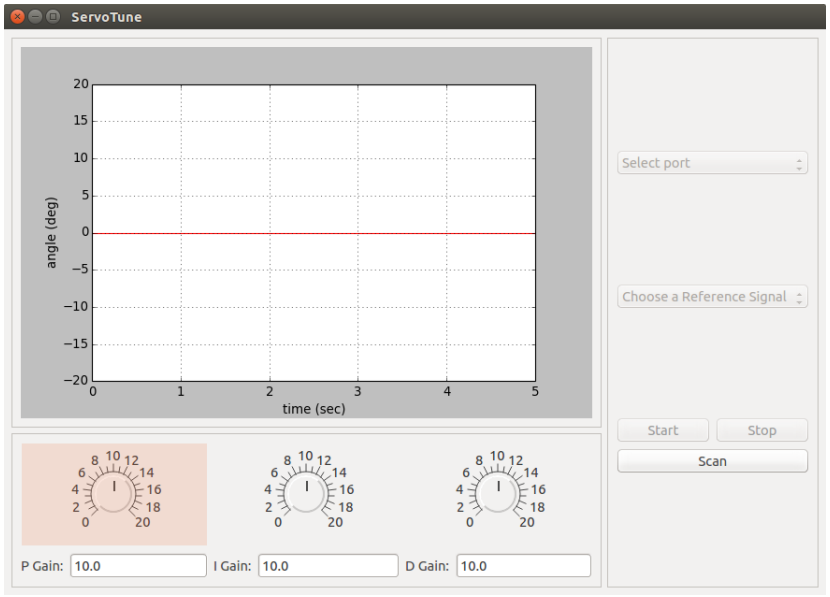
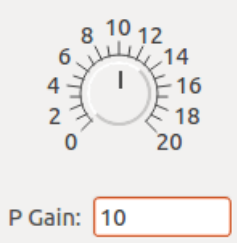
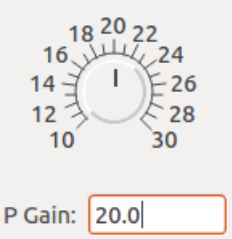


Fig. A.19: Main Window

user enters the value 20 in the *Line Edit*, the press of *enter* will result to a range of $[10 - 30]$ with $step = 0.1$ and of course $mid = 20$. The result is shown in Fig. A.20b,



(a) knob(P) original configuration



(b) knob(P) new configuration

Fig. A.20: knob(P) configuration through *mid* value

The user also has the choice to change the other two parameters (*range* and *step*). Under the *"Tools"* there are three more tabs, *knob(P)*, *knob(I)* and *knob(D)*. By clicking any of them, a new *Dialog* window opens as it is shown in Fig. A.21. This *input dialog* window is modeless, which means that does not block the main window. Therefore, the user can also reconfigure the knobs while the process is running and not only in advance.

Creating Reference Signals

Under the *Signal* option in the tool bar, the are two more tabs, ***New Reference Signal*** and ***New Periodic Reference Signal***. The first choice is to create a *custom* signal and the later allows the user to create either a *square* or a *sawtooth* signal.

New Reference signal

By selecting *New Reference Signal* the window of Fig. A.22 pops up. This windows contains,

- A *Line Edit* to enter the name of the signal
- The *Figure* that shows the signal as the user creates it
- A *Line Edit* to enter the Sampling Time
- A *Line Edit* to enter a point in the form $time\_val, angle\_val$. *Time* is the x-axis coordinate of the point and *angle* the y-axis coordinate of the point
- A button that allows the user to enter the *next* point
- A button (OK) to finish the process of crating a signal and to return to the main window.

There are some points that need to be noticed. The first one has to do with the name of the signal. In the main window, there is a *combo* box that -will- contain all the signals that the user

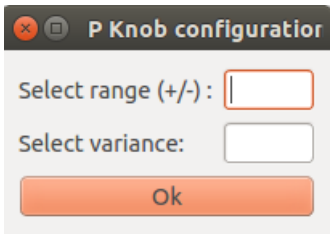


Fig. A.21: knob(P) configuration window

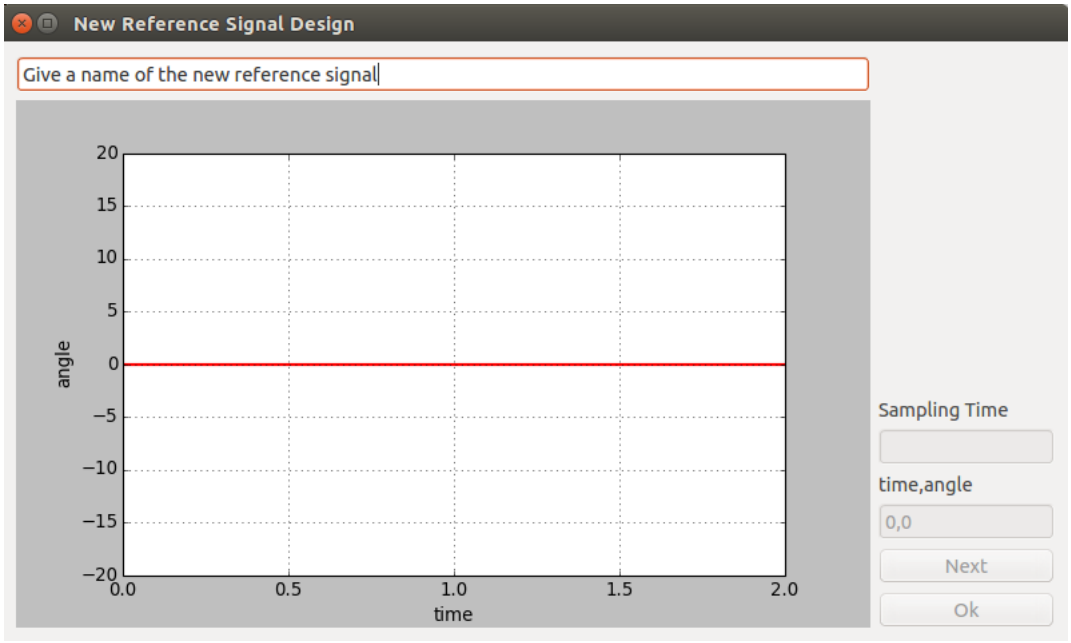


Fig. A.22: New Reference Signal Main Window

created (see Fig. A.19). In the *New reference Signal* window of Fig, A.22, by clicking the *OK* button the name of the signal will populate the list of the *combo* box in the main window. Therefore, if the user enters a name that already exist, he will not be able to continue, unless he will change the name. The second point has to do with the form of the point(time,angle). Table A.6, shows a sequence of points and Fig. A.23 the resulting signal.

Table A.6: Example of a sequence of points for custom refrence signal

0,15	1,15	1,-20	1.5,-20	1.5,10	3,10	3,0	4,0

Note!: All the reference signals start from (0,0).

The final point that needs to be noticed, is the *Sampling Time*. By clicking the button OK, except of adding the signal to the *combo* box in the main window, it also samples the line. In the example of Fig. A.23 the *Sampling Time* is 0.02 and the total duration of the signal is 4 seconds. That means that after the sampling, the signal is a *list* of $4/0.02 = 200$ points. The reason for that is explained in Section A.7.1. The value of the *Sampling Time* **must be** equal to the *control loop* as this was defined in Section A.4.2.

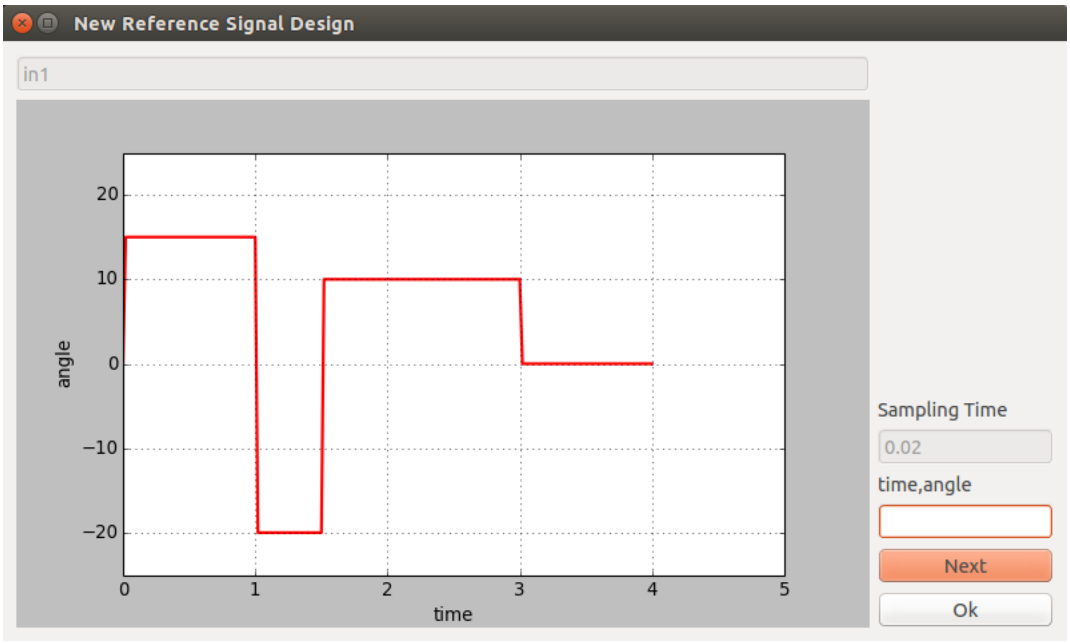


Fig. A.23: The signal results from the point of Table A.6

New Periodic Reference Signal

Apart of custom reference signals, the user also has the option to create *square* or *sawtooth* signals. By selecting **Signal→New Periodic Reference Signal** from the tool bar, a new *Dialog* window will appear, as show in Fig. A.24. For either the *square* or the *sawtooth* signal the parameters for the user to configure are,

- The *Name* of the signal
- The total *Time* of the signal
- The *Sampling Time*
- The *Amplitude* of the signal
- The *Frequency* of the signal

The *Name* of the signal is working with the same way as for the custom reference signal described in Section A.7. If the name already exists, the user cannot proceed unless he change the name. The total *Time* and the *Sampling Time* also work as described in Section A.7. And finally, the *Amplitude* and the *Frequency*, are the actual parameters of the signal. There is also a *combo* box that allows the user to choose the type of the signal.

Once all the fields are proper complete, the OK button will become active and the user will return to the main window where, if he choose in the *combo* box the signal just created, he will see the result plotted in the main figure. Fig. A.25, shows the result of the signal created in Fig. A.24.

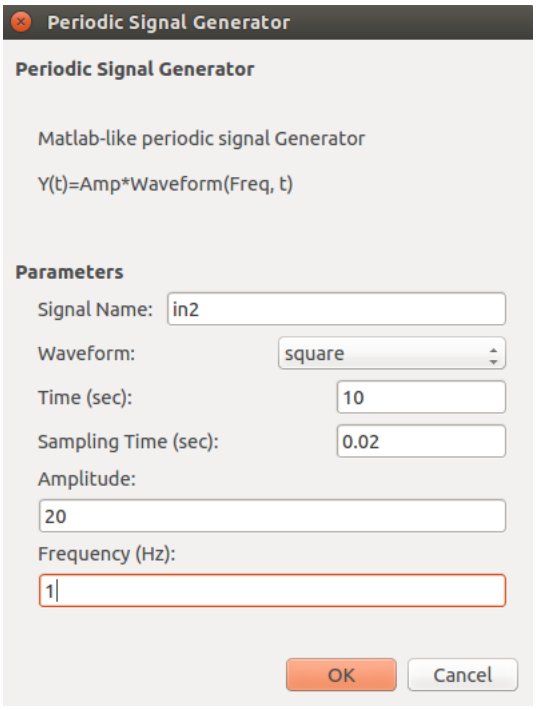


Fig. A.24: Main window for creating a Periodic Reference Signal

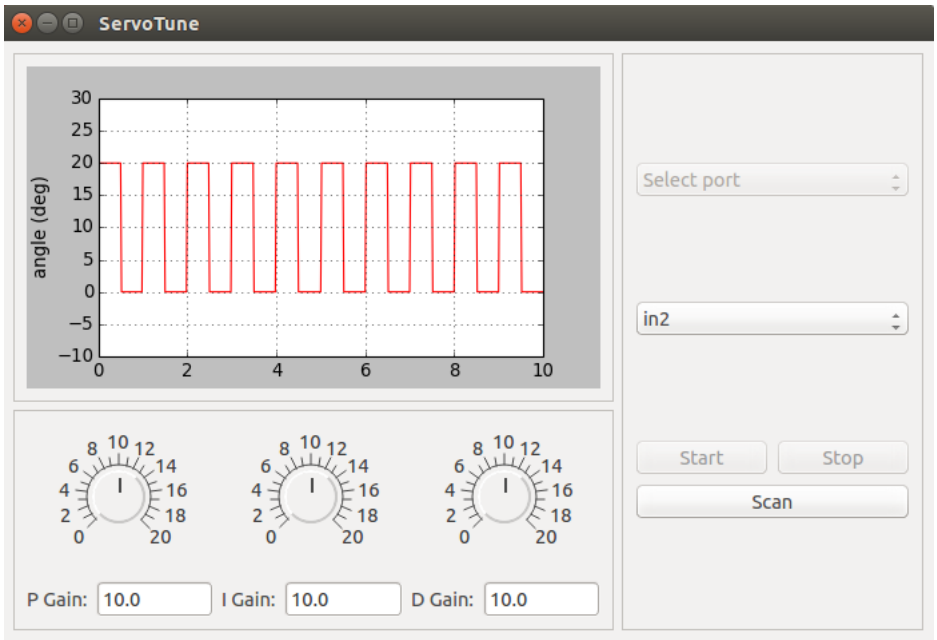


Fig. A.25: The signal created on Fig. A.24 plotted in the main figure.

A.7.1 Configuration section

On this part of the main window the user first has to *Scan* for any available port and there is a button for it. The list of all the available ports will appear in the specific *combo* box.

After the creation of at least one reference signal, it must be selected through the according *combo* box. Once selected, the signal will be plotted in the main figure.

After the selection of the port and the reference signal, the *Start* button will become active and the process is ready to begin.

Communication

The communication between the application (Python) and the controller (Arduino) was implemented through the serial interface. To achieve this, the **pySerial** library was used [29].

While it is possible to create specific timed loops in *python*, under any operating system this loop can not be guaranteed to be accurate every time. Something that is easy to achieve in a micro-controller such as Arduino, in a way that was described in Section A.4.2. For that reason, every communication between Arduino and Python is triggered by Arduino.

By pressing the *Start* button on the main window of the application, Arduino is restarted through the *pySerial* library to ensure that both sides are synchronized. After that, Arduino, for every control loop, sends once the character 's' that stands for start. If Python will receive this character, both sides are ready for the data exchange. Since the communication is 1-on-1 it is possible for both sides to know exactly what to expect from the other side. That makes simple the process of validating that the data were transferred correctly.

Python side

Since Python-application received the 's' character from Arduino, is ready to send data to it. That data is a fixed serial *word* consists of 10 bytes. Table A.7, shows an example of this serial word.

Table A.7: Serial word to be sent to Arduino

`start_cmd` (1 byte)	`reference-float` (4 bytes)	`gain_cmd` (1 byte)	`gain-float` (4 bytes)

The start command can be one of the following bytes,

Table A.8: Start commands

start_cmd	**Byte**	**Description**
`startCx_f`	`0xf0`	Full word: reference + gain
`startCx_e`	`0xf1`	"Empty" word: reference + zeros
`stopCx`	`0xf2`	Stop word: zeros + zeros

where the gain command can be one of the following bytes,

Table A.9: Gain commands

Command	Byte	Description
`P_Gain`	`0xf3`	Update the P gain
`I_Gain`	`0xf4`	Update the I gain
`D_Gain`	`0xf5`	Update the D gain
`No_Gain`	`0xf6`	No gain update

Every time, the data exchange from both sides is happening inside the "time window" of the control loop. Since this loop is at the level of milliseconds, it is impossible for the user to change the value of two knobs on the same time. Therefore, there is no need for the *serial word* to contain the float values of all three gains. If that was the case, then every *Serial word* sent from Python to Arduino would consist of 17 bytes. Instead, with the use of the *gain command* we indicate to Arduino which gain to update every time.

Also, it is obvious that most of the *serial words* that are sent from the application wouldn't contain any change in any gain value, as the user is not able to change values that fast. For that reason, the *"startCx_e"* command was introduced, to indicate to Arduino, that on this word there is only the new reference value, a useful information for the word decomposition from the Arduino side, as it is shown in Section A.7.1. The *"startCx_f"* is for the case where there is a change on one gain and the *serial word* is "full". And finally, *"stopCx"* indicates to Arduino, that this is a *serial word* with all zeros, which means that the whole reference signal was sent. Listing A.13, shows the Python code for constructing the *Serial word*.

It is clear now why the reference signal is sampled and why in Section A.7, was pointed out that the *Sampling Time* of the reference signal and the *control loop* **must be** equal. If the reference signal had different sampling time, then the timing of both the signals (reference and output) wouldn't coincide in the plot at the main figure and of course, the output wouldn't be representative of what the user would want.

The process of communication between these two sides contains mainly, data sent, data received and refreshing the main figure. All these processes are time consuming and must be complete before the new time "window" of the next control loop. If the implementation of all these code is done in a "serial" manner, then there would be time "windows" where there would be no time to check any change in the knobs. This would translate to the user as some kind of "lag" (delay) in the movement of the knobs. To solve this issue, the communication part of the code, was originally implemented using the threading interface, [10]. It was observed though, that the communication was not following the time "window" (there were times that needed two or even three control loops in order to send the new reference value). The reason of that was, as it is stated in [10],

"...due to the *Global Interpreter Lock*, only one thread can execute Python code at once...If you want your application to make better use of the computational resources of multi-core machines, you are advised to use **multiprocessing**."

Hence by using the "multiprocessing" interface [9] one processor is used for the communication process allowing to simultaneously exchange data with Arduino, refresh the main figure and change the value of any knob.

The most "*multiprocessing*"-safe way to transfer data between the *process* and the main application, is by use of Queues [9]. For that a reason a FIFO Queue was created named `gain_q` . Every

time a knob is moved, first the name ('P', 'I' or 'D') of the knob and then the value of the knob are added to the `gain_q`. Therefore if the size of the Queue is equal to 2, that means there is a change to a knob. An information used for the construction of the *Serial word*.

Listing A.13 Construction of Serial word

```
# the reference value to send
self.ref = self.reference[self.ref_counter]
# value1 is the first float to be send (the reference)
value1 = struct.pack('%sf' % 1, self.ref)

# Update the gains
if self.gain_q.qsize() == 2:
    let = self.gain_q.get()
    self.new_gain = self.gain_q.get()

    if let == 'P':
        self.command_gain = command.p
    elif let == 'I':
        self.command_gain = command.i
    elif let == 'D':
        self.command_gain = command.d
elif self.gain_q.qsize == 1:    # if size less than 2 (smth went wrong!), ...
    clear the queue
    trash = self.gain_q.get()

# If the knobs didn't change, send just the reference (startCx_e)
if self.new_gain == self.prev_gain:
    command_start = command.startCx_e
    self.command_gain = command.x
    # value2 is the second float to be send (the gain), in that case, 0.
    value2 = struct.pack('%sf' % 1, float(0))
else:
    self.prev_gain = self.new_gain
    command_start = command.startCx_f
    value2 = struct.pack('%sf' % 1, self.new_gain)

# Construct the buffer
buf = bytearray([command_start, ord(value1[0]), ord(value1[1]), ...
    ord(value1[2]), ord(value1[3]),
                 self.command_gain, ord(value2[0]), ord(value2[1]), ...
    ord(value2[2]), ord(value2[3])])
```

Arduino side

As it was already mentioned, it is very important the data exchange between *Python* and *Arduino* to be complete in every "time window" defined by Arduino as *control loop*. The *ATmega328* micro-controller (Arduino), has a **U**niveral **S**ynchronous and **A**synchronous serial **R**eceiver and **T**ransmitter (USART). The user can find more details in the datasheet [5], but what is need to be clear, is that it receives 8-bit of data in every "step".

In order to achieve the communication as fast as possible, the use of interrupts was necessary. Arduino has already implemented a *Serial* library that can be used, but it is already using the desired interrupts. Therefore, the programming of the USART had to be done manually. In order to explain how the code works, first is necessary to explain the set-up of the variables.

First the incoming data buffer is defined, ***bufferRx***

Listing A.14 Buffer to store the incoming data

```
volatile unsigned char counterRx = 0;
volatile unsigned char bufferRx[9] = {0,0,0,0,0,0,0,0,0};
```

As it is already mentioned earlier, *Python side* is sending a word of length of 10 bytes every time. The strategy is to receive the full word, and then decompose it accordingly. There is also a counter ***counterRx*** to allow indexing this buffer. The reason these variable are defined as *volatile* is in order to be able to used inside an interrupt routine.

The definition of the floats that are received and sent are shown in Listing A.15. Here is worth noticing, that for the Arduino environment, *Floating-point* numbers are stored as 32 bits (4 bytes).

Listing A.15 Floats and their pointers to be sent/received

```
// Number to be sent
float pos = 0;
unsigned char *pointer_pos = (unsigned char *)&pos;
// Numbers to be received
float ref = 0;
float *pointer_ref = (float *)&bufferRx[1];
float gain = 0;
float *pointer_gain = (float *)&bufferRx[6];
// Actual P/I/D gains
float P = 0;
float I = 0;
float D = 0;
```

The *pos* float is used to store the position value that was read from the sensor and will be sent to the *Python side*. As it is mentioned, the structure of the incoming *serial word* is well known (Table A.7). Therefore it is certain that in $bufferRx[1]$ will be the first byte of the *reference-float*. By defining a float as *ref*, we conserve 4 bytes in the memory. With the use of pointer, we can connect that space of the memory with the specific part of the buffer. With that way, the value of the float *ref*, is defined by the bytes $bufferRx[1:4]$. Similarly the value of the float *gain* is defined by the bytes $bufferRx[6:9]$.

Manual Serial

To start a *Serial* communication, first the USART module needs to be initiated. At the very top of the Arduino code, some definition are made that specify the *BAUDRATE* of the port.

Listing A.16 BAUDRATE definitions

```
// USART initialization def's
#define FOSC 16000000UL   //clock speed
#define BAUD 115200     //desired baud rate
#define MYUBRR (FOSC/4/BAUD-1)/2
```

Now it is possible to initialize the port,

Listing A.17 USART_Init

```
void USART_Init (unsigned int ubrr)
{
  UCSR0A = 0;
  UCSR0A |= (1<<U2X0);
  /* Set baud rate */
  UBRR0H = (unsigned char)(ubrr>>8);
  UBRR0L = (unsigned char)(ubrr);
  UCSR0B = B00000000;
  // Enable receiver and transmitter
  UCSR0B |= (1 << RXEN0) | (1 << TXEN0);
  UCSR0C = B00000000;
  // Set frame: 8data, 1 stp
  UCSR0C |= (1 << UCSZ01) | (1 << UCSZ00);

}
```

And a function that can be used to send data,

Listing A.18 Function to transmit data

```
void USART_Tx (char data)
{
  /* Wait for empty transmit buffer */
  while ( !(UCSR0A & (1<<UDRE0)) ) {
  }
  /* Put data into buffer, sends the data */
  UDR0 = data;
}
```

Every time there are 8 bits that are received successfully, an interrupt is triggered. These byte, is stored to the register **UDR0**. To take advantage of this interrupt, the content of *UDR0* is stored

directly to the *bufferRx*, inside the interrupt routine. All it has to be taken care for, is to increase the *counterRx* every time and set it to zero again when needed.

Listing A.19 Incoming data interrupt routine

```
ISR(USART_RX_vect) {
  bufferRx[counterRx] = UDR0;
  counterRx++;
}
```

Control Loop

There are three boolean flags, namely *com*, *txOn* and *rxOn*. The *com* flag is to indicate if the communication inside the time window of one control loop is complete. The rest two flags, are to indicate which part of the code to execute according to either if it has to receive data (*rxOn*==*true*) or it has to transmit data (*txOn*==*true*). The pieces of the code for each of the cases are following,

Listing A.20 Read and Send data

```
if (txOn==true) {
  /* Controller calculation */
    // Here you apply your controller, for example,
    pos = ref + P;
    /* --- end of Controller --- */
    for (i=0; i<4; i++) {
        USART_Tx(pointer_pos[i]);
    }
    for (i=0; i<4; i++) {
        USART_Tx(eof[i]);
    }
    txOn = false;
    sendOnce = true;
    com = false;  //finished the com, now wait for the rest of Ts
}

if (rxOn == true) {
  switch (bufferRx[0]) {
        case 0xf0:  //read reference + gain
            rxOn = false;
            ref = (*pointer_ref);
            // Gains update (one at a time)
            if (bufferRx[5] == 0xf3) {
                P = (*pointer_gain);
            }else if (bufferRx[5] == 0xf4) {
                I = (*pointer_gain);
            }else if (bufferRx[5] == 0xf5) {
                D = (*pointer_gain);
            }
            txOn = true;
```

```
            break;
        case 0xf1:  //read only reference
            rxOn = false;
            txOn = true;
            ref = (*pointer_ref);
            break;
        case 0xf2:  //stop
            rxOn = false;
            txOn = false;
            break;
    }
}
```

Appendix B

Code examples

B.1 serialProxy

This is part of the library that was created to facilitate the communication between the Odroid-C1 and the motors.

Listing B.1 serialProxy.cpp

```
#include <stdio.h>      /* Standard input/output definitions */
#include <string.h>     /* String function definitions */
#include <unistd.h>     /* UNIX standard function definitions */
#include <fcntl.h>      /* File control definitions */
#include <errno.h>      /* Error number definitions */
#include <termios.h>    /* POSIX terminal control definitions */
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdint.h>
#include <stdarg.h>
#include <iostream>
#include <stdlib.h>
#include <typeinfo>
#include <time.h>
#include <ros/ros.h>
#include <ros/package.h>
#include <ros/console.h>
#include <sstream>

#include "motor_driver/serial_proxy.h"


/* ~~ Constructor ~~ */
MotorIO::MotorIO() {

    serialDevName = "/dev/ttyUSB0"; // Default port (FTDI programmer).

}
```

```
MotorIO::MotorIO(const char * device, char *name) {
    serialDevName = device;
    /* Odroid C1 exposed UART pins
            "dev/ttyS2" -> #8 and #10
            "dev/ttyS0" -> UART molex connector
    */
    portName = name;
}


void MotorIO::start() {
    struct termios port_options;
    int16_t status;
    speed_t baud = B500000;

    fd = open(serialDevName, O_RDWR | O_NOCTTY | O_NDELAY);
    if (fd == -1) {
        ROS_INFO("Couldn't open port '%s'!", serialDevName);
    }else {
        ROS_INFO("***Device '%s' opened succesfully.", serialDevName);
    }

    fcntl(fd, F_SETFL, O_RDWR);

    // Get the current options of the port
    tcgetattr(fd, &port_options);
    /* Set the baud rate */
    cfmakeraw(&port_options);
    cfsetispeed(&port_options, baud);
    cfsetospeed(&port_options, baud);
    // Enable the receiver and set local mode
    port_options.c_cflag |= (CLOCAL | CREAD);
    /* Setting the Parity Checking */
    // No parity (8N1: 8bits, No parity, 1 stop bit)
    port_options.c_cflag &= ~PARENB;
    port_options.c_cflag &= ~CSTOPB;
    port_options.c_cflag &= ~CSIZE;
    port_options.c_cflag |= CS8;
    /* set the character size */
     /* Mask the character size bits */
    /* Setting Hardware Flow Control */
    // To disable hardware flow control (CTS, RTS):
    // port_options.c_cflag &= ~CRTSCTS;
    /* Choosing Raw Input */
    // Raw input is unprocessed. Input characters are passed through exactly as they
    // are received, when they are received.
    port_options.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG);
    /* Choosing Raw Output */
    // Raw output is selected by resetting the OPOST option in the c_oflag member:
    port_options.c_oflag &= ~OPOST;
    // port_options.c_iflag &= ~(IXON | IXOFF | IXANY);
    /* Setting Read Timeouts */
    // VMIN specifies the minimum number of characters to read. If it is set to 0,
    // then the VTIME value specifies the time to wait for every character read.
    // Note that this does not mean that a read call for N bytes will wait for N ...
        characters
    // to come in. Rather, the timeout will apply to the first character and the ...
        read call
```

```
    // will return the number of characters immediately available (up to the number ...
        you request).
    port_options.c_cc [VMIN]  = 0;  // Minimum number of characters for ...
        noncanonical read (MIN).
    port_options.c_cc [VTIME] = 1;      // Timeout in deciseconds for noncanonical ...
        read (TIME).
    //                                  1 seconds (10 deciseconds)
    /* Set the new options for the flag */
    tcsetattr (fd, TCSANOW | TCSAFLUSH, &port_options);

    ioctl (fd, TIOCMSET, &status);
    status |= TIOCM_DTR ;
    status |= TIOCM_RTS ;
    ioctl (fd, TIOCMSET, &status);

    usleep(10000);  // 10ms

    bufferTx[0] = 0xFF;
    bufferTx[1] = 0xFF;
    rx_enabled = false;
    rx_complete = false;
    // write(fd, bufferTx,2);
    // Generate the CRC16 lookup table
    generate_CRC16();
    dummy = 0x00;
}


void MotorIO::stop() {

    close(fd);
    ROS_INFO("*** Device '%s' closed succesfully.\r\n", serialDevName);

}

bool MotorIO::read_response(int16_t num) {
    int16_t rcv = read(fd, bufferRx, num);
    if (rcv != num) {
        uint8_t *pnt;
        uint8_t veces = 0x00;
        while ((rcv!=num) && (veces<0x04)) {
            pnt = &bufferRx[rcv];
            int aux_rcv = read(fd, pnt, num-rcv);
            rcv = rcv + aux_rcv;
            veces += 1;
        }
        if (veces >= 4) {
            usleep(100);
            tcflush(fd,TCIOFLUSH);
            return false;
        }else {
            rx_complete = true;
            return true;
        }

    }else{
        rx_complete = true;
        return true;
    }
```

```
}

uint8_t MotorIO::ping(uint8_t slave_address) {
    //
    // Returns the address of the pinged slave OR 0 zero if no response to ping
    //
    // printf("Scanning for Arduino in '%s'...\r\n", i2cDevName);
    // ROS_INFO("Scanning for motor in '%s'...", portName);
    dummy += 1;
    // ROS_INFO("dummy = %x", dummy);
    bufferTx[2] = slave_address;
    bufferTx[3] = TX_LENGTH;
    bufferTx[4] = PING;
    bufferTx[5] = 0x00;
    bufferTx[6] = 0x00;
    bufferTx[7] = 0x00;
    bufferTx[8] = 0x00;
    uint8_t num_send = _crcTx(bufferTx, TX_LENGTH);

    if (write(fd, bufferTx, num_send) != num_send) {
        ROS_INFO("NopEp! Couldn't send!!");
    }else {
        rx_enabled = true;
    }
    usleep(100); //required to make flush work, for some reason
    tcflush(fd, TCIOFLUSH);

    if (rx_enabled == true) {
        if (read_response(7) == true) {
            uint8_t rx_error = bufferRx[4];
            if ((_crcRx(bufferRx)) && (rx_error == 0x00)) {
                ROS_INFO("Found motor at address %x ...", bufferRx[2]);
                // return bufferRx[2];
                return 0x00;
            }else {
                if (rx_error != 0) {
                    // ROS_INFO("Error found on data packet from motor with address ...
                        %x is %x", bufferRx[2], rx_error);
                    return 0x02;
                }else {
                    // ROS_INFO("Error in CRC_rx");
                    return 0x03;
                }
                // TODO - Error debug
            }
        }else {
            // ROS_INFO("No response from %x !!", slave_address);
            return 0x01;
        }
        rx_enabled = false;
        rx_complete = false;
    }
    usleep(100); //required to make flush work, for some reason
    tcflush(fd,TCIOFLUSH);
}


uint8_t MotorIO::setDuty(uint8_t slave_address, uint8_t duty) {
    ROS_INFO("Sending duty: %x", duty);
```

```
    bufferTx[2] = slave_address;
    bufferTx[3] = TX_LENGTH;
    bufferTx[4] = DUTY;
    bufferTx[5] = duty;
    bufferTx[6] = 0x00;
    bufferTx[7] = 0x00;
    bufferTx[8] = 0x00;
    uint8_t num_send = _crcTx(bufferTx, TX_LENGTH);

    if (write(fd, bufferTx, num_send) != num_send) {
        ROS_INFO("NopEp! Couldn't send!!");
    }else {
        rx_enabled = true;
    }
    usleep(100); //required to make flush work, for some reason
    tcflush(fd, TCIOFLUSH);

    if (rx_enabled == true) {
        if (read_response(7) == true) {
            uint8_t rx_error = bufferRx[4];
            if ((_crcRx(bufferRx)) && (rx_error == 0x00)) {
                // ROS_INFO("Found motor at address %x ...", bufferRx[2]);
                // return bufferRx[2];
                return 0x00;
            }else {
                if (rx_error != 0) {
                    ROS_INFO("Error found on data packet from motor with address %x ...
                        is %x", bufferRx[2], rx_error);
                    return 0x02;
                }else {
                    ROS_INFO("Error in CRC_rx");
                    return 0x03;
                }
                // TODO - Error debug
            }
        }else {
            ROS_INFO("No response from %x !!", slave_address);
            return 0x01;
        }
        rx_enabled = false;
        rx_complete = false;
    }
    usleep(100); //required to make flush work, for some reason
    tcflush(fd,TCIOFLUSH);

}



uint8_t MotorIO::update_motors_list(uint8_t *lst) {
    int16_t idx = 0;
    ROS_INFO("Scanning for motors in '%s'...", portName);

    for (int16_t addr=0;addr<6;addr++) {
        uint8_t e = ping(addr);
        if (e == 0x00) {
            lst[idx] = addr;
        }
```

```
        idx += 1;
    }
    return idx;
}


uint8_t MotorIO::get_velocity(uint8_t slave_address, int16_t &velocity) {
    ROS_INFO("Getting velocity...");

    bufferTx[2] = slave_address;
    bufferTx[3] = TX_LENGTH;
    bufferTx[4] = GET_VEL;
    bufferTx[5] = 0x00;
    bufferTx[6] = 0x00;
    bufferTx[7] = 0x00;
    bufferTx[8] = 0x00;
    uint8_t num_send = _crcTx(bufferTx, TX_LENGTH);

    if (write(fd, bufferTx, num_send) != num_send) {
        ROS_INFO("NopEp! Couldn't send!!");
    }else {
        rx_enabled = true;
    }
    usleep(100); //required to make flush work, for some reason
    tcflush(fd, TCIOFLUSH);

    if (rx_enabled == true) {
        if (read_response(7) == true) {
            uint8_t rx_error = bufferRx[4];
            if ((_crcRx(bufferRx)) && (rx_error == 0x00)) {
                // ROS_INFO("Found motor at address %x ...", bufferRx[2]);
                // return bufferRx[2];
                int16_t *p = (int16_t *)&bufferRx[5];
                velocity = *p;
                return 0x00;
            }else {
                if (rx_error != 0) {
                    ROS_INFO("Error found on data packet from motor with address %x ...
                        is %x", bufferRx[2], rx_error);
                    return 0x02;
                }else {
                    ROS_INFO("Error in CRC_rx");
                    return 0x03;
                }
                // TODO - Error debug
            }
        }else {
            ROS_INFO("No response from %x !!", slave_address);
            return 0x01;
        }
        rx_enabled = false;
        rx_complete = false;
    }
    usleep(100); //required to make flush work, for some reason
    tcflush(fd,TCIOFLUSH);
}


uint8_t MotorIO::feedback3(uint8_t slave_address, float &pos, float &vel, float ...
```

```
    &curr) {
    bufferTx[2] = slave_address;
    bufferTx[3] = TX_LENGTH;
    bufferTx[4] = FEEDBACK3;
    bufferTx[5] = 0x00;
    bufferTx[6] = 0x00;
    bufferTx[7] = 0x00;
    bufferTx[8] = 0x00;
    uint8_t num_send = _crcTx(bufferTx, TX_LENGTH);
    if (write(fd, bufferTx, num_send) != num_send) {
        ROS_INFO("Nop! Couldn't send!!");
    }else {
        rx_enabled = true;
    }
    usleep(600);
    if (rx_enabled == true) {
        // int a = read(fd, bufferRx, 19);
        // ROS_INFO("I read %i",a);
        // if (a != 19) {
        if (read(fd, bufferRx, 19)!=19){
            ROS_INFO("No response from '%x'!",slave_address);
        }else {
            uint8_t rx_error = bufferRx[4];
            if ((_crcRx(bufferRx)) && (rx_error == 0x00)) {
                p = (float *)&bufferRx[5];
                v = (float *)&bufferRx[9];
                c = (float *)&bufferRx[13];
                pos = *p;
                vel = *v;
                curr = *c;

            }else {
                // TODO - Error debug
                ROS_INFO("Error in Rx buffer of motor with address %x", bufferRx[2]);
            }
        }
        rx_enabled = false;
    }
}


uint8_t MotorIO::set_gain(uint8_t slave_address, uint8_t gain, float val) {
    bufferTx[2] = slave_address;
    bufferTx[3] = TX_LENGTH;
    switch (gain) {
        case 'P':
            bufferTx[4] = PGAIN;
            break;
        case 'I':
            bufferTx[4] = IGAIN;
            break;
        case 'D':
            bufferTx[4] = DGAIN;
            break;
    }
    uint8_t *pnt = (uint8_t *)&val;
    for (uint8_t i=0;i<4;i++) {
        bufferTx[5+i] = *(pnt+i);
    }
```

```
    uint8_t num_send = _crcTx(bufferTx, TX_LENGTH);
    if (write(fd, bufferTx, num_send) != num_send) {
        ROS_INFO("Nop! Couldn't send!!");
    }else {
        rx_enabled = true;
    }
    usleep(600);
    if (rx_enabled == true) {
        // int a = read(fd, bufferRx, 19);
        // ROS_INFO("I read %i",a);
        // if (a != 19) {
        if (read(fd, bufferRx, 7)!=7){
            ROS_INFO("No response from '%x'!",slave_address);
        }else {
            uint8_t rx_error = bufferRx[4];
            if ((_crcRx(bufferRx)) && (rx_error == 0x00)) {
                ROS_INFO("Found motor at address %x", bufferRx[2]);
                return bufferRx[2];
            }else {
                // TODO - Error debug
                ROS_INFO("Error in Rx buffer of motor with address %x", bufferRx[2]);
                return bufferRx[2];
            }
        }
        rx_enabled = false;
    }
}


uint8_t MotorIO::goal_position(uint8_t slave_address, float pos) {
    bufferTx[2] = slave_address;
    bufferTx[3] = TX_LENGTH;
    bufferTx[4] = GOAL_POSITION;
    uint8_t *pnt = (uint8_t *)&pos;
    for (uint8_t i=0;i<4;i++) {
        bufferTx[5+i] = *(pnt+i);
    }
    uint8_t num_send = _crcTx(bufferTx, TX_LENGTH);
    // for (int i=0;i<num_send;i++) {
    //  ROS_INFO("BufferTx[%i] = %x", i, bufferTx[i]);
    // }
    if (write(fd, bufferTx, num_send) != num_send) {
        ROS_INFO("Nop! Couldn't send!!");
    }else {
        rx_enabled = true;
    }
    usleep(600);
    if (rx_enabled == true) {
        if (read(fd, bufferRx, 7)!=7){
            ROS_INFO("No response from '%x'!",slave_address);
        }else {
            uint8_t rx_error = bufferRx[4];
            if ((_crcRx(bufferRx)) && (rx_error == 0x00)) {
                // ida y vuelta succesfull
                return bufferRx[2];
            }else {
                // TODO - Error debug
                ROS_INFO("Error in Rx buffer of motor with address %x", bufferRx[2]);
                return bufferRx[2];
```

```
            }
        }
        rx_enabled = false;
    }
}

uint8_t MotorIO::goal_velocity(uint8_t slave_address, float vel) {
    bufferTx[2] = slave_address;
    bufferTx[3] = TX_LENGTH;
    bufferTx[4] = GOAL_POSITION;
    uint8_t *pnt = (uint8_t *)&vel;
    for (uint8_t i=0;i<4;i++) {
        bufferTx[5+i] = *(pnt+i);
    }
    uint8_t num_send = _crcTx(bufferTx, TX_LENGTH);
    // for (int i=0;i<num_send;i++) {
    //  ROS_INFO("BufferTx[%i] = %x", i, bufferTx[i]);
    // }
    if (write(fd, bufferTx, num_send) != num_send) {
        ROS_INFO("Nop! Couldn't send!!");
    }else {
        rx_enabled = true;
    }
    usleep(600);
    if (rx_enabled == true) {
        if (read(fd, bufferRx, 7)!=7){
            ROS_INFO("No response from '%x'!",slave_address);
        }else {
            uint8_t rx_error = bufferRx[4];
            if ((_crcRx(bufferRx)) && (rx_error == 0x00)) {
                // ida y vuelta succesfull
                return bufferRx[2];
            }else {
                // TODO - Error debug
                ROS_INFO("Error in Rx buffer of motor with address %x", bufferRx[2]);
                return bufferRx[2];
            }
        }
        rx_enabled = false;
    }
}









void MotorIO::generate_CRC16() {
    /* Calculates the CRC16 Look-up table */

    const ushort generator = 0x1021;
    for (int divident = 0; divident < 256; divident++) /* iterate over all possible ...
        input byte values 0 - 255 */
```

```
    {
        unsigned short curByte = (unsigned short)(divident << 8); /* move divident ...
            byte into MSB of 16Bit CRC */
        for (unsigned char bit = 0; bit < 8; bit++)
        {
            if ((curByte & 0x8000) != 0)
            {
                curByte <<= 1;
                curByte ^= generator;
            }
            else
            {
                curByte <<= 1;
            }
        }
        crctable16[divident] = curByte;
    }
}


bool MotorIO::_crcRx(uint8_t *buf) {
    /* Checks the incoming data if they 'agree' with the CRC
        Returns 'true' if there was no error in the data transmission,
        and 'false' otherwise.
    */

    uint8_t len = buf[3];
    int16_t crc = 0;
    int16_t *p = (int16_t *)&buf[3+len];  // points to RX_CRC

    // Check incoming CRC
    for (uint8_t i=0; i<len; i++)
    {
        uint8_t pos = (uint8_t)( (crc >> 8) ^ buf[i+3]); // equal: ((crc ^ (b << ...
            8)) >> 8)
        // Shift out the MSB used for division per lookuptable and XOR with the ...
            remainder
        crc = (int16_t)((crc << 8) ^ (int16_t)(crctable16[pos]));

    }
    // ROS_INFO("CRC is: %x", crc);
    // ROS_INFO("CRC_pointer is: %x", *p);
    if (*p == crc) {
        return true;
    }else {
        return false;
    }
}


uint8_t MotorIO::_crcTx(uint8_t *buf, uint8_t len) {
    /* Calculates the CRC16 of the buffer we send, and adds the
        two CRC bytes at the end of the given buffer.
    */

    int16_t crc = 0;
    uint8_t *p = (uint8_t *)&crc;
    // calculate crc
    for (uint8_t i=0; i<len; i++) {
```

```
        uint8_t pos = (uint8_t)( (crc >> 8) ^ buf[i+3]);
        crc = (int16_t)((crc << 8) ^ (int16_t)(crctable16[pos]));
    }
    buf[3+len] = *p;
    buf[4+len] = *(p+1);
    return 5+len;
}
```

Listing B.2 serialProxy.h

```
#include <stdint.h>
#include <stdlib.h>


#ifndef SERIAL_COM_H
#define SERIAL_COM_H


#define         PING                0x01
#define         GOAL_POSITION       0x02
#define         GOAL_VELOCITY       0x03
#define         GOAL_VEL_PROFILE    0x04
#define         DISABLE_TX          0x05
#define         ENABLE_TX           0x06

#define         FEEDBACK3           0x11

#define         PGAIN               0x21
#define         IGAIN               0x22
#define         DGAIN               0x23
#define         DUTY                0x24

#define         GET_VEL             0x25


#define         TX_LENGTH           0x06




class MotorIO {

private:
    const char* serialDevName;
    char *portName;

    int fd;
    bool rx_enabled, rx_complete;

    /* UART buffers and variables*/
    unsigned char bufferRx[32];
    unsigned char bufferTx[32];

    /* CRC variables*/
    unsigned short crctable16[256];
    void generate_CRC16();
```

```
    uint8_t _crcTx(uint8_t *, uint8_t);
    bool _crcRx(uint8_t *buf);

    bool read_response(int16_t);
public:
    const char *devFTDI;
    const char *dev1;
    const char *dev2;
    float position;
    float *p;
    //
    float velocity;
    float *v;
    //
    float current;
    float *c;
    uint8_t dummy;
    //
    uint8_t motors_left[6], motors_right[6];
    // Constructor
    MotorIO();
    MotorIO(const char *, char *);


    void start();
    void stop();
    uint8_t ping(uint8_t);
    //
    uint8_t goal_position(uint8_t, float);
    uint8_t goal_velocity(uint8_t, float);
    uint8_t set_gain(uint8_t, uint8_t, float);
    //
    uint8_t feedback3(uint8_t, float &, float &, float &);
    uint8_t update_motors_list(uint8_t *);
    uint8_t setDuty(uint8_t, uint8_t);
    uint8_t get_velocity(uint8_t, int16_t &);



};

#endif
```

B.2 Arduino code

This is the Arduino code that was used to communicate with the Odroid. It handles proper the aperiodic task of Odroid requesting data.

Listing B.3 serial odroid (main file

```
// USART initialization def's
#define FOSC 20000000UL   //clock speed
```

```
#define BAUD 500000     //desired baud rate
//#define MYUBRR FOSC/16/BAUD-1
//#define MYUBRR (FOSC/4/BAUD-1)/2
//#define MYUBRR (FOSC/8/BAUD)-1
#define MYUBRR  0x004

#define ADDR    0x14
#define IN_A 4
#define IN_B A3


/* ********************** Variables ********************** */

/*          Control loop            */
boolean control_loop = false;
/*        Magnetic encoder          */
uint8_t u8byteCount;
uint8_t u8data;
uint32_t u32result = 0;
uint32_t u32send;
/*        SerialCom buffers         */

/*        Control variables         */
float position = 180.55;
uint8_t *p = (uint8_t *)&position;
float goal_position;
float velocity = 181.55;
uint8_t *v = (uint8_t *)&velocity;
float goal_velocity;
float current = 182.55;
uint8_t *c = (uint8_t *)&current;
float _PGAIN;
float _IGAIN;
float _DGAIN;


/*        USART variables         */
unsigned char bufferRx[32];
unsigned char bufferTx[32];
unsigned char counterRx = 0;
volatile boolean write_flag = false;
volatile boolean itsMe = false;
uint8_t num_send = 0;
boolean eRX = false;  // end of incoming buffer
volatile uint8_t error = 0;
float *rx_pnt32;

/* Counter2 compare match interrupt - for control loop*/
ISR(TIMER2_COMPA_vect) {
//  delayMicroseconds(70);  // read sensor time
  if (control_loop == 0){
    control_loop = 1;
  }
}

ISR(TIMER1_COMPA_vect) {
  // Disable the counter
//  digitalWrite(12, HIGH);
  sei();
```

```
  digitalWrite(13, LOW);
  TCCR1B = 0x00;
  TCNT1 = 0x00;
  if (_crcRx(bufferRx)) { // CRC the incoming data
    error = 0x00;
    if (bufferRx[2] == ADDR) {
      itsMe = true;
    }else {
      itsMe = false;
    }
  }else {
    error = 0x09; // error in checksum
    if (bufferRx[2] == ADDR) {
      itsMe = true;
    }else {
      itsMe = false;
    }
  }

  if (itsMe == true) {
    UCSR0B |= (1<<TXEN0);
    uint8_t inst = bufferRx[4];
    num_send = fill_bufferTx(inst);
    digitalWrite(10, HIGH);
    if (write_flag == true) {
      write_flag = false;
      for (uint8_t i=0; i<num_send; i++) {
        USART_Tx(bufferTx[i]);
      }
      UCSR0B &= ~(1<<TXEN0);
    }
    digitalWrite(10, LOW);
  }
  itsMe = false;
}

ISR(USART_RX_vect) {
  bufferRx[counterRx] = UDR0;
  counterRx++;
//  digitalWrite(10, !digitalRead(10));
  // Incoming buffer is of fixed size (11)
//  if (counterRx == 13) {
//    eRX = true;
//    TCNT1 = 0x00;
//    // Start timer1
//    TCCR1B |= B00001011;
//    counterRx = 0;
//  }
  if (counterRx>2) {
    if ((bufferRx[counterRx-1]==0x0F) && (bufferRx[counterRx-2]==0x0f)) {
      counterRx = 0;
      eRX = true;
      TCNT1 = 0x00;
      // Start timer1
      digitalWrite(13, HIGH);
      TCCR1B |= B00001011;
    }
  }
```

```
}


void setup() {
  pinMode(10,OUTPUT);  // SPI pulse
  pinMode(4, OUTPUT);  // Direction pin
  pinMode(A3, OUTPUT);  // Direction pin
  pinMode(11,OUTPUT);  // OC2A
  pinMode(3, OUTPUT);  // PWM
  pinMode(13, OUTPUT);
  pinMode(12, OUTPUT);

  /* --------------------  Serial Iinit.  --------------------- */
  USART_Init(MYUBRR);

  /* ----------  Timer 2 (control loop) configuration  -------- */
  TCCR2A = 0;
  TCCR2B = 0;    //DON'T KNOW WHY - Init. Value of reg = B00000100 (!!)

  TCCR2A |= B00000010;
  TCCR2B |= B00000111;
  // DONT FORGET
  TIMSK2 |= B00000010;
  control_loop = 1;
//  OCR0A = 155; // 14 = 1ms, 155 = 10ms
  OCR2A = 96; // (prescaler:111 -> 4.983ms period

  /* ------------------  SPI configuration  ------------------- */
//  DDRB = 0;  //Check if needed
//  // configure SCK(PB5) and Slave Select(PB2) as output, MISO(PB4) as input
//  DDRB = (1 << PB5) | (1 << PB2) | (0 << PB4);
//  // configure SPI as master, SPR0=1 -> fosc/16 CHANGE: SPR0 = 0 -> fosc/4
//  // 16 Mhz XTAL:
////  SPCR = (1 << SPE) | (1 << MSTR) | (0 << CPOL) | (1 << SPR0) | (CPHA << 1);
//  // 20 Mhz XTAL:
//  SPCR = (1 << SPE) | (1 << MSTR) | (0 << CPOL) | (1 << SPR1) | (CPHA << 1);

  /* ---------- Timer 0 - Configuration (FAST_PWM) ------------ */
//  TCCR0A = 0;
//  TCCR0B = 0;    //DON'T WHY - Init. Value of reg = B00000100 (!!)
//  TCCR0A |= B00110011;
//  TCCR0B |= B00000010;  // 9.8Khz (bridge works up to 20 MHz BUT...
  /* ------------------------------------------------------------*/

  /* ---------- Timer 1 - Configuration (FAST_PWM) ------------ */
  TCCR1A = 0;
  TCCR1B = 0;    //DON'T WHY - Init. Value of reg = B00000100 (!!)
  TCCR1A |= B00100000;
//  TCCR2B |= B00000010;
  OCR1A = 5;  // for counting 20us (with 64 presc.) before ISR
  TIMSK1 |= B00000010;
  /* ------------------------------------------------------------*/


  bufferTx[0] = 0xFF;
  bufferTx[1] = 0xFF;
  bufferTx[2] = ADDR;
  sei();
}
```

```
void loop() {
  if (control_loop == true) {

    // End of Control loop 'TASK'
    control_loop = false;
  }

}

void position_update(int16_t &position) {
  uint8_t u8data;  uint32_t u32result;
  // Pulse to initiate new transfer
  digitalWrite(10, HIGH);
  digitalWrite(10, LOW);
  //Receive the 3 bytes (AS5145 sends 18bit word)
  for (uint8_t byteCount=0; byteCount<3; byteCount++) {
     u32result <<= 8;  // left shift the result so far - first time shifts ...
   0's-no change
     SPDR = 0xFF;  // send 0xFF as dummy (triggers the transfer)
     while ( (SPSR & (1 << SPIF)) == 0);  // wait until transfer complete
     u8data = SPDR;  // read data from SPI register
     u32result |= u8data;  //store the byte
  }
  // TODO!  Check the flags before continue
  u32result >>= 12;
  int *ssi_pnt16 = (int *)&u32result;
  position = *ssi_pnt16;
}

int velocity_update(int pos, int prev_pos) {
  if ((pos - prev_pos) < -400) {
    pos = pos + 4096;
  }else if ((pos - prev_pos) > 3700) {
    prev_pos = prev_pos + 4096;
  }
  return (pos-prev_pos);
}



void USART_Init (unsigned int ubrr)
{
  UCSR0A = 0;
  UCSR0A |= (1<<U2X0);
  /* Set baud rate */
  UBRR0H = (unsigned char)(ubrr>>8);
  UBRR0L = (unsigned char)(ubrr);
  UCSR0B = B00000000;
  // Enable receiver and transmitter
  UCSR0B |= (1 << RXEN0) | (1 << TXEN0);
  // Enable receiver only
//  UCSR0B |= (1 << RXEN0);
  // Enable RX Complete Interrupt
  UCSR0B |= (1 << RXCIE0);
  UCSR0C = B00000000;
  // Set frame: 8data, 1 stp
  UCSR0C |= (1 << UCSZ01) | (1 << UCSZ00);
```

```
}

void USART_Tx (char data)
{
  /* Wait for empty transmit buffer */
  while ( !(UCSR0A & (1<<UDRE0)) ) {
  }
  /* Put data into buffer, sends the data */
  UDR0 = data;
}
```

Listing B.4 CRC functions

```
unsigned short crctable16[256] = {
  0x0000, 0x1021, 0x2042, 0x3063, 0x4084, 0x50A5, 0x60C6, 0x70E7,
  0x8108, 0x9129, 0xA14A, 0xB16B, 0xC18C, 0xD1AD, 0xE1CE, 0xF1EF,
  0x1231, 0x0210, 0x3273, 0x2252, 0x52B5, 0x4294, 0x72F7, 0x62D6,
  0x9339, 0x8318, 0xB37B, 0xA35A, 0xD3BD, 0xC39C, 0xF3FF, 0xE3DE,
  0x2462, 0x3443, 0x0420, 0x1401, 0x64E6, 0x74C7, 0x44A4, 0x5485,
  0xA56A, 0xB54B, 0x8528, 0x9509, 0xE5EE, 0xF5CF, 0xC5AC, 0xD58D,
  0x3653, 0x2672, 0x1611, 0x0630, 0x76D7, 0x66F6, 0x5695, 0x46B4,
  0xB75B, 0xA77A, 0x9719, 0x8738, 0xF7DF, 0xE7FE, 0xD79D, 0xC7BC,
  0x48C4, 0x58E5, 0x6886, 0x78A7, 0x0840, 0x1861, 0x2802, 0x3823,
  0xC9CC, 0xD9ED, 0xE98E, 0xF9AF, 0x8948, 0x9969, 0xA90A, 0xB92B,
  0x5AF5, 0x4AD4, 0x7AB7, 0x6A96, 0x1A71, 0x0A50, 0x3A33, 0x2A12,
  0xDBFD, 0xCBDC, 0xFBBF, 0xEB9E, 0x9B79, 0x8B58, 0xBB3B, 0xAB1A,
  0x6CA6, 0x7C87, 0x4CE4, 0x5CC5, 0x2C22, 0x3C03, 0x0C60, 0x1C41,
  0xEDAE, 0xFD8F, 0xCDEC, 0xDDCD, 0xAD2A, 0xBD0B, 0x8D68, 0x9D49,
  0x7E97, 0x6EB6, 0x5ED5, 0x4EF4, 0x3E13, 0x2E32, 0x1E51, 0x0E70,
  0xFF9F, 0xEFBE, 0xDFDD, 0xCFFC, 0xBF1B, 0xAF3A, 0x9F59, 0x8F78,
  0x9188, 0x81A9, 0xB1CA, 0xA1EB, 0xD10C, 0xC12D, 0xF14E, 0xE16F,
  0x1080, 0x00A1, 0x30C2, 0x20E3, 0x5004, 0x4025, 0x7046, 0x6067,
  0x83B9, 0x9398, 0xA3FB, 0xB3DA, 0xC33D, 0xD31C, 0xE37F, 0xF35E,
  0x02B1, 0x1290, 0x22F3, 0x32D2, 0x4235, 0x5214, 0x6277, 0x7256,
  0xB5EA, 0xA5CB, 0x95A8, 0x8589, 0xF56E, 0xE54F, 0xD52C, 0xC50D,
  0x34E2, 0x24C3, 0x14A0, 0x0481, 0x7466, 0x6447, 0x5424, 0x4405,
  0xA7DB, 0xB7FA, 0x8799, 0x97B8, 0xE75F, 0xF77E, 0xC71D, 0xD73C,
  0x26D3, 0x36F2, 0x0691, 0x16B0, 0x6657, 0x7676, 0x4615, 0x5634,
  0xD94C, 0xC96D, 0xF90E, 0xE92F, 0x99C8, 0x89E9, 0xB98A, 0xA9AB,
  0x5844, 0x4865, 0x7806, 0x6827, 0x18C0, 0x08E1, 0x3882, 0x28A3,
  0xCB7D, 0xDB5C, 0xEB3F, 0xFB1E, 0x8BF9, 0x9BD8, 0xABBB, 0xBB9A,
  0x4A75, 0x5A54, 0x6A37, 0x7A16, 0x0AF1, 0x1AD0, 0x2AB3, 0x3A92,
  0xFD2E, 0xED0F, 0xDD6C, 0xCD4D, 0xBDAA, 0xAD8B, 0x9DE8, 0x8DC9,
  0x7C26, 0x6C07, 0x5C64, 0x4C45, 0x3CA2, 0x2C83, 0x1CE0, 0x0CC1,
  0xEF1F, 0xFF3E, 0xCF5D, 0xDF7C, 0xAF9B, 0xBFBA, 0x8FD9, 0x9FF8,
  0x6E17, 0x7E36, 0x4E55, 0x5E74, 0x2E93, 0x3EB2, 0x0ED1, 0x1EF0
};

//static unsigned short CRC16(uint8_t *buf)
boolean _crcRx(uint8_t *buf)
{

    uint8_t len = buf[3];
    int16_t crc = 0;
    int16_t *p = (int16_t *)&buf[3+len];  // points to RX_CRC
```

```
    // Check incoming CRC
    for (uint8_t i=0; i<len; i++)
    {
        /* XOR-in next input byte into MSB of crc, that's our new ...
    intermediate divident */
        uint8_t pos = (uint8_t)( (crc >> 8) ^ buf[i+3]); /* equal: ((crc ^ ...
    (b << 8)) >> 8) */
        /* Shift out the MSB used for division per lookuptable and XOR with ...
    the remainder */
        crc = (int16_t)((crc << 8) ^ (int16_t)(crctable16[pos]));

    }
    if (*p == crc) {
      return true;
    }else {
      return false;
    }
}

uint8_t _crcTx(uint8_t *buf, uint8_t len) {
  int16_t crc = 0;
  // calculate crc
  for (uint8_t i=0; i<len; i++) {
     uint8_t pos = (uint8_t)( (crc >> 8) ^ buf[i+3]);
     crc = (int16_t)((crc << 8) ^ (int16_t)(crctable16[pos]));
  }
  uint8_t *p = (uint8_t *)&crc;
  buf[3+len] = *p;
  buf[4+len] = *(p+1);
  return 5+len;
}
```

Listing B.5 fill buffer tx

```
uint8_t fill_bufferTx(uint8_t inst) {
  uint8_t num;
  switch (inst) {
    case 0x01:                                  // PING
      bufferTx[3] = 0x06;
      bufferTx[4] = error;
      bufferTx[5] = bufferRx[5];
      bufferTx[6] = 0x02;
      bufferTx[7] = 0x03;
      bufferTx[8] = 0x04;
      num = _crcTx(bufferTx, 0x06);
      write_flag = true;
      break;
    case 0x02:                                  // GOAL_POSITION
      rx_pnt32 = (float *)&bufferRx[5];
      goal_position = *rx_pnt32;
      bufferTx[3] = 0x02;
      bufferTx[4] = error;
      num = _crcTx(bufferTx, 0x02);
      write_flag = true;
```

```
      break;
    case 0x03:                                  // GOAL_POSITION
      rx_pnt32 = (float *)&bufferRx[5];
      goal_velocity = *rx_pnt32;
      bufferTx[3] = 0x02;
      bufferTx[4] = error;
      num = _crcTx(bufferTx, 0x02);
      write_flag = true;
      break;
    case 0x11:                                  // FEEDBACK3
      bufferTx[3] = 14;
      bufferTx[4] = error;
      for (uint8_t i=0;i<4;i++) {
        bufferTx[5+i] = *(p+i);
        bufferTx[9+i] = *(v+i);
        bufferTx[13+i] = *(c+i);
      }
      num = _crcTx(bufferTx, 14);

      write_flag = true;
      break;
    case 0x21:                                  // SET_GAIN(P)
      // Update the gain
      rx_pnt32 = (float *)&bufferRx[5];
      _PGAIN = *rx_pnt32;
      bufferTx[3] = 0x02;
      bufferTx[4] = error;
      num = _crcTx(bufferTx, 0x02);
      write_flag = true;
      break;
    case 0x22:                                  // SET_GAIN(I)
      // Update the gain
      rx_pnt32 = (float *)&bufferRx[5];
      _DGAIN = *rx_pnt32;
      bufferTx[3] = 0x02;
      bufferTx[4] = error;
      num = _crcTx(bufferTx, 0x02);
      write_flag = true;
      break;
    case 0x23:                                  // SET_GAIN(D)
      // Update the gain
      rx_pnt32 = (float *)&bufferRx[5];
      _IGAIN = *rx_pnt32;
      bufferTx[3] = 0x02;
      bufferTx[4] = error;
      num = _crcTx(bufferTx, 0x02);
      write_flag = true;
      break;

  }
//  if (num == 0x07) {
//    digitalWrite(10, HIGH);
//  }
  return num;
}
```