Final Master Thesis

MASTER'S DEGREE IN AUTOMATIC CONTROL AND ROBOTICS

Simulation of the Assistance of an Exoskeleton on Lower Limbs Joints Using OpenSim

MEMORY

Author: Dídac Coll Pujals
Advisors: Dr. Joan Aranda López
Manuel Vinagre Ruiz

Date: September, 2017

Escola Tècnica Superior
d'Enginyeria Industrial de Barcelona

Abstract

The huge amount of people who suffer from some form of disability implies that more research has to be done in order to develop efficient robotic technologies for them. Yet, research costs can not be neglected. In this matter, software simulators play a significant role as an alternative approach to minimize such expenses.

This project aims to make use of OpenSim API and Simbody API, two C++ libraries in order to prove the feasibility of coupling and simulating an exoskeleton model with a musculoskeletal model. Due to the few information related about how to get it, this thesis embraces many others sub-goals strictly related to the achievement of the primary purpose.
The first sub-objective has been to learn OpenSim and Simbody API. As solution proposed in this project has been to create a double pendulum model in OpenSim.
The second one has been design a model of an exoskeleton. As a result, it has been implemented in OpenSim, and a controller has been designed to test it.

The successful achievement of the previous sub-objectives have allowed to create the simulation of the exoskeleton coupled to the musculoskeletal model and simulate it in OpenSim.

Last but not least, the documentation throughout the project in conjunction with the successful achievement of the proposed goals has made possible the creation of this thesis which indeed aims to keep working with the exoskeleton coupled to musculoskeletal model or even become a reference in future robotic orthosis projects build in OpenSim and Simbody APIs.

Contents

List of Figures

List of Tables

Acronyms

AC Alternating Current

API Application Programming Interface

CAN Controller Area Network

DC Direct Current

DoF Degree of Freedom

GUI Graphical User Interface

IMU Inertial Measurement Unit

NCSRR National Center for Simulation in Rehabilitation Research

obj Object File

PID Proportional Integral Derivative

PWM Pulse Width Modulation

SDK Software Development Kit

SimTK Simbios Biosimulation Toolkit

Glossary

$C(\Theta, \dot{\Theta})$ general coriolis and centrifugal matrix.

$M(\Theta)$ general mass and inertia matrix.

$N(\Theta)$ general gravity matrix.

$T(\Theta, \dot{\Theta})$ total kinetic energy.

T_i kinetic energy in ith body.

$U(\Theta)$ matrix of inputs

$V(x)$ Lyapunov energy function.

$\mathcal{L}\left(\Theta, \dot{\Theta}\right)$ lagrangian equation.

u input computed by the control law.

FreeCad FreeCAD is a parametric 3D modeler made primarily to design real-life objects of any size. Parametric modeling allows you to easily modify your design by going back into your model history and changing its parameters. FreeCAD is open-source and highly customizable, scriptable and extensible.[23]

OpenSim is a tool for modeling simulation of movement. It has to be distinguished between OpenSim API, and OpenSim GUI. When it is talking about OpenSim API, it is referring to C++ library used to simulate models. When it is talking about the OpenSim GUI is the graphical user interface tool which is used in Windows to simulate models.

q set of positions.

s set of auxiliary variables

SimBody API is a high-performance, industrial-grade open source C++ library providing sophisticated treatment of articulated multibody systems with special attention to the needs of biomedical simulation.

SolidWorks is a solid modeling computer-aided design (CAD) and computer-aided engineering (CAE) computer program that runs on Microsoft Windows. SolidWorks is published by Dassault Systèmes [22].

u set of velocities.

1 Introduction

Nowadays there are approximately over a billion of people [1] who suffer from some form of disability. The impairments can actually be of many types, ranging from cognitive, developmental, intellectual, physical or sensory. Many of them directly affecting the motor system producing as a result some type of mobility impairment. Hence mobility impairments include people across numerous types of physical disabilities which can classified into upper or lower limb and can be produced due to trauma, orthopedic, muscular dystrophy, problems in central nervous system or diseases. Consequently, it is estimated that there are more than 60 thousand people in need of a wheelchair [2] in order to overcome their disability. Thus, there is an obvious necessity to implement new technologies to make the lives of those who suffer from such impairments much easier.

Complete mobility recovery of a limb in result from an impairment, is indeed a difficult goal to achieve, in some cases even impossible. Nevertheless, current technology allows to make rehabilitation easier. One example of that is the use of robotics. This science allows to implement exercises of rehabilitation that a doctor can not do with the same accuracy as a robot can. This is reflected with repetitive rehabilitation movements, it is one of the best examples because a doctor never will be able to make exactly the same movement.

1.1 Technological Aids

Exist a lot of types of robots and robotic orthosis are able to recover a partially lost limb mobility. Moreover, even though orthosis can also do the same actions in those cases where the mobility is completely lost, the subject of this master thesis will be focused solely in the rehabilitation case, therefore it will not cover with the same detail other branches such as assistive or prosthesis robots.

Figure 1.1: Food orthosis.

Exoskeletons represent an example of robots used for rehabilitation purposes. Thus, this project will be particularly focused on the H1 Exoskeleton model. The main reason because of this, it is due to the fact that

the Polytechnic University of Catalonia actually owns one of these. The figure 1.2 shows the exoskeleton model. Detailed information regarding this model will be covered in the concepts section.

Figure 1.2: The H1 Exoskeleton.

1.2 Motivation

As mentioned before, many people suffer from some type of physical impairment, whether it is permanent or partial, there are some cases wehre there exists the possibility of recovery. In such case, the treatment in order to recover the lost limb requires time and a specialist that helps with it. Although in the future, a possible alternative could be based on new technologies, right now implementing hardware such as exoskeletons still has some noteworthy disadvantages: it is very expensive, not only the device itself —hardware and software, but also the time that is needed to run necessary tests and find possible exercises which adapt the best as possible to the pathology that a person suffers. Moreover, working directly with a exoskeleton and to make proves with a patient can put him at risk of suffer some kind of injure.

In this way, there is a need to find alternative options in order to save both time and money when finding the best solution for each patient's pathology. For this reason, software simulators such as Matlab, Ros, RoboDK, SimBody, OpenSim, and others, play an important role as these can model hence simulate the physical systems which can lead to better design orthosis designs, ultimately minimizing costs by saving time and money.

The project will be centered around two main software solutions, SimBody API and OpenSim. The first one is an open source C++ library provides sophisticated treatment of articulated physic systems as mechanism, skeletons, etc. Likewise, OpenSim is also a comprehensive C++ library build from SimBody, that allows, among many others things, to develop models of musculoskeletal structures and create dynamic simulations of movement as well. Thus, OpenSim is often described as a software that is able to model humans, animals, and robots.

Figure 1.3: Example of musculoskeletal model created with OpenSim.

OpenSim is a good software tool to implement because of his applications in simulating, and studying musculoskeletal body systems. Moreover, it is particularly interesting tool as the software has been developed by National Center for Simulation in Rehabilitation Research (NCSRR) where many people of different science branches converge working altogether in its development[5]. This means that OpenSim includes a lot of types of modeled real musculoskeletal bodies. Unfortunately, the main problem of OpenSim lies in the fact that the number of robotic orthosis implementations[8] —such as exoskeletons— jointly with musculoskeletal bodies is much lower. Thus, to develop an OpenSim model in which interaction forces can be studied

between a robot orthosis and a musculoskeletal body represent a great option to contribute in the project as well as a great opportunity.

The fact that so many people need rehabilitation due to partial impairment, implies that better methods have to be found in order to obtain efficient treatments. It seems that robotic orthosis will be an important option to contemplate in the future, as nowadays it is not efficient enough yet.

2 Objectives

Taking into account the previous described importance to continue developing solutions that minimize the costs associated with impairments recovery through robotics, the main objective of this project is to demonstrate the feasibility to create a model of an exoskeleton in OpenSim to be couple to a musculoskeletal OpenSim model. Because of the high level of complexity associated with the proposed goal, the design will not consist of a complete body robotic orthosis, but a lower limb exoskeleton that will be coupled to the right leg of an existing body. Prior carrying out the main objective, some sub-objectives must be actually completed as well. —Those are.

1. **Learn OpenSim and Simbody API:** Due to both software are a very extended C++ library, a previous study of both of them will be done, with the sole purpose to find an answer at the following questions:
   - How to create a OpenSim model?
   - How to create a controller for this OpenSim Model?
   - Validate the Dynamics of the OpenSim model.
2. **Design a Exoskeleton:** Before implementing the exoskeleton coupled to a musculoskeletal OpenSim model, the exoskeleton needs to be designed. —That includes.
   - Implement a visualization of it. In order to make a visual simulation, it will be needed a visual representation of the exoskeleton.
   - Obtain the dynamics of the model.
   - Design a controller. Taking into account the dynamics, design a controller that will be able to handle the positions of each joint.
3. **Couple the Exoskeleton to a Musculoskeletal OpenSim Model:** being the most important objective, and the main reason of this project.
   - How to couple the Exoskeleton to a Musculoskeletal OpenSim Model? Find out a way to link them with forces.
   - Extract the feedback variables including interaction forces: the H1 Exoskeleton has his own sensors in order to measure the position, velocity, and torque of each joint. Moreover, it has sensors that can measure the forces between a body and it. With the aim of emulating the H1 Exoskeleton, these variables have to be obtained in OpenSim.
4. **Document it for future work:** With the aim of facilitating future projects, the main idea of this goal is to provide a good documentation regarding how to implement using OpenSim.

3 Concepts

3.1 Rehabilitation

What exactly means rehabilitation? The World Health Organization (WHO) defines it as a set of measures that assist individuals who experience, or are likely to experience, disability to achieve and maintain optimal functioning in interaction with their environments. Nowadays, physiotherapists are those specialized in restoring optimal function to people with injuries to the muscles, bones, ligaments or nervous system. Thus, robot orthosis are a support to help and make the doctor's work easier in order to implement exercises of rehabilitation that can accelerate the process of recuperation.

3.1.1 Rehabilitation robotics

Focusing on rehabilitation robotics, it is a field of research dedicated to understanding and augmenting rehabilitation through the application of robotic devices. The robotic systems in rehabilitation can be classified as follows:

- Upper Limbs
  - Hands Support
  - Exoskeleton
- Lower Limbs
  - Treadmill
  - Exoskeleton

These robots are designed with applications of techniques that determine the adaptability level of the patient. There are many types of exercises used for such purpose:

- **Active assisted exercises:** the patient moves his hand in a predetermined pathway without any force pushing against it.
- **Active constrained exercises:** it is the movement of the patient's arm with an opposite force. If the arm's movement exceeds the pathway limits, the robot actuates.
- **Active resistivity exercises:** the patient arm's movement finds an opposite force exercised by the robot.
- **Passive exercises:** also known as passive range of motion; this means how far you can move your joints in different directions.
- **Adaptive exercises:** the robot has to adapt to the exercises because it has not ever done before and it has to adapt to the new unknown pathway.

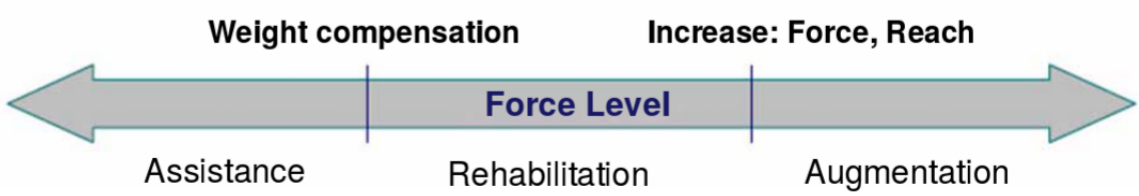


Figure 3.1: Robot application fields in function of weight compensation and force applied.

3.2 Exoskeleton Orthosis

Exoskeletons are a clear example of robotic orthoses contribute as a recovery assistance. These ones can be classified as following:

- Full body

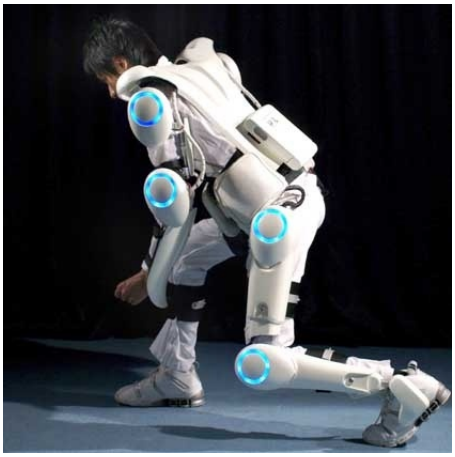


Figure 3.2: Full body Hall Exoskeleton.

- Upper extremeties

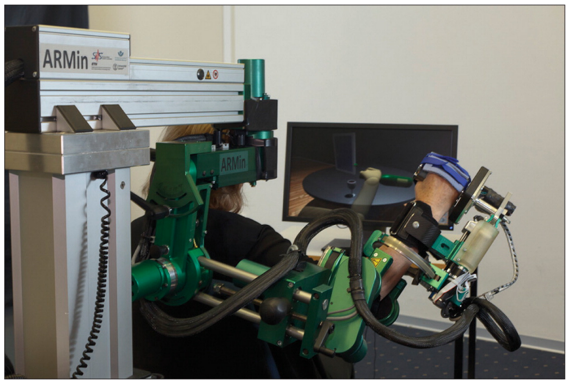


Figure 3.3: ARMin exoskeleton.

- Lower extremeties

Figure 3.4: H1 Exoskeleton.

3.2.1 H1 Exoskeleton

This subsection it is going to explain the main characteristics of the H1 Exoskeleton as his mechanical structure, actuators, electronics and software which is designed by Magdo Bortole. Detailed information can be found in *Design and Control of a Robotic Exoskeleton for Gait Rehabilitation* [3].

Mechanical structure

It is a lightweight design about 9 kg. In order to achieve this weight the H1 Exoskeleton's mechanical structure is made of aluminum and stainless steel. With six DoF, the mechanical structure is designed to allow active and passive movements in the sagittal plane. Three of his joints are powered and these ones are: hip, knee and ankle joints. In order to avoid injure users, the exoskeleton's joint ranges are mechanically limited. The maximum values are shown in the following table:

Table 3.1: Maximum joint range angles.

	Hip	**Knee**	**Ankle**
Flexion	100º	100º	20º
Extension	20º	5º	15º

The structure of the exoskeleton was designed that the thigh and the shank links can be adjusted by a mechanism of two telescopic bars which are fixed in different positions by screws. The same can be done with the foot length.

Actuators

The actuators that are used in this exoskeleton are DC brushless because compared with other type of motors offer a reduced noise, reduced volume, and high efficiency. Use this type of motors allows to place coaxially with the joints, not taking up much volume compared with other motors. One disadvantage that can be found on DC brushless motors is the high speed and low torque that they have. Taking into account that exoskeletons need slow speed and a good torque, harmonic drive or also called strain wave gears are used, too. These types of gears allows to reduce speed and increase torque without taking much volume.

Six motors are used, three for each leg. Each one has a 90 W of power and as it has been said previously, they are placed coaxially with each joint.

Power System

Each joint motor is driven by AZBH12A8. A servo drive that can generate Pulse Width Modulation (PWM). This drive is protected against over-voltage, over-current, over-heating, invalid communications and short-circuits. His dimensions are 63.5 x 50.8 x 16.8 mm, and 86 grams of weight.

Moreover, to give power to exoskeleton can be used a battery with 22.5 VDC or 230 VAC.

Sensors

Two types of sensors can be found in H1 Exoskeleton. These ones are kinematic and kinetic. The first ones are used for measuring angular position, velocity and acceleration. the second ones are in charge of measure force of interaction between user's limb and exoskeleton.

A potentiometer is placed in each joint. they are used as angular position sensors. The 10 kΩ has a tight linearity and long rotational life. This potentiometer contains a toothed pulley that jointly with a toothed belt that transmits motor motion can measure the joint position.

H1 Exoskeleton is equipped with strain gauges attached at each exoskeleton link. These ones are used as force sensors, basically measures the torque between the subject's limb and the exoskeleton. The actuators torque can be measured, too. This is due to servo drive can measure the current that motors are using.

The exoskeleton is provided with an Inertial Measurement Unit (IMU) that can take information about subject's inclination.

Communications

All sensors are connected to small electronic boards located at each joint. From each joint to the embedded computer that controls the exoskeleton are communicated with a CAN bus. In this way less wires are needed to communicate all the elements of the H1 exoskeleton. Thus, with each of this six CAN buses the information of all I/O are sent.

The embedded computer take all of this data and can send it via Bluetooth or CAN bus.

Software

The control algorithm designed for exoskeleton is implemented using Simulink. This one is a graphical interface tool that can generate the source code for the device based on the graphical model. Taking into account that standard PCs do not have input/output hardware for real time communications and signal acquisitions a stack with a PC104 is used where Matlab and Simulink models can be executed in real time.

3.3 Simbody

Digging deep into Simbody, we can find a set of application programs and Software Development Kit (SDK) embraced all of them with the name Simbios Biosimulation Toolkit (SimTK). The SDK includes a family of application programming interfaces (APIs) in which physics simulation can be done. Some examples of what can be done with them are: vector and matrix arithmetic, linear algebra, numerical integration, optimization, and so on.

The software that contains SimTK are:

- **Simbody:** it is an API for performing internal coordinate simulations of multibody systems.
- **OpenMM:** it is an API for GPU-accelerated computation of molecular force fields.
- **MolModel:** it is an API that uses Simbody to build internal coordinate molecule models that can use OpenMM-accelerated force fields.
- **OpenSim API:** it uses Simbody to build internal coordinate models of biomechanical systems.

3.3.1 Multibody System

A multibody system can be defined as a model of a physical system composed of mass-carrying objects. These mass-carrying objects can be interpreted as subsystems of the multibody system, each of them can be rigid or nearly rigid but they can move significantly relative to each other. Some examples are:

- **Human Skeleton:** his subsystems bodies are the rigid bones.

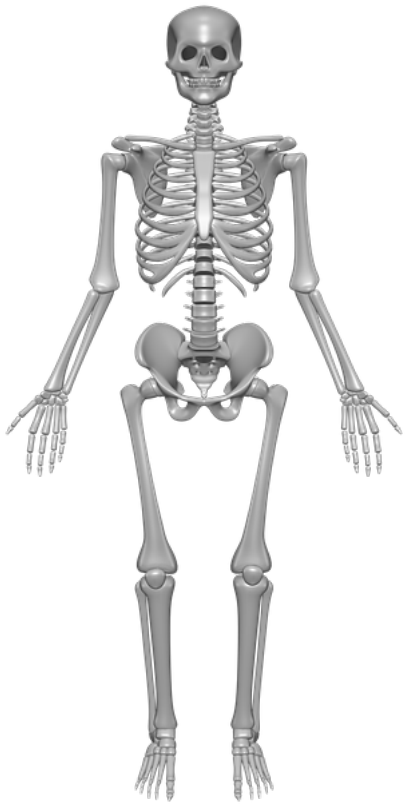


Figure 3.5: Example of skeleton.

- **Protein:** a collection of a rigid structure of atoms that move relative to each other.

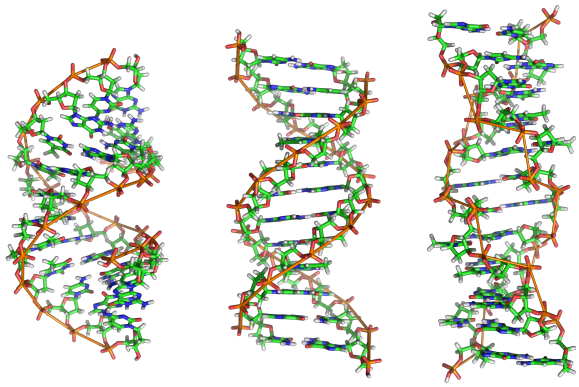


Figure 3.6: Example of DNA molecule

- **Automobile engine:** gears, pistons are examples of rigid parts that move relative to each other.

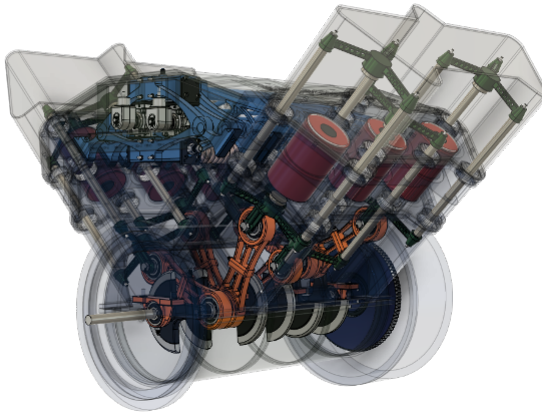


Figure 3.7: Example of engine with all the mechanics that implies.

Simbody has a particular way to describe systems. These systems or models are called internal coordinate multibody systems. The easiest way to describe all of the bodies that a multibody system is composed of, it is specifying six degrees of freedom for each one. The problem appears when all of the bodies have to join because a lot of constraints have to be defined. Thus, rather than do it as it has been explained previously, Simbody specifies the ways in which multibody systems bodies can actually move: for example, in the case of a skeleton, the angle which knee is bent. In other words, Simbody allows you to describe a multibody system in whatever way is most natural. Thus, it is a software that puts things easy because take care of all hard parts: calculating internal forces, transforming between internal and Cartesian coordinates, imposing constraints and so on.

3.3.2 Mathematics inside Simbody

Multibody system has to be seen as a system of equations that describes the behavior of a physic system. As a result, the state of the system at any moment in time is described by a vector of state variables. This is state vector is represented by $\mathbf{y}$ and the main objective of simulating is to integrate the following equation:

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(t, \mathbf{y}). \tag{3.1}$$

Where:
$\mathbf{f}$*: reflects the forces acting on bodies and laws of physics.*

The equation 3.1 represents the equation of motion, integrating it produce $\mathbf{y}(t)$ that represents a trajectory.

Digging into a vector of state variables and taking an human skeleton as example, it can be subdivided into:

- Generalized coordinates: it is represented with $\mathbf{q}$ and taking as reference a skeleton would represent the set of angles for all the joints, and the orientation and position of the torso.

- Generalized speeds: it is represented with $\mathbf{u}$ and it would correspond to angular and linear velocities.
- Auxiliary variables: it is represented with $\mathbf{z}$. Doing a simulation of a skeleton doing some sort of exercise, a possible example of auxiliary variable would be the total energy done in this exercise. Thus, auxiliary variables do not give information about the configuration of the multibody system.

$$\mathbf{y} = [\mathbf{q}, \mathbf{u}, \mathbf{z}]. \tag{3.2}$$

As it has been said previously, the use of generalized coordinates avoids the need for most constraints, but in some cases additional ones are needed. These constraints has to accomplish the following equation:

$$\mathbf{c}(t, \mathbf{q}, \mathbf{u}) = 0. \tag{3.3}$$

Simbody also contains event trigger functions. They are useful for reading multibody's data or turn on/off constraints. An event is said to occur when a trigger function crosses through 0:

$$\mathbf{e}(t, \mathbf{y}) = 0. \tag{3.4}$$

Thus, when an event occurs, the corresponding event handler is invoked, which can modify the state in arbitrary, discontinuous ways.

In order to track if constraints are turned on or not, a new type of variables called discrete variables $\mathbf{d}$ will be needed. Something important to highlight here is that these variables are not modified by equation 3.1. Only can be modified by event handlers. Thus, the equations (3.1, 3.3, 3.4) showed previously may all depend on discrete variables. So including these variables to these equations, we obtain:

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(\mathbf{d}; t, \mathbf{y}), \tag{3.5}$$

$$\mathbf{c}(\mathbf{d}; t, \mathbf{q}, \mathbf{u}) = 0, \tag{3.6}$$

$$\mathbf{e}(\mathbf{d}; t, \mathbf{y}) = 0. \tag{3.7}$$

As it will be noticed, there is a semicolon rather than a coma after $\mathbf{d}$. This is a reminder that discrete variables are held constant during continuous intervals when t and $\mathbf{y}$ are changing.

3.3.3 System and States

In Simbody, we can distinguish between two things: those that remain constants and immutable during time simulation and those ones that change during the course of a simulation. The first ones are represented by the system and the second ones by the state.

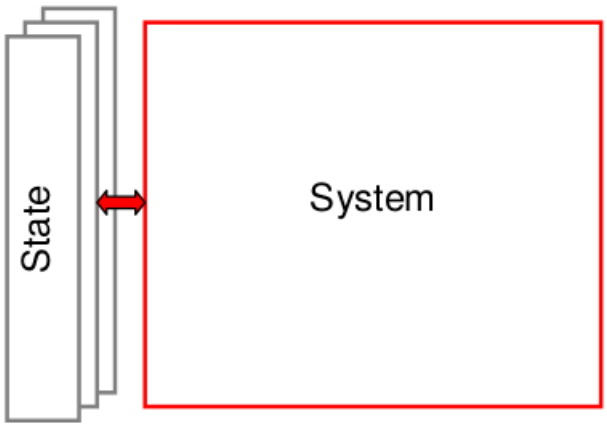


Figure 3.8: This figure represents System that once is build is immutable and everything that changes is stored in separate state objects.

In other words, a System object contains the bodies with all physical properties that defines them such as mass, inertia properties, dimensions, and so on. Moreover, System provides all the logic needed for simulation, instead a State object is purely a place for storing data.

- **System provides the following logics for simulation:**
  - Defines what information will be stored in a State.
  - Provides routines to calculate force function, vector of constraints, and vectors of event trigger functions.
  - Provides routines to handle events when they occur.
- **State stores:**
  - Time t.
  - Continuous state variables $\mathbf{y}$.
  - Discrete state variables $\mathbf{d}$.

3.3.4 System and Subsystems

Previously, it has been said that a System is in charge of giving all the basic logic to do the simulations, but actually who is in charge of it are the Subsystems not by the System itself. Thus, what Subsystems basically are, is those objects that compose a system. To understand in a better way of what we refer with Subsystems perform logic, it is going to illustrate it with an example: the total force calculated by the System is the sum of the forces calculated by all of its Subsystems. Check the figure 3.9:

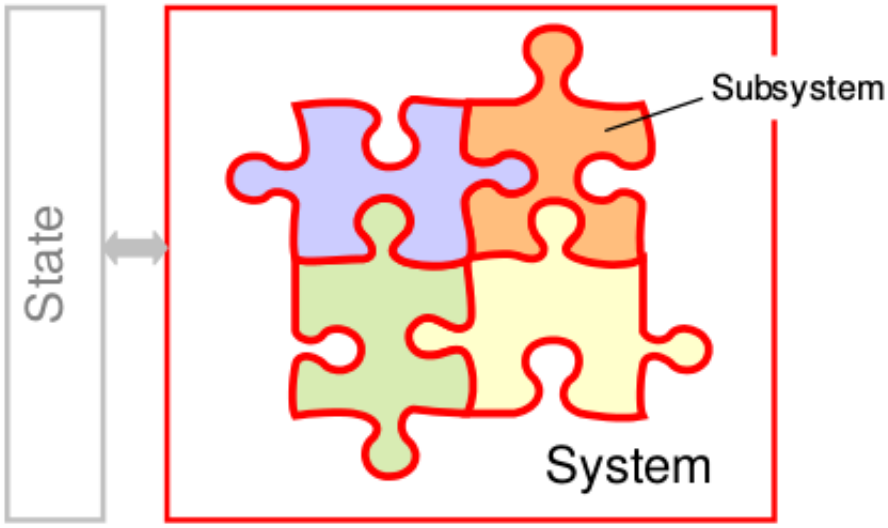


Figure 3.9: This image shows a representation of a System and his Subsystems.

Having Subsystems inside a System allows you to create it in a modular way since Subsystems can interact with each other. For example, Subsystem might define a subset of bodies, all the forces, constraints, and events related to them; another one can define all the state variables; a different Subsystem can define the forces acting on them, and so on.

3.3.5 The Realization Cache

In spite of the fact that state variables t, $\mathbf{y}$, and $\mathbf{d}$ represent a complete description of the system's state at a given time, there are other variables that might be interesting to know. These variables are:

- Position.
- Forces acting on each body.
- Accelerations.
- Values of event trigger functions.

The previous variables can be computed using state variables. The main problem here is they need some computational time to acquire them. For this reason, a State object provides space for storing these derived values. This space is called realization cache, and the process of calculating the values stored in it is known as realizing the state. Each of these variables cannot be computed at the same time, this means there is a sequence to compute each of them. Thus, the realization cache is therefore divided into a series of stages. When you want to get information from the cache, you must first make sure the state has been realized up to the stage that the information belongs to. Check figure 3.10.

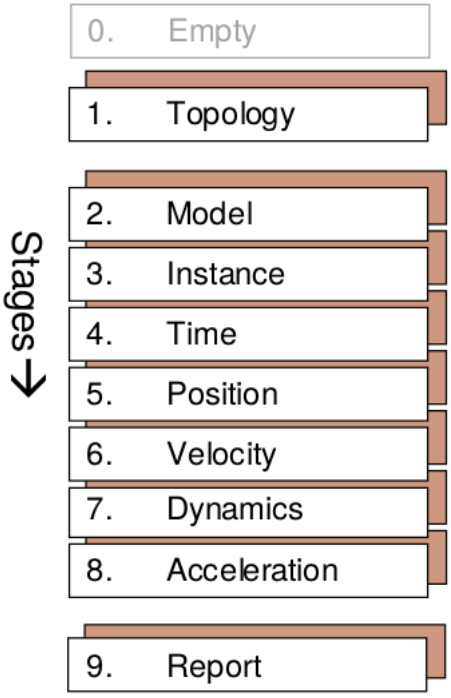


Figure 3.10: Organization of the different stages. The order is taking into account the stage that has to be computed to acquire the following stage.

The first four stages are involved in the initial construction and initialization of the system. Thus, these first stages are not important during time simulation, but yes, when you are interested in writing extensions to Simbody. In the following lines we are going to dig into more detail explaining every stage:

- **Empty:** Before a new constructed State object has been realized, it belongs to an empty stage that contains no information at all, and is not specific to any particular System.
- **Topology:** When a State gets realized to this stage, it is set to become a particular System's State. This means, allocating space in the cache for the System's data that needs to be stored.
- **Model:** It defines how many state variables become fixed. For example, Simbody allows to choose to use between Euler or quaternion angles. Depends on the choice three or four generalized coordinates will be created, respectively.

- **Instance:** At this stage, it is known which forces, constraints, and events are enabled.
- **Time:** No information has been calculated yet. Only information about state variables t, $\mathbf{y}$ and $\mathbf{d}$ are available.
- **Position:** At this stage, the position of all the bodies in Cartesian coordinates are known.
- **Velocity:** At this stage, the velocities of all bodies in Cartesian coordinates are known, along with the amount by which the constraints are violated.
- **Dynamics:** At this stage, the force acting on each body is known, along with the total kinetic and potential energy of the system.
- **Acceleration:** At this stage, the time derivatives of all continuous state variables are known, along with the values of all event trigger functions.
- **Report:** It is a stage that it is not normally realized, but it is available in case a System can calculate values that are not required for time integration, but might be needed by an event handler or for later analysis.

3.4 OpenSim

As it has been said previously, OpenSim is a free open source software package that allows to build, exchange, and analyse computer models of musculoskeletal systems and dynamic simulations of movement. As Simbody package, OpenSim was created by Stanford University and the first version of it was introduced at the American Society of Biomechanics Conference in 2007.

The latest version of OpenSim offers a Graphical User Interface (GUI) written in Java but it is only available by MAC and Windows. In order to create models an place the different bodies, it makes the job easier than programming it.

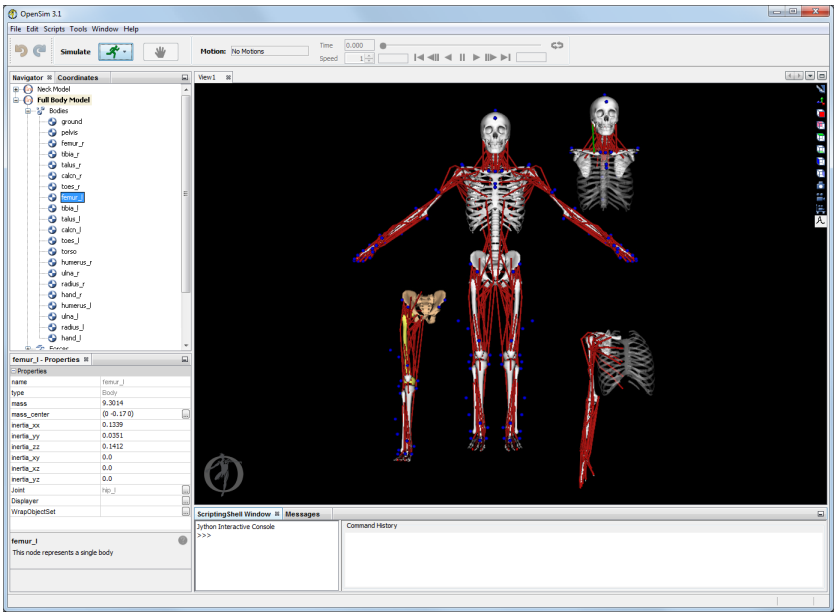


Figure 3.11: Screen record that shows OpenSim's GUI with different musculoskeletal models.

As it has been said previously, OpenSim is built on the computational and simulation core provided by SimTK. This core includes low-level, math and matrix algebra libraries, such as LAPACK, as well as Simbody, the infrastructure that allows to define a dynamic system and his states, and solve them. The following figure shows the three interface layers of OpenSim built on SimTK:

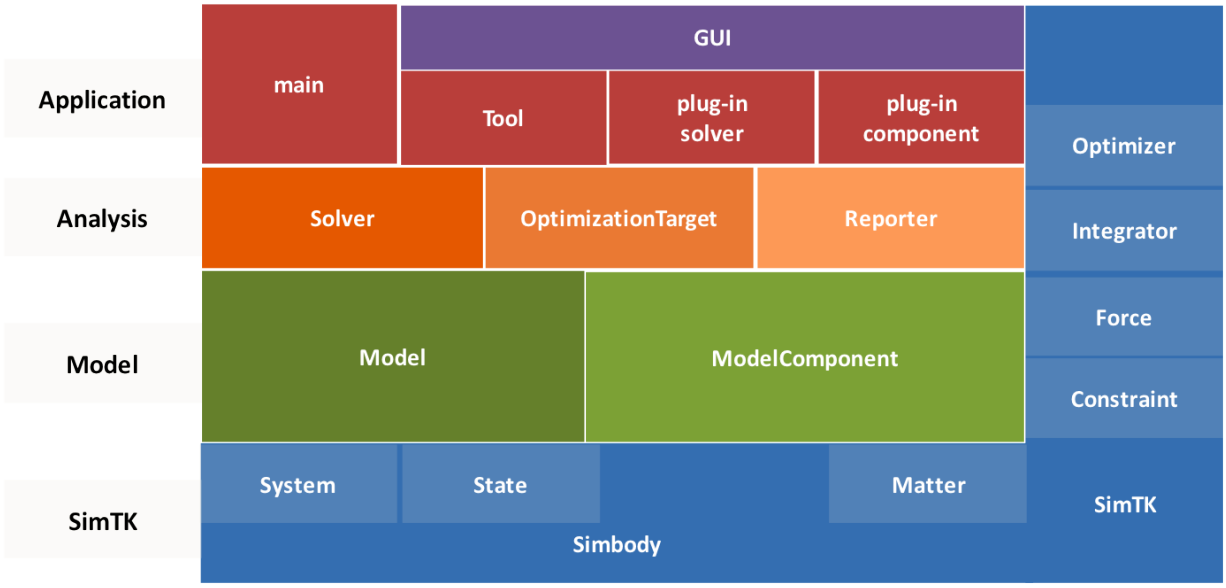


Figure 3.12: The three interface layers of OpenSim built on SimTK.

To understand better how OpenSim works, it is going to explain in more details the main parts which it is

composed of:

- **Manager:** this class is in charge to manage the execution of a simulation.
- **Optimizer:** the target of the optimizer class is to find a local minimum to an objective function.
- **Analysis:** this class allows to the user creates some plugins in order to make force analysis.
- **Dynamics Engine:** it is a wrapper class to use the SimTK Simbody dynamics engine as the underlying engine for OpenSim.
- **Model:** One of the main classes because it is in charge of creating a model or to call an existing one. Thus, it specifies the interface to a musculoskeletal model and can read this in from an XML file and modify it via OpenSim's API.
- **ModelComponent:** This class is used for adding computational components to the underlying SimTK::System or called MultibodySystem, too. In other words, it specifies the interface that components must satisfy in order to be part of the system and provides a series of helper methods for adding variables such as states, discrete, cache among others.

Some features that OpenSim lets to implement are:

- Importing Experimental Data.
- Scaling.
- Inverse Kinematics.
- Inverse Dynamics.
- Static Optimization.

3.4.1 Importing Experimental Data

OpenSim allows to analyse experimental data that has been collected from:

- Marker trajectories or joint angles from motion capture.
- Force data, typically could be ground reaction forces and moments and/or centers of pressure.
- Electromyography, data taken from brain.

3.4.2 Scaling

There are lots of generic models that have been created for using with OpenSim. Sometimes, this models can be useful to use with your experiments. For this reason, OpenSim lets scale the model to match the experimental data collected for your subject. The main reason to do that is to modify the model's

anthropometry to match with the subject's one. It is an important step in order to solve inverse kinematics and inverse dynamics problems because these solutions are sensitive to the accuracy of the scaling step. The things that are adjusted when an OpenSim model is scaled are the mass, the inertia tensors and the dimensions of the body segments.

3.4.3 Inverse Problem

Using experimental measured data from motion and forces of a subject to generate the kinematics and kinetics of a musculoskeletal model, Opensim enables to solve the inverse Dynamics problem.

Inverse Dynamics

The inverse Dynamics tool is in charge of determining the generalized forces that cause a particular motion, and its results can be used to infer how muscles are actuated to generate that motion.

Inverse Kinematics

The Inverse Kinematics tool computes generalized coordinate values which position the model in a pose that best matches for a particular experimental data recorded of a subject.

3.4.4 Static Optimization

Static Optimization resolves the net joint moments into individual forces at each instant in time. In other words, using a minimizing criteria finds the optimal force that should be applied in this time instance to a model's body.

3.5 Hardware used for simulation

The hardware that has been used for performing the simulations, it is a Sony VAIO laptop with the following software and hardware specifications:

- Operating System: Ubuntu 16.04 LTS
- Memory: 5.7 GiB
- Processor: Intel® Core™ i5 CPU M 480 @2.67GHz x 4
- Graphics: Gallium 0.4 on AMD REDWOOD (DRM 2.49.0 / 4.10.0-33-generic, LLVM 4.0.0)
- OS type: 64-bit
- Disk: 240.4 GB

4 Working With OpenSim

In this chapter will be designed a first basic OpenSim model based on a double pendulum example which has already been created in Simbody but it will be adapted for being used with OpenSim. Moreover, each joint pendulum will contain an actuator which will be controlled by a controller that has been designed, too. Finally, some test will be done in order to check if the controller works, and another test to compare the dynamics that computes OpenSim with the theoretical dynamics computed.

4.1 Pendulum design in OpenSim

4.1.1 Create the pendulum

First of all, to create a model or calling an existing one —in order to modify it, the following class has to be implemented:

`OpenSim::Model`

The class Model allows to create an object that will contain the model which has been designed. This class, among other things, allows to set the model's and author's name, and so on. Moreover, provide a set of tools that allows adding forces, bodies, controllers, and constraints that gives shape to the design. In summary, it is the main class of OpenSim that makes our models work.

```
1  //////////////////////
2  // DEFINE VARIABLES //
3  //////////////////////
4  SimTK::Vec3 gravity(0,-9.8065, 0); // Model'ls gravity value
5  // Time simulation
6  double initTime(0), endTime(10); // In seconds
7
8  ///////////////////
9  // CREATE MODEL //
10 ///////////////////
11 Model osimPendulum;
12 osimPendulum.setName("ControlDoublePendulum"); // Set the name of the model
13 osimPendulum.setAuthors("Dídac Coll");         // Set the name of the author
14 osimPendulum.setUseVisualizer(true);
15 osimPendulum.setGravity(gravity);
```

Listing 4.1: This code creates an object Model and sets, among other things, the model's, and author's name, and add the gravity force.

Once done a little introduce to the Model class, it is time to digging into the code which has created the double pendulum. The first thing that has to be done after being created an object of type model, it is to create pendulum bodies. Using the class `OpenSim::Body` will create these bodies. An object Body is used for representing a reference frame which describes mass properties and geometries. Thus, using the Body class, it will be created two cylinders and a cube that will compose the pendulum system. See the following table to see a brief summarize of the pendulum's physic properties:

Table 4.1: Pendulum's Bodies Properties

Links	**Cylinders**			**Cub**	
Properties	Mass	Height	Diameter	Mass	Side Length
Value	0.1 kg	0.5 m	0.06 m	20 kg	0.2 m

```
1  ////////////////////
2  // CREATE BODIES //
3  ////////////////////
4  // Define Mass properties, dimensions and inertia
5  double cylinderMass = 0.5, cylinderLength = 2.5, cylinderDiameter = 0.30;
6  SimTK::Vec3 cylinderDimensions(cylinderDiameter, cylinderLength, cylinderDiameter);
7  SimTK::Vec3 cylinderMassCenter(0, cylinderLength/2, 0);
8  SimTK::Inertia cylinderInertia = SimTK::Inertia::cylinderAlongY(cylinderDiameter/2.0,
9  cylinderLength/2.0);
10 OpenSim::Body *firstCylinder = new OpenSim::Body("firstCylinder", cylinderMass,
11 cylinderMassCenter, cylinderMass*cylinderInertia);
12 // Set a graphical representation of the cylinder
13 firstCylinder->addDisplayGeometry("cylinder.vtp");
14 // Scale the grahical cylinder(1 m tall, 1 m diameter) to match with body's dimensions
15 GeometrySet& geometrySet = firstCylinder->updDisplayer()->updGeometrySet();
16 DisplayGeometry& newDimCylinder = geometrySet[0];
17 newDimCylinder.setScaleFactors(cylinderDimensions);
18 newDimCylinder.setTransform(Transform(SimTK::Vec3(0.0, cylinderLength/2.0, 0.0)));
19 // Add Sphere in Joint place
20 firstCylinder->addDisplayGeometry("sphere.vtp");
21 // Scale sphere
22 geometrySet[1].setScaleFactors(SimTK::Vec3(0.5));
```

Listing 4.2: This code creates the first cylinder of the pendulum.

In the code 4.2 has to be distinguished two main parts: the first one defines the pendulum physical properties, and the last one defines how it will look like in the simulation. Taking a look at the previous code, OpenSim uses some classes of Symbody as

`SimTK::Vec3`

and

`SimTK::Inertia::cylinderAlongY`

in order to define model's mass properties and dimensions. The first class defines a vector object of three dimensions (x, y, z), and the second one defines the inertia properties of a cylinder along the y axis. By default, Body class will not create a display shape to show in the visual simulation when this will run. For this reason, it has to be added what in OpenSim is called a display geometry with:

```
firstCylinder->addDisplayGeometry("cylinder.vtp")
```

The previous code adds a visualization of a cylinder. it has defined dimensions by default. Thus, it needs to be adapted to the Body's dimensions. Check the following figure to see what the listing 4.2 generates:

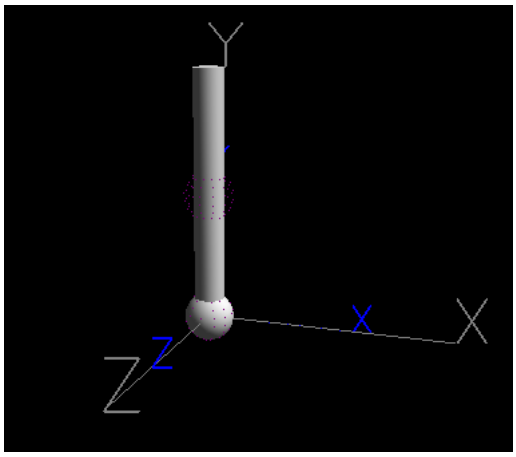


Figure 4.1: Geometry generated by the code of listing 4.2.

Once the first pendulum's cylinder is defined , it has used to copy it to obtain another cylinder with the same properties as the first one. At the end of the pendulum it has added a block.

All bodies and their visualization to display in the simulation are defined so far. Now, they has to be joined among them. To do that, it has to call the class `OpenSim::Joint`. This class generates a generic joints, it can be useful if we want to create your own personalized joints. In this case, subclasses derived from it are those that we are interested in. The table 4.2 shows the main subclasses of Joint class.

Table 4.2: Children classes of Joint class

Joint subclasses
BallJoint
CustomJoint
EllipsoidJoint
FreeJoint
GimbalJoint
PinJoint
PlanarJoint
SliderJoint
UniversalJoint
WeldJoint

Among the previous subclasses, it has been used to PinJoint to define the type of joint between ground with the first cylinder, and the first cylinder with the second one. At the end of the second cylinder, a WeldJoint has been used to link the second pendulum's cylinder with the cube. On the one hand, PinJoint generates a type of joint with one rotational DoF in the Z axis. On the other hand, WeldJoint does not generate any DoF, it can be considered the block is attached to the second cylinder.

One time joints are defined, the coordinates of them has to be configured. For example, setting the range of

PinJoints or naming them. Finally, the bodies are ready to be added to the Model. The lines 29, 30, and 31 of the listing 4.3 are in charge of adding all the bodies to it. If the figure is checked , it will be seen what the code described until here creates.

```
// Create 1 degree-of-freedom pin joints between ground, first cylinder, and second cylinder
// , weldjoint to attach block to second cylinder
SimTK::Vec3 orientationInGround(0), locationInGround(0),
locationInParent(0.0, cylinderLength, 0.0),
orientationInChild(0), locationInChild(0);

PinJoint *firstJoint = new PinJoint("firstJoint", ground, locationInGround,
orientationInGround, *firstCylinder,
locationInChild, orientationInChild);
PinJoint *secondJoint = new PinJoint("secondJoint", *firstCylinder, locationInParent,
orientationInChild, *secondCylinder,
locationInChild, orientationInChild);
WeldJoint *endPendulum = new WeldJoint("endPendulum", *secondCylinder,
locationInParent, orientationInChild,
*block, locationInChild, orientationInChild);

// Set range of joints coordinates
double range[2] = {-SimTK::Pi*2, SimTK::Pi*2};
// Get coordinate set of firstJoint
CoordinateSet& fJointCoordinates = firstJoint->upd_CoordinateSet();
fJointCoordinates[0].setName("q1");
fJointCoordinates[0].setRange(range);
// Get coordinate set of secondJoint
CoordinateSet& sJointCoordinates = secondJoint->upd_CoordinateSet();
sJointCoordinates[0].setName("q2");
sJointCoordinates[0].setRange(range);

// Add Bodies
osimPendulum.addBody(firstCylinder);
osimPendulum.addBody(secondCylinder);
osimPendulum.addBody(block);
```

Listing 4.3: This code creates joints between each body, set the coordinates of each joint, and finally add all of them to the system.

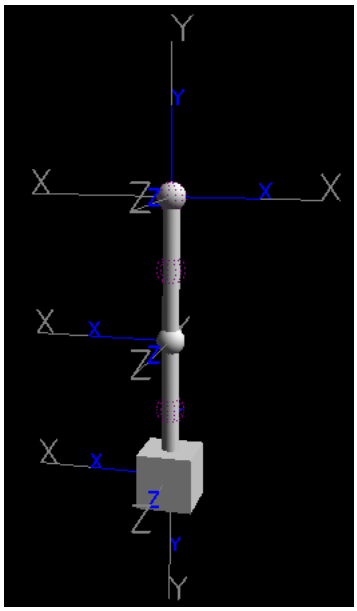
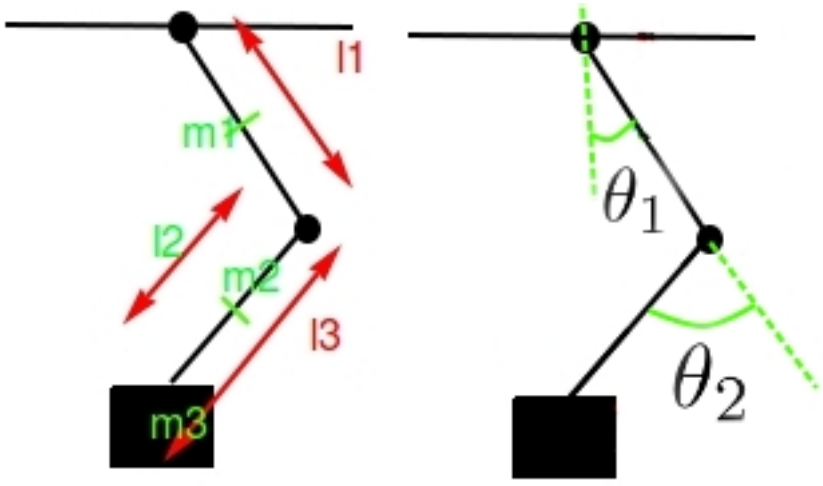


Figure 4.2: Double pendelum created with OpenSim API.

4.1.2 Double Pendulum Dynamics

Before getting into more detail with the controller, first, it has been studied theoretically the dynamics of the double pendulum. This study will be useful as a review of the forces that interact with it. Thus, the equations will give information about the way in which motion of the pendulum arises from torques applied by the actuators or from external forces applied to the double pendulum. In this case, it has been considered this pendulum as a manipulator or a serial-chain that will contain actuators which will be controlled.

Two main problems related to the dynamics of a manipulator is needed to be solved. The first one, it is controlling it. In other words, given a reference position or a trajectory, we want to find the vector of torques τ that will be needed to apply in each joint in order that pendulum reaches the position or do the trajectory. The second problem is to calculate how the mechanism will move under application of a set of joint torques.



(a) Masses and lengths of each pendulum part.

(b) Angles of each pendulum joint.

Figure 4.3: Double pendulum analysis.

In this case, it is going to use an ”energy-based” approach to obtain the dynamics equations based on Lagrangian dynamic formulation [9].

On the one hand, we have the kinetic energy which can be expressed as

$$T_i = \frac{1}{2} m_i v_i^2 + \frac{1}{2} I_i \omega_i^2, \tag{4.1}$$

Where:
m_i*: it is the mass of link ”i”.*
v_i*: it is the linear velocity of link ”i”.*
I_i*: it is the inertia of link ”i”.*

ω_i*: it is the angular velocity of link "i".*

where the first term is the kinetic energy due to the linear velocity of the link and the second term is the kinetic energy due to angular velocity. The total kinetic energy of the system is the sum of each link's kinetic energy which is expressed as

$$T = \sum_{i=1}^{n} T_i. \tag{4.2}$$

The linear velocity and the angular velocity are functions of the angular position and his derivative (Θ,$\dot{\Theta}$). As a result, the kinetic energy can be described as a function of joint position and velocity, $T(\Theta,\dot{\Theta})$ and now the equation can be expressed as

$$T(\Theta, \dot{\Theta}) = \frac{1}{2}\dot{\Theta}^T M(\Theta)\dot{\Theta}, \tag{4.3}$$

where $M(\Theta)$ is the n x n pendulum general mass matrix. The equation 4.3 is known as a quadratic form [11].

On the other hand, we have that a pendulum is described not only by his kinetic energy but also by his potential energy. This potential energy can be expressed as ith link's potential energy (4.4).

$$P_i = m_i g_0 d_i. \tag{4.4}$$

Where:
m_i*: it is the mass of link "i".*
g_0*: it is the universal constant gravity.*
d_i*: it is the distance between the center of the joint and the mass center of ith link.*

As the kinetic energy, the total potential energy can be expressed as a sum of the potential energy of each link (4.5).

$$P = \sum_{i=1}^{n} P_i, \tag{4.5}$$

Taking into account that d_i will depend on Θ, the potential energy can be described by a scalar formula as a function of joint position, $P(\Theta)$.

Once known the potential and the kinetic energy of the pendulum, the Lagrangian dynamic formulation plays his roles because provides a means of deriving the equations of motion from a scalar function called the Lagrangian [10],

$$\mathcal{L}(\Theta, \dot{\Theta}) = T(\Theta, \dot{\Theta}) - P(\Theta). \tag{4.6}$$

The equation of motion is

$$\frac{d}{dt}\frac{\partial \mathcal{L}}{\partial \dot{\Theta}} - \frac{\partial \mathcal{L}}{\partial \Theta} = \tau, \tag{4.7}$$

where τ is a $n\ x\ 1$ vector of actuator torques. Thus, the pendulum's motion equation is

$$\frac{d}{dt}\frac{\partial T(\Theta, \dot{\Theta})}{\partial \dot{\Theta}} - \frac{\partial T(\Theta, \dot{\Theta})}{\partial \Theta} + \frac{\partial P(\Theta)}{\partial \Theta} = \tau. \tag{4.8}$$

Now it is time to develop the equations of the pendulum which was shown in the figure 4.3. First of all, it is going to obtain the linear position equations —x_i, y_i— and their linear velocities —$\dot{x}_i$,$\dot{y}_i$— of each link's mass center

$$\begin{aligned}
x_1 &= d_1 s_1 & \dot{x}_1 &= \tfrac{l_1}{2} c_1 \dot{\theta}_1 \\
y_1 &= -d_1 c_1 & \dot{y}_1 &= \tfrac{l_1}{2} s_1 \dot{\theta}_1 \\
x_2 &= l_1 s_1 + d_2 s_{12} & \dot{x_2} &= (l_1 c_1 + \tfrac{l_2}{2} c_{12})\dot{\theta}_1 + \tfrac{l_2}{2} c_{12} \dot{\theta}_2 \\
y_2 &= -l_1 c_1 - d_2 c_{12} & \dot{y}_2 &= (l_1 s_1 + \tfrac{l_2}{2} s_{12})\dot{\theta}_1 + \tfrac{l_2}{2} s_{12} \dot{\theta}_2 \\
x_3 &= l_1 s_1 + l_3 s_{12} & \dot{x}_3 &= (l_1 c_1 + \tfrac{l_3}{2} c_{12})\dot{\theta}_1 + l_3 c_{12} \dot{\theta}_2 \\
y_3 &= -l_1 c_1 - l_3 c_{12} & \dot{y}_3 &= (l_1 s_1 + l_3 s_{12})\dot{\theta}_1 + l_3 s_{12} \dot{\theta}_2.
\end{aligned}$$

where d_i it is equal to $\frac{l_i}{2}$, and then by simplification, we have that s_i is $sin(\theta_i)$, s_{ij} is $sin(\theta_i + \theta_j)$, c_i is $cos(\theta_i)$, and c_{ij} is $cos(\theta_i + \theta_j)$.

Now, if we take equations (4.1) and (4.3), we obtain that pendulum's kinetic energy is

$$T = \frac{1}{2}(m_1 v_1^2 + I_{z1}\omega_1^2 \dot{\theta}_1^2 + m_2 v_2^2 + I_{z2}\omega_2^2 + m_3 v_3^2 + I_{z3}\omega_3^2). \tag{4.9}$$

Where:

$$v_1^2 = \dot{x}_1^2 + \dot{y}_1^2, \quad \omega_1^2 = \dot{\theta}_1^2,$$
$$v_2^2 = \dot{x}_2^2 + \dot{y}_2^2, \quad \omega_2^2 = \dot{\theta}_1^2 + \dot{\theta}_2^2,$$
$$v_3^2 = \dot{x}_3^2 + \dot{y}_3^2, \quad \omega_3^2 = \dot{\theta}_1^2 + \dot{\theta}_2^2$$

Applying Lagrange to the kinetic equation, we obtain the following

$$\frac{d}{dt}\left(\frac{\partial \mathcal{L}}{\partial \theta_1}\right) =$$

$$\left(I_{z1} + I_{z3} + \frac{m_1 l_1^2 + 4I_{z1}}{4} + m_2 l_1^2 + m_3 l_1^2 + \frac{m_2 l_2^2}{4} + m_3 l_3^2 + m_2 l_1 l_2 c_2 + 2m_3 l_1 l_3 c_2\right)\ddot{\theta}_1$$
$$+ \left(I_{z2} + I_{z3} + \frac{m_2 l_2^2}{4} + m_3 l_3^2 + \frac{m_2 l_1 l_2}{2} c_2 + m_3 l_1 l_3 c_2\right)\ddot{\theta}_2$$
$$+ \left(-m_2 l_1 l_2 s_2 + 2m_3 l_1 l_3\right)\dot{\theta}_1\dot{\theta}_2 - \left(\frac{m_2 l_1 l_2 s_2}{2} + m_3 l_1 l_3 s_2\right)\dot{\theta}_2^2$$

$$\frac{\partial \mathcal{L}}{\partial \theta_1} = 0$$

$$\frac{d}{dt}\left(\frac{\partial \mathcal{L}}{\partial \dot{\theta}_2}\right) = \left(I_{z2} + I_{z3} + \frac{m_2 l_2^2}{4} + m_3 l_3^2 + \frac{m_2 l_1 l_2 c_2}{2} + m_3 l_1 l_3 c_2\right)\ddot{\theta}_1$$
$$+ \left(I_{z2} + I_{z3} + \frac{m_2 l_2^2}{4} + m_3 l_3^2\right)\ddot{\theta}_2 - \left(m_2 l_1 l_2 s_2 + m_3 l_1 l_3 s_2\right)\dot{\theta}_1\dot{\theta}_2$$

$$\frac{\partial \mathcal{L}}{\partial \theta_2} = -\frac{m_2 l_1 l_2 s_2}{2}\dot{\theta}_1^2 - m_3 l_1 l_3 s_2 \dot{\theta}_2^2 - \left(\frac{m_2 l_1 l_2 s_2}{2} + m_3 l_1 l_3 s_2\right)\dot{\theta}_1\dot{\theta}_2$$

If we analyse the potential energy of each pendulum's mass, we obtain that

$$P_1 = -m_1 g_0 d_1 c_1$$

,

$$P_2 = -m_2 g_0 (l_1 c_1 + d_2 c_2)$$

,

$$P_3 = -m_3 g_0 (l_1 c_1 + l_3 c_2)$$

, where d_i is $\frac{l_i}{2}$, and c_i is $cos(\theta_i)$. Finally, applying the equation (4.5), we obtain that the total potential energy of the pendulum is

$$P = -m_1 g_0 \frac{l_1}{2} c_1 - m_2 g_0 \left(l_1 c_1 + \frac{l_2}{2} c_{12} \right) - m_3 g_0 (l_1 c_1 + l_3 c_{12}). \tag{4.10}$$

Applying Lagrange we obtain those parameters that are influenced by the gravity in the motion equation

$$\frac{\partial P}{\partial \theta_1} = g_0 \left(m_1 \frac{l_1}{2} s_1 + m_2 (l_1 s_1 + \frac{l_2}{2} s_{12}) + m_3 (l_1 s_1 + l_2 s_{12}) \right)$$

$$\frac{\partial P}{\partial \theta_2} = g_0 \left(m_2 \frac{l_2}{2} s_{12} + m_3 l_3 s_{12} \right)$$

.

Once it is solved the Lagrangian equations, we want to give the previous parameters with the dynamics equation form —that is

$$M(\Theta)\ddot{\Theta} + C(\Theta, \dot{\Theta})\dot{\Theta} + N(\Theta, \dot{\Theta}) = \tau. \tag{4.11}$$

Where:
$M(\Theta)$: it is $n\ x\ n$ matrix which is called as manipulator mass matrix.
$C(\Theta, \dot{\Theta})$: it is the $n\ x\ 1$ Coriolis matrix.
$N(\Theta, \dot{\Theta})$: it is the gravity term.

Finally, the pendulum motion equation is

$$\begin{pmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{pmatrix} \begin{pmatrix} \ddot{\theta}_1 \\ \ddot{\theta}_2 \end{pmatrix} + \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} \begin{pmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \end{pmatrix} + \begin{pmatrix} N_1 \\ N_2 \end{pmatrix} = \begin{pmatrix} \tau_1 \\ \tau_2 \end{pmatrix}, \tag{4.12}$$

where the components of $M(\theta)$ are equal to

$$M_{11} = I_{z1} + I_{z3} + \frac{m_1 l_1^2 + 4 I_{z1}}{4} + m_2 l_1^2 + m_3 l_1^2 + \frac{m_2 l_2^2}{4} + m_3 l_3^2 + m_2 l_1 l_2 c_2 + 2 m_3 l_1 l_3 c_2$$
$$M_{12} = I_{z2} + I_{z3} + \frac{m_2 l_2^2}{4} + m_3 l_3^2 + \frac{m_2 l_1 l_2}{2} c_2 + m_3 l_1 l_3 c_2$$
$$M_{21} = I_{z2} + I_{z3} + \frac{m_2 l_2^2}{4} + m_3 l_3^2 + \frac{m_2 l_1 l_2 c_2}{2} + m_3 l_1 l_3 c_2$$
$$M_{22} = I_{z2} + I_{z3} + \frac{m_2 l_2^2}{4} + m_3 l_3^2,$$

and the matrix Coriolis and centrifugal elements C_{11}, C_{12}, C_{21} and C_{22} are

$$C_{11} = (-m_2 l_1 l_2 s_2 + 2 m_3 l_1 l_3)\, \dot{\theta}_2$$
$$C_{12} = -\left(\tfrac{m_2 l_1 l_2 s_2}{2} + m_3 l_1 l_3 s_2 \right) \dot{\theta}_2$$
$$C_{21} = -\tfrac{m_2 l_1 l_2 s_2}{2} \dot{\theta}_1 - \left(\tfrac{3 m_2 l_1 l_2 s_2}{2} + 2 m_3 l_1 l_3 s_2 \right) \dot{\theta}_2$$

$$C_{22} = -m_3 l_1 l_3 s_2 \dot{\theta}_2$$

Finally, gravity components are equal to

$$N_1 = g_0 \left(m_1 \tfrac{l_1}{2} s_1 + m_2 (l_1 s_1 + \tfrac{l_2}{2} s_{12}) + m_3 (l_1 s_1 + l_2 s_{12}) \right)$$
$$N_2 = g_0 \left(m_2 \tfrac{l_2}{2} s_{12} + m_3 l_3 s_{12} \right),$$

4.1.3 Pendulum Controller

The previous study of the double pendulum dynamics has allowed to determine which forces interact with it, and how it is going to involve when some motion will be applied to it. Once these equations have been determined, it is time to discuss how to design a controller which will allow us to cause a desired motion or place it to a desired position.

Given the current position and velocity of the pendulum, and adding a sate feedback, a possible control law can be

$$\tau = M(\Theta) \left(\ddot{\Theta}_d - K_v \dot{e} - K_p e \right) + C(\Theta, \dot{\Theta}) \dot{\Theta} + N(\Theta, \dot{\Theta}). \tag{4.13}$$

Where:
$\ddot{\Theta}_d$: is the reference acceleration
K_v and K_p: are constant gain matrices.
e: it is the error produced by $\Theta - \Theta_d$

In this case, only the position will be controlled. Thus, the trajectory right now it is not important. In order to check if the previous control law is useful for the pendulum. It has been made a study with a control-Lyapunov function because it is wanted that pendulum remains stable by any given reference state. In brief, we want a Lyapunov function $P(x)$ that by any state x there exists a control $u(x,t)$ such that the system can be brought to the zero state to the desired state by applying this control law u [12][13]. First of all let us remember the Lyapunov theorem:

Let $\dot{x} = f(x)$, $f(0) = 0$, and $0 \in \Omega \subset \mathbf{R^n}$. Assume that $V : \Omega \to \mathbf{R}$ is a $\mathbf{C^1}$ function. if

(1) $V(0) = 0$
(2) $V(x) > 0$, for all $x \in \Omega$, $x \neq 0$
(3) $\frac{d}{dt} V(x) \leq 0$ along all trajectories of the system in Ω

then $x = 0$ is locally stable. Furthermore, if also

(4) $\frac{d}{dt} V(x) < 0$, for all $x \in \Omega$, $x \neq 0$

then $x = 0$ is locally asymptotically stable. Finally, if

(5) $V(x) \to \infty$ as $\|x\| \to \infty$

then $x = 0$ is globally asymptotically stable.

Thus, let us apply this theorem to the pendulum in order to obtain a stable control law $u(\theta, t)$. Thus, the pendulum is a non-linear system whose dynamics are the equation (4.11) obtained applying Lagrange.

Given a certain angle as reference (θ_d) and knowing the actual state (θ), we can compute the error — $e = \theta_d - \theta$. This error will allow the control system to compute how much torque will require each joint to reach the position given by the users. Not only the error between the position is used to compute the torque, but also the velocity error — $\dot{e} = \dot{\theta}_d - \dot{\theta}$. Thus, let us define function r as

$$r(\theta) = \dot{e} + \alpha e. \tag{4.14}$$

Then a Control-Lyapunov candidate is then

$$V(\theta) = \frac{1}{2} r^2, \tag{4.15}$$

because it is positive definite for all $\theta \neq 0$, and $\dot{\theta} = 0$. Thus, the first and the second rule of Lyapunov theorem are accomplished. Let us compute the derivative of V

$$\begin{aligned} \dot{V} &= r\dot{r} \\ &= (\dot{e} + \alpha e)(\ddot{e} + \alpha \dot{e}), \end{aligned} \tag{4.16}$$

The main objective is to get the time derivative to be

$$\begin{aligned} \dot{V} &= -kV \\ &= -k\frac{1}{2}(\dot{e} + \alpha e)^2, \end{aligned} \tag{4.17}$$

because V it is globally positive definite and this means $\dot{V}$ will be globally stable. if we develop the equations

$$\begin{aligned}(\ddot{e}+\alpha\dot{e})(\dot{e}+\alpha e) &= -\frac{1}{2}k\,(\dot{e}+\alpha e)^2\\ (\ddot{e}+\alpha\dot{e}) &= -\frac{k}{2}(\dot{e}+\alpha e)\\ (\ddot{\theta}_d-\ddot{\theta}+\alpha\dot{e}) &= -\frac{k}{2}(\dot{e}+\alpha e).\end{aligned}$$

Finally, we obtain the angular acceleration is equal to

$$\ddot{\theta} = \ddot{\theta}_d + (\frac{k}{2}+\alpha)\dot{e} + \alpha\frac{k}{2}e \tag{4.18}$$

In this case, only specific positions will be controlled. Thus, the expression can be simplified because the θ_d becomes constant and whose derivatives are equal to 0. For this reason our control law becomes:

$$u(\theta,t) = -(\frac{k}{2}+\alpha)\dot{\theta} + \alpha\frac{k}{2}(\theta_d-\theta). \tag{4.19}$$

If we take this acceleration for each joint we gonna obtain the control law vector

$$U(\Theta) = \begin{pmatrix} u(\theta_1,t)\\ \vdots \\ u(\theta_n,t)\end{pmatrix} \tag{4.20}$$

and replace the vector accelerations by the control law in the dynamic equation (4.11), the dynamics equation becomes as follows

$$\tau = M(\Theta)U(\Theta) + C(\Theta,\dot{\Theta})\dot{\Theta} + N(\Theta,\dot{\Theta}). \tag{4.21}$$

4.1.4 Controller Implementation in OpenSim

In order to control the double pendulum a new class, called ”PendulumController”, has been implemented, . This class derives from a parent class belonging to OpenSim called ”controller” which contains all the tools to design a basic controller with actuators.

Before giving a detailed information of this new class, our model needs actuators towards generating a force that give as a result a movement on our pendulum. OpenSim offers a list of actuators or even create our own ones. In this case, we are interested in apply torques in each pendulum's joint in the z coordinates. Thus, it has been used a class which is called ”TorqueActuator” that makes exactly what is wanted.

```
1 // Creaate Actuators
2 TorqueActuator* firstJointActuator = new TorqueActuator();
3 firstJointActuator->setName("firstJointActuator");
4 firstJointActuator->setBodyA(osimPendulum.getBodySet().get("firstCylinder"));
5 firstJointActuator->setBodyB(osimPendulum.getGroundBody());
6 firstJointActuator->setOptimalForce(20);
```

Listing 4.4: This code creates an actuator placed in the first joint.

The double pendulum needs two actuators. Moreover, they will be controlled by the class that has been mentioned previously. In the PositionController class has been created some methods to compute a PD control. Remembering the equation (4.19); the parameter that multiplies the current error will be called K_p and the scalar variable that multiplies the derivative part will be K_v,

$$K_p = \alpha\frac{k}{2},$$
$$K_v = \frac{k}{2} + \alpha,$$

with these two variables, which will be defined by the user; the angle reference; and the two methods (`controlFirstJoint()`,and `controlSecondJoint()`), which has been created; the desired acceleration will be computed. Take a look at the following code:

```
double controlFirstJoint(SimTK::State s,double ref,double Kp,double Kv,
                                                    double *getErr=nullptr) const
{
        //double k = Kp, alpha = Kv;
        double xt_0 = getAngleFromJoints("q1",s); // Get actual joint state x(t)
        static double xt_1 = 0; // x(t-1)
        double err = getError(ref, xt_0); // Compute the error e = x_r - x(t)
        if(getErr != nullptr) // Return the error in a pointer, In case of needed it
        *getErr = err;
        double dx = (xt_0 - xt_1)/0.033; // Compute the derivative dx = (x(t)-x(t-1))/timeSample
        xt_1 = xt_0;
        //double torque = alpha*k/2.0*err - (k/2 + alpha)*dx; // Control law with k and alpha
        double torque = Kp*err - Kv*dx; // Control law
        return torque;
}
```

Listing 4.5: This code creates an actuator placed in the first joint.

OpenSim offers methods to obtain the dynamics that affects each joint. Thanks to this, it saves a lot of time because there is no need to implement new methods that contains the equations to compute it. Thus, calling the following methods

```
void SimTK::SimbodyMatterSubsystem::multiplyByM () const
```

and

```
void SimTK::MattherSubsystem::multiplyBySystemJacobianTranspose() const
```

can compute the mass and intertia dynamics, and the gravity dynamics, respectively. The method calculates the product of the transposed kinematic Jacobian and the vector of forces — in this case gravity forces, one per body, in simulation time to produce a generalized force. In our case this generalized force will become double pendulum's gravity compensation. The same happens with Coriolis and centrifugal forces, there is a method to obtain them. This method is called

```
void SimTK::MatterSubsystem::calcResidualForceIgnoringConstraints()
```

The following step has been done is to add the controller to the OpenSim model in the `int main(){}` part. See the following core:

```
// Add a controller
PendulumController *pendControl = new PendulumController(-SimTK::Pi/1.5, 0.0);
pendControl->setActuators(osimPendulum.updActuators());
osimPendulum.addController(pendControl);
```

Listing 4.6: This code creates an object of type PendulumController, give the actuators to the controller, and add the controller in the OpenSim model.

Finally, the state has to be created and initialized. The integrator that will integrate the model has to be created, too. OpenSim offers some different algorithm to integrate models. In this case has been used the Kutta-Merson method [4] beucause it is the one used in the OpenSim tutorials. The listing 4.7 shows how to do it.

```
// Initialize system
SimTK::State &si = osimPendulum.initSystem();

// Set Up Visualizer
osimPendulum.updMatterSubsystem().setShowDefaultGeometry(true);
SimTK::Visualizer& viz = osimPendulum.updVisualizer().updSimbodyVisualizer();
viz.setBackgroundColor(SimTK::Black);

// Set integrator
SimTK::RungeKuttaMersonIntegrator integrator(osimPendulum.getMultibodySystem());
Manager manager(osimPendulum, integrator);
manager.setInitialTime(initTime); manager.setFinalTime(endTime);
manager.integrate(si);
        }
```

Listing 4.7: This code creates the state, the up the visualizer and set the integrator.

4.1.5 Results

This part shows the results obtained by the controller designed in the OpenSim. In this case, the double pendulum controller has been only created to demonstrate that it is possible to control a model in OpenSim. Thus, right now it is only important if the system can be stabilized or not. The values of K_p and K_v of each Joint controller can be seen in the table 4.3.

Table 4.3: Control parameters of each joint.

Control Parameters		
Joint 1	K_p	0.1
	Kv	2
Joint 2	K_p	0.08
	Kv	2

Let us check which response gives these values of K_p, and K_v for each joint.

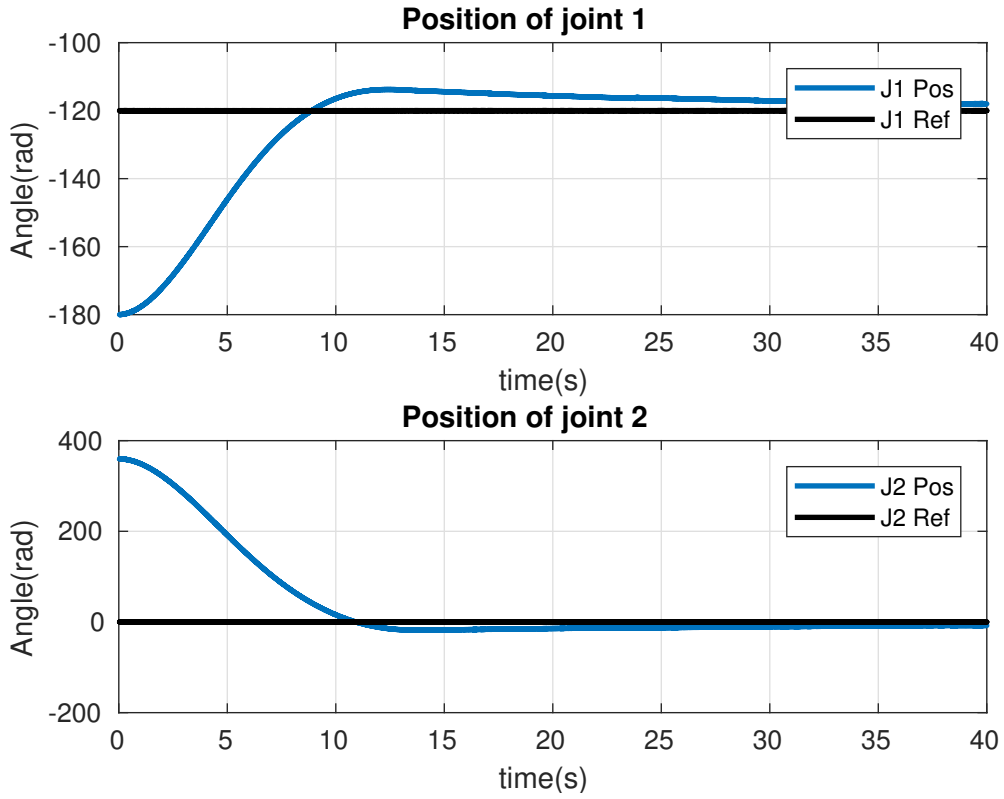


Figure 4.4: Position of joints 1 and 2 respect their references.

In this case, the references that have been chosen for each joint are 120°, and 0°for joint one and two, respectively. As the figure 4.4 shows, both joints reach his target point. As it can be seen the response of both actuators are very slow, as it was wanted. Moreover, joint one and two have an overshoot of 10.84 % and 31.55 %, respectively. The pic time in the first case it is equal to 12.55 s and in the second case it is equal to 13.8 s. If we analyse the form that takes the joint position's responses, we can conclude the roots of the system with these values of the controller are complex roots which are placed in the left half Pole-Zero plane. For this reason, there are overshoot in both cases.

Finally, it will be analysed the error produced for each joint: if we take a look at the figure 4.5, we can observe how the errors of joints one and two evolves in time.

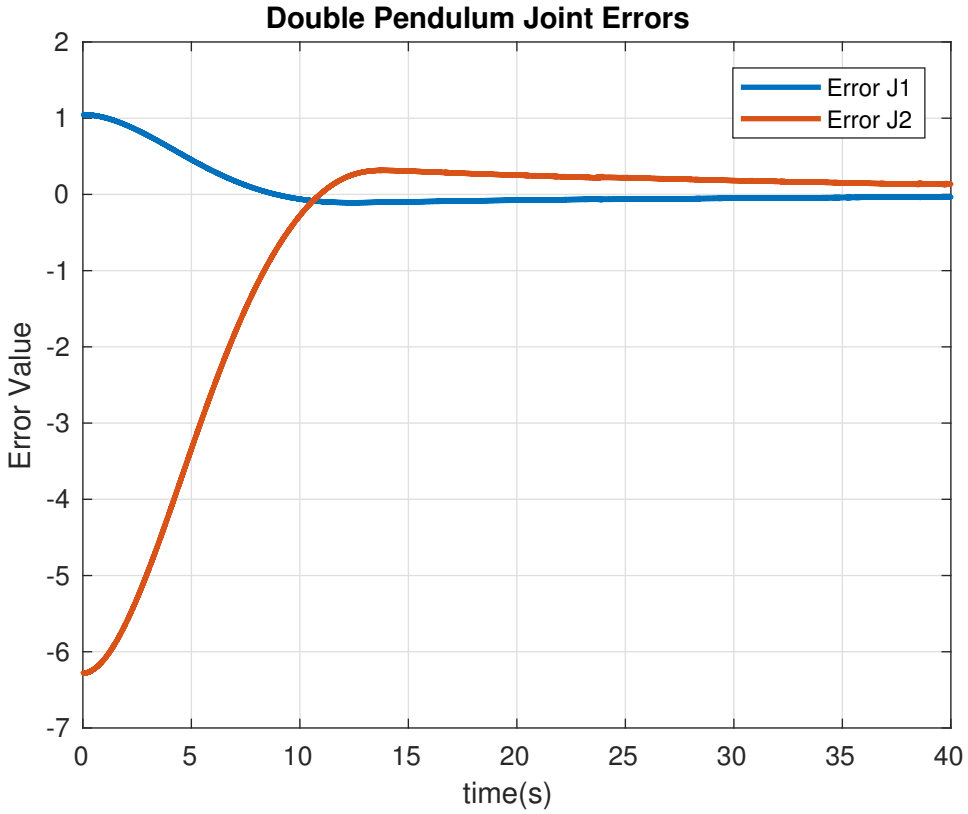
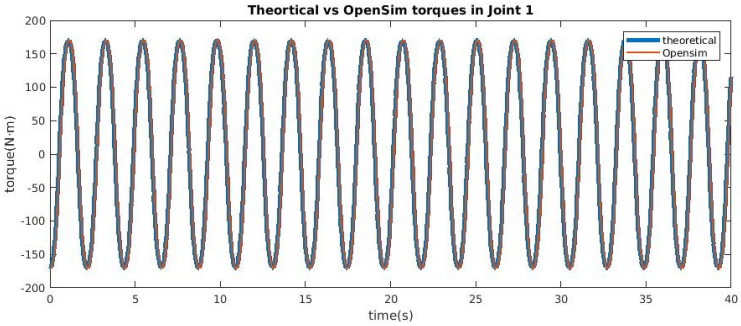


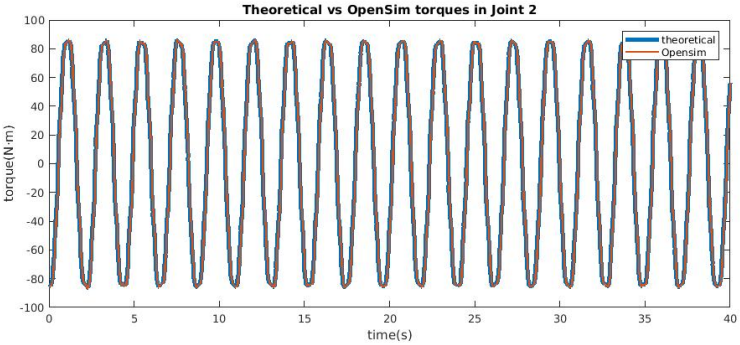
Figure 4.5: A plot of the both errors from joints one and two.

One interesting thing that can be observed in the error plot, it is joint 2 needs more time to reach the 0 error. It has sense, taking into account that K_p value of the controller of the joint's 2 actuator is lower than joint's one. At the end of the simulation when time it is equal to 40 seconds, the joint's one error it is equal to -0.0348, and the joint's two error it is equal to 0.1355.

Moreover, it has been done a test which has been consisted in comparing the theoretic gravity dynamic components of the equations 4.11, and 4.12 with the OpenSim gravity components. The test has been carried out with the following way: both pendulum joints have been set $210°$ referenced in ground axis and it has been released of this position without affecting the pendulum any external force, apart from gravity.



(a) Theoric and OpenSim gravities affecting Joint 1.



(b) Theoric and OpenSim gravities affecting Joint 2.

Figure 4.6: This graphics show the theoretical gravity and OpenSim gravity that affects each joint.

As it can be seen, the theoretical part, and the OpenSim part seems to be equal but in order to check if the error is 0 between the theoretical and OpenSim gravity components, it has been computed the error between them —That is.

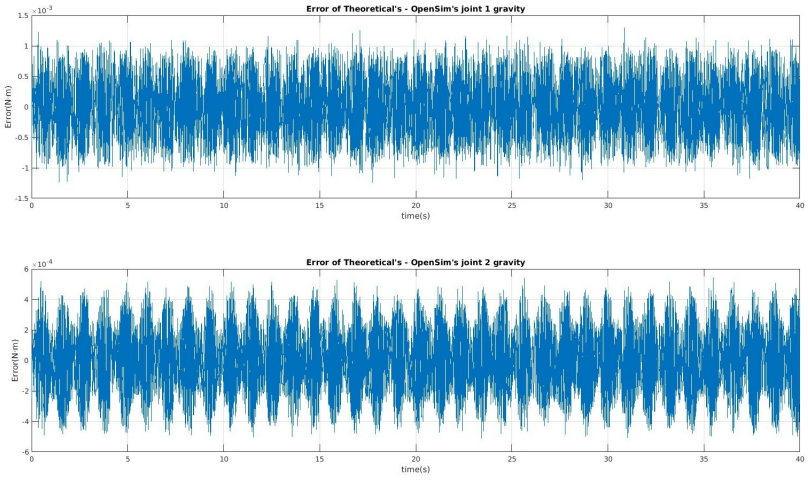


Figure 4.7: Error between theoretical and OpenSim gravity components of both joints.

The error between both components are very low, but they are not equal to 0. Check the figure 4.7

5 Exoskeleton Implemented in OpenSim

This section it is going to explain the main points of the exoskeleton designed, and how it has been implemented in OpenSim. In order to avoid large waiting times when integrating the model, it has been opted to implement the exoskeleton only in a body leg. Thus, only the right parts of a body, and the exoskeleton —including pelvis— have been implemented.

5.1 Exoskeleton

The exoskeleton has been created in OpenSim from scratch. As the double pendulum example, the different bodies that make up the robotic orthosis have been defined, and set the different inertias and masses of each one. These parameters have been obtained from a Simulink model just created for being used in Matlab. In this way, the model it is composed of an exoskeleton foot, shin, thigh and hip. The masses and inertia values of each body part can be seen in the tables 5.1, and 5.2 respectively.

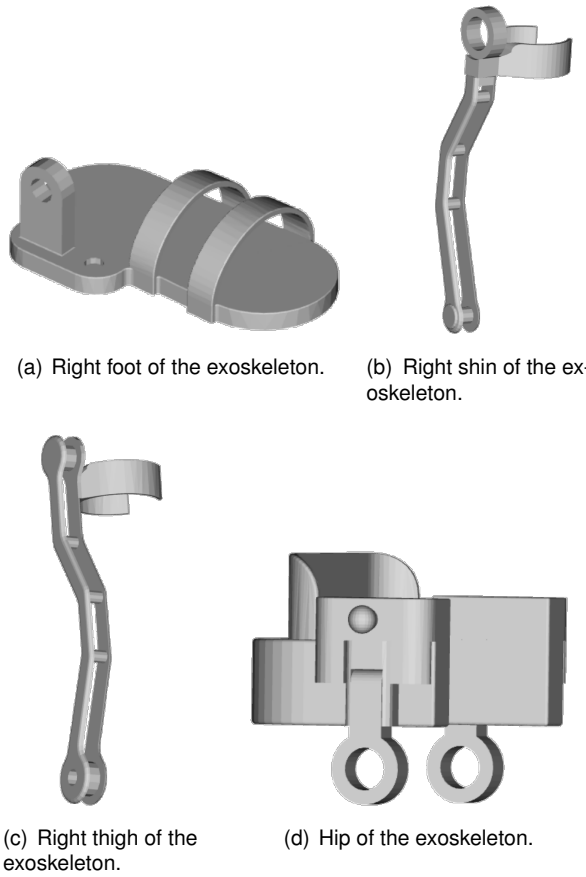
Table 5.1: Masses of each exoskeleton body.

Masses	
Foot	0.5107 kg
Shin	0.1957 kg
Thigh	0.2138 kg
Hip	0.8531 kg

Table 5.2: Inertia values of each exoskeleton body in each matrix's position.

Inertia						
	xx	**yy**	**zz**	**xy, yx**	**xz, zx**	**yz, zy**
Foot	0.3374	0.0085	0.3378	-0.0271	0.0025	-0.0333
Shin	0.0589	0.0032	0.0565	0.0022	0.0005	-0.0121
Thigh	0.0155	0.0047	0.0139	-0.00314	0.0020	-0.0052
Hip	0.0125	0.0144	0.0069	0.0021	0.0001	0.0

OpenSim API allows to use Object Files (obj), in other words, 3D models designed with 3D tools as SolidWorks. In this way, a 3D exoskeleton model has been implemented in order to show a 3D visualization during the simulation. It does not pretend to have the same appearance as H1 Exoskeleton, but the main objective of it is distinguishing both bodies during the simulation. The files, acquired from GrabCad page [15], were in `.stl` extension. As the OpenSim API only accepts files of type ".obj", these ones have been transformed to obj format with FreeCad. The shape of each exoskeleton body part can be seen in the figure 5.1



(a) Right foot of the exoskeleton.

(b) Right shin of the exoskeleton.

(c) Right thigh of the exoskeleton.

(d) Hip of the exoskeleton.

Figure 5.1: The subfigures a, b, c, and d, show the different 3D body parts of the exoskeleton implemented in the OpenSim model.

The previous body parts have been joined among themselves using the tools that OpenSim provides. As the double pendulum case, it has been used Pinjoint joints —table 4.2. In this way, right foot of the exoskeleton, and right shin of the exoskeleton, and the second one with the right thigh of the exoskeleton, and so on, have been joined with a PinJoint. This means that only rotation in the z axis is allowed for each exoskeleton's joint. Finally, the exoskeleton takes the following appearance:

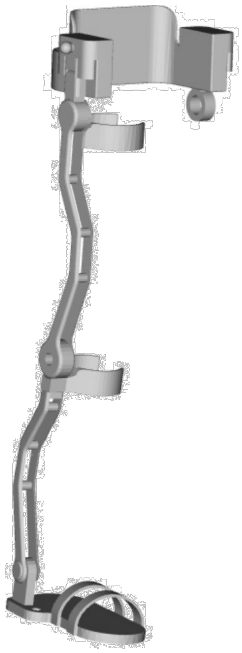


Figure 5.2: Full 3D exoskeleton OpenSim model.

In order to illustrate how the bodies have been linked, the OpenSim GUI offers a tool that allows to see which bodies form each joint, this can be seen in the figure 5.3.

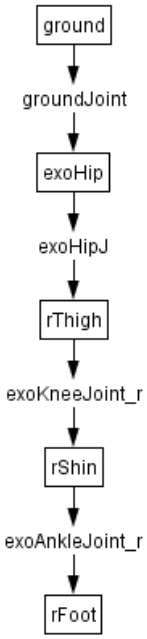


Figure 5.3: Bodies and joints scheme of the exoskeleton.

5.2 Controller of the Exoskeleton

As it has been said, the behaviour of the exoskeleton will be similar of a triple pendulum model. Thus, the dynamics have not been computed because the equations expressions becomes complex and longer to compute. Moreover, the dynamics of a double pendulum have been already computed and checked that the theoretical and OpenSim results were similar. In this way, it is going to compute the dynamic of the exoskeleton using the tools that OpenSim offers, and use it jointly a control-law to compute to design a controller to test the model.

```
1 void SimTK::SimbodyMatterSubsystem::multiplyByM(const State & state,
2                                                 const Vector & a,
3                                                 Vector & Ma
4                                                 ) const
```

Listing 5.1: This code computes the torque generate by the inertia and masses of the model giving to it the accelerations [18].

The code of the listing 5.1 does the hard work of computing the torques generated by the inertia and masses of the body system giving to the function the accelerations. In this case, the accelerations will be the desired ones which have been computed by the controller.

Each joint has his own controller based on a Proportional Integral Derivative (PID) which takes the form of the following equation

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_v \dot{e}(t), \tag{5.1}$$

where K_p, K_i, and K_v are scalar values that are selected by the user. Taking into account these values the response of each joint will be different. In this case, it is wanted a slow response because it is a system prepared to work with humans, and it is wanted to avoid the risk of hurting them. This means, the K_p value is going to be low. The following code contains the algorithm to compute the desired acceleration of exoskeleton hip joint:

```
double PositionController::
computeDesiredAngularAccelerationHipJoint(const State &s,
                                          double* getErr = nullptr,
                                          double* getPos = nullptr) const
{
        // Obtain the coordinateset of the joint desired
        const Coordinate& coord = _model->getCoordinateSet().get("exoHipJoint_coord");
        double xt = coord.getValue(s); // Actual angle x(t)
        *getPos = xt;
        static double errt1 = hipJointRefAngle;
        double err = hipJointRefAngle - xt; // Error
        double iErr = (err + errt1)*0.033; // Compute integration
        double dErr = (err - errt1)/0.033; // Compute the derivative of xt
        errt1 = err;
        *getErr = err;
        // cout << "Error ExoHipJoint = " << err << endl;
        double  kp = k_exohip[0], kv = k_exohip[1], ki = k_exohip[2];
        double desAcc = kp*err + kv*dErr + ki*iErr;
        return desAcc;
}
```

Listing 5.2: This code computes the desired acceleration generated by the error created between the reference position and actual state.

Other tools that have been used are methods to compute Coriolis and centrifugal, and gravity forces. The code of each one can be seen in the listings 5.3, and 5.4.

```
void SimTK::SimbodyMatterSubsystem::
        calcResidualForceIgnoringConstraints( const State & state,
                                              const Vector & appliedMobilityForces,
                                              const Vector_< SpatialVec > & appliedBodyForces,
                                              const Vector & knownUdot,
                                              Vector & residualMobilityForces
                                              ) const
```

Listing 5.3: This code computes the torque generate by the coriolis, and centrifugal forces, or other forces that can affect the system [18].

```
SimTK::Vector calcGravityCompensation(const SimTK::State &s) const
{
        SimTK::Vector g;
        _model->getMatterSubsystem().
                multiplyBySystemJacobianTranspose(s,
                                                  _model->getGravityForce().getBodyForces(s),
                                                  g);
        return g;
}
```

Listing 5.4: This method return a vector which contains the gravity components of each joint.

Once obtained the values that gives these methods, it is applied the formula of the dynamics 4.11 so that compute the total torque needed for each joint with the objective to achieve the reference set by the user. Finally, only it is needed to take this torque values and send them to the actuators. The code of the listing 5.5 computes the total torques, and the listing 5.6.

```
double nTorqueExoHip = (dynTorque.get(0)
                        + coriolis.get(0)
                        - (grav.get(0))
                        )/_model->getActuators().get("exoHipActuator_r").getOptimalForce();
double nTorqueExoKnee = (dynTorque.get(1)
                        + coriolis.get(1)
                        - grav.get(1)
                        )/_model->getActuators().get("exoKneeActuator_r").getOptimalForce();
double nTorqueExoAnkle = (dynTorque.get(2)
                        + coriolis.get(2)
                        - grav.get(2)
                        )/_model->getActuators().get("exoAnkleActuator_r").getOptimalForce();
```

Listing 5.5: This piece of code computes the torque that has to be applied to each joint in order to achieve the desired position.

```
Vector torqueControl(1, -nTorqueExoHip);
_model->updActuators().get("exoHipActuator_r").addInControls(torqueControl, controls);
Vector torqueControl2(1, -nTorqueExoKnee);
_model->updActuators().get("exoKneeActuator_r").addInControls(torqueControl2, controls);
Vector torqueControl3(1, nTorqueExoAnkle);
_model->updActuators().get("exoAnkleActuator_r").addInControls(torqueControl3, controls);
```

Listing 5.6: The code sends the total torque values to each joint of the exoskeleton.

5.3 Code developed

The C++ code of the exoskeleton has been designed meticulously. In this way, the code for future developers is easier to understand. All the code created for this model have been divided in different classes which each of them have been written in different files. These parts are:

- Exoskeleton Main Program (exoMain.cpp): This file will contain all the code needed to call the other files and classes to make the exoskeleton work, and the integrator.
- Configuration (configuration.h): it is special file written to put all the exoskeleton parameters there, making the change of them, in case of needing it, easier. Moreover, the file allows to enable and disable gravity, friction in each joint, and so on.
- Exoskeleton Body (exoskeleton.h/.cpp): These files contains a class with series of methods which defines: the bodies, the type of link between these bodies, the actuactors in each joint, the coordinates range of each joint, and so on.
- Controller (positioncontroller.h/.cpp): The controller files contain a class with all the needed methods in order to take control upon to all joints. Some of them are:
  - `computeDesiredAngularAccelerationHipJoint()`
  - `computeDynamicsTorque()`
  - `computeGravityCompensation()`

- Print OpenSim Information (printOpenSimInformation.h/.cpp): These files contains a library that has been developed in order to print variables in a console or in a file.

5.4 Results

The test has been carried out has consisted in giving to each joint a constant reference —table 5.3, and the values of each PID controller used can be found to the table 5.4.

Table 5.3: Angle target

Joints	**Angle Value**
ExoHip Joint	50º
ExoKnee Joint	-93.3º
ExoAnkle Joint	20º

Table 5.4: Exoskeleton PID parameters

Joints	K_p	K_i	K_v
ExoHip Joint	1.25	4.0	5.0
ExoKnee Joint	0.75	4.0	5.0
ExoAnkle Joint	0.5	4.0	5.0

The PID parameters have been chosen randomly, but with a low K_p value, in order to check if the controller works and trying to produce a slow response. In this way, with the previous references, and the PID's values have been obtained the following results:

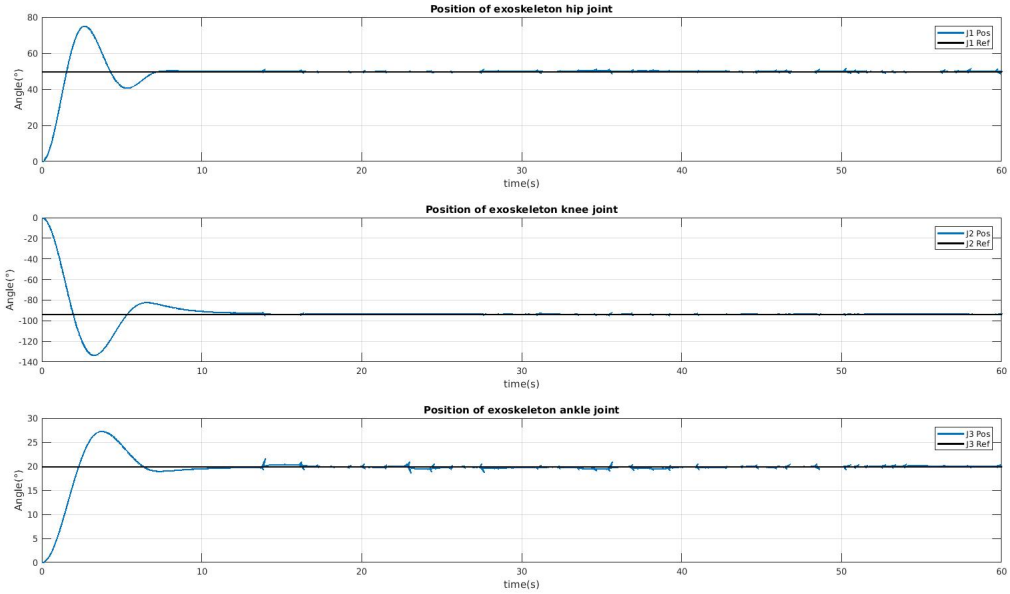


Figure 5.4: Response of each exoskeleton joint.

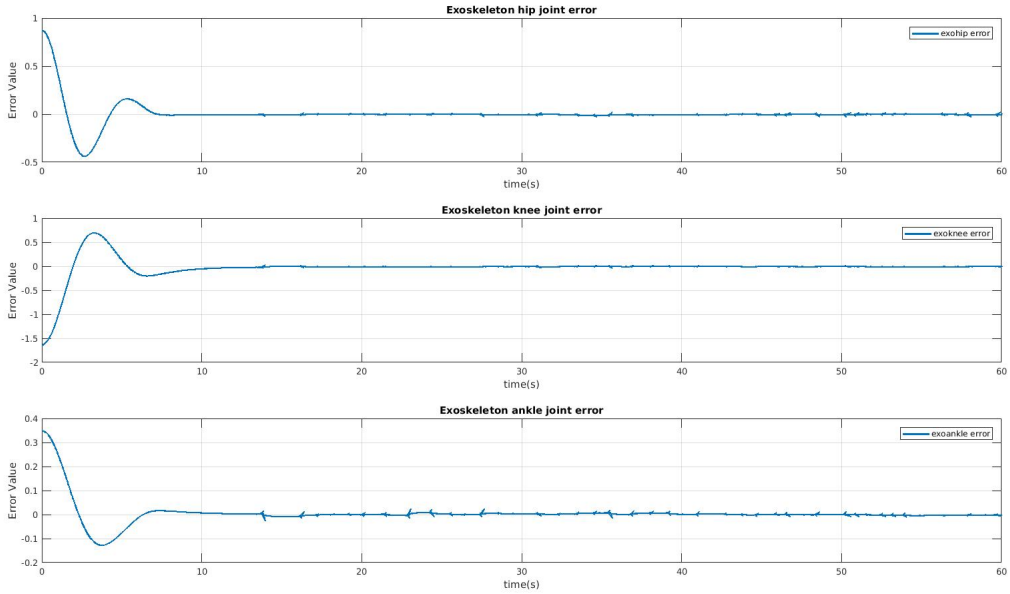


Figure 5.5: Error of each exoskeleton joint.

The response of the system obtained in each joint is a second order response [16]. This means that the poles of the system are in the left half Pole-Zero plane [17], and are unreal. For this reason, there is an overshoot in all responses —figure 4.4. For example, the exoskeleton ankle joint exceeds 12.68 % the reference value of 20°. Moreover, the response of the system is very slow, it takes 8 seconds to stabilize. It

has to be highlighted that all joints reach his target with an error almost 0.

6 Exoskeleton Coupled to OpenSim Model

This section is the main proposal of this project because the coupling of a mechanical exoskeleton to a OpenSim Model has not been implemented yet with the OpenSim C++ API, or at least it has not been documented. Taking into account that the main parts of OpenSim has already been explained —create a model, integrate it— in the previous section, this part will focus on which musculoskeletal OpenSim model has been used, how to couple it to the exoskeleton's model, how to control all the system, and the results obtained.

6.1 OpenSim Musculoskeletal Body

OpenSim offers at his official page a variety of musculoskeletal body models that are open-source and available for users. The model chosen was `leg6dof9musc` [10]. It is a simplified skeleton model which has 6 degrees of freedom and 9 muscles. It is composed of 7 bodies which represent four physiological segments —pelvis, thigh, shank, and foot. In order to call it, it is going to write the model name when the object of type Model is declared

```
OpenSim::Model osimModel("skeleton.osim")
```

. The model has been modified so that can be adapted to our necessities. In this case, the muscles have been disabled and the pelvis has been fixed at position (0,0,0) of the ground —the listing 6.1 is in charge of it. The main idea behind fixing the model is to make it behaves like a triple pendulum but with some constraints. These constraints will be due to limits of body articulations.

```
1 const Vec3 locInParent(0), orInParent(0), locInBody(0), orInBody(0);
2 WeldJoint* pelvisGround = new WeldJoint("pelvisground",
3                                         osimModel.getGroundBody(),
4                                         locInParent,
5                                         orInParent,
6                                         osimModel.updBodySet().get(1),
7                                         locInBody,
8                                         orInBody);
```

Listing 6.1: This code place the leg6dof9musc model in 0 and attach it in the ground with a WeldJoint.

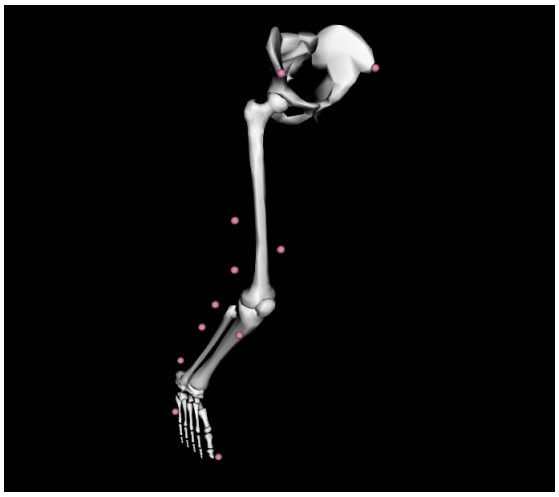


Figure 6.1: Visualization of the OpenSim leg6dof9musc model.

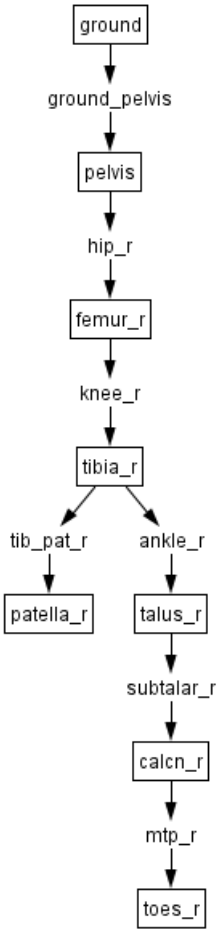


Figure 6.2: Bodies and joints of OpenSim leg6dof9musc model.

6.2 Exoskeleton with the Musculoskeletal Body

Some modifications have been needed to do with exoskeleton's code. The main modifications affect directly at the exoskeleton joints. Now, the exoskeleton hip is not fixed to (0,0,0) coordinate of the OpenSim API. Moreover, the joints of the exoskeleton in OpenSim have been defined from pelvis to foot, but in this case, it has been defined the other way around, from foot to pelvis because the first one has been attached to the musculoskeletal's foot with a WeldJoint joint. —See the figure 6.3.

```
// Create a weldjoint behind skeleton and exoskeleton foot.
const Vec3 exoBodyLocInParent(-0.15, 0.07, 0.1), exoBodyOrInParent(0),
                                  exoBodyLocInBody(0, 0, 0), exoBodyOrInBody(0);
                                  WeldJoint exoBodyJ_r ("exoBodyJoint_r",
                                  osimModel->updBodySet().get(7), // toe_r
                                  exoBodyLocInParent,
                                  exoBodyOrInParent,
                                  rFoot,
                                  exoBodyLocInBody,
                                  exoBodyOrInBody);
jointSet->insert(jointSet->getSize(),exoBodyJ_r);
bodySet->insert(bodySet->getSize(),rFoot);
```

Listing 6.2: This code is in charge of create a weld joint between the right toe and the foot of the exoskeleton.

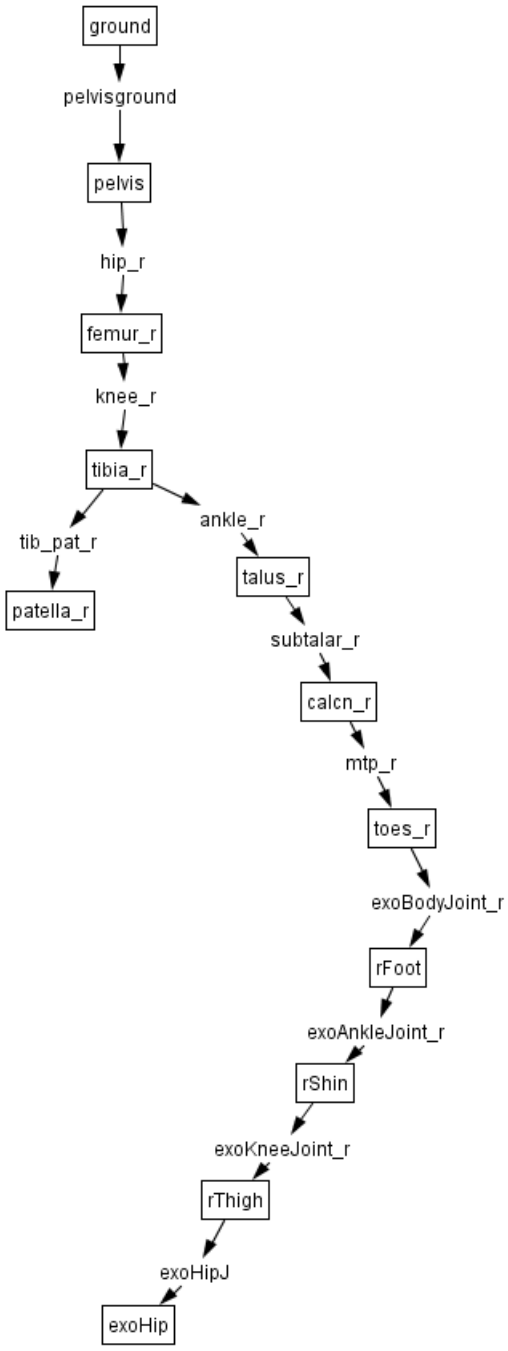


Figure 6.3: Bodies and joints scheme once both systems have been coupled to each other.

The main reason of doing this is to make easier the fact of coupling both systems because only coupling them with forces make the system completely unstable.Thus, the rest of the parts of the exoskeleton which will be coupled with the musculoskeletal body with some types of forces. OpenSim provides an extended list of forces that are already defined:

- Actuator,
- Bushing force,
- Coordinate limit force,
- External force,
- and others.

Moreover, with the OpenSim API it is easy to create your own class derived from Force class with the aim of creating your own force. In this case, after understanding what exactly do each one of them, it has been opted to couple both models with bushing forces, the main reasons of it will be explained in the next section.

6.2.1 Bushing Forces

There are lots of types of forces that can be implemented in OpenSim [19]. One of them are bushing forces. This type of force has six degrees of freedom, three translations and three rotations, they can be free or constrained. In the last case, these constraints can be translational and rotational stiffness and damping forces. In other words, it can create translational and rotational springs in six degrees of freedom of a specific point. This means bushing forces are ideal to combine two different objects in a point. For this reason, bushing forces have been used to model the exoskeleton body grippers.

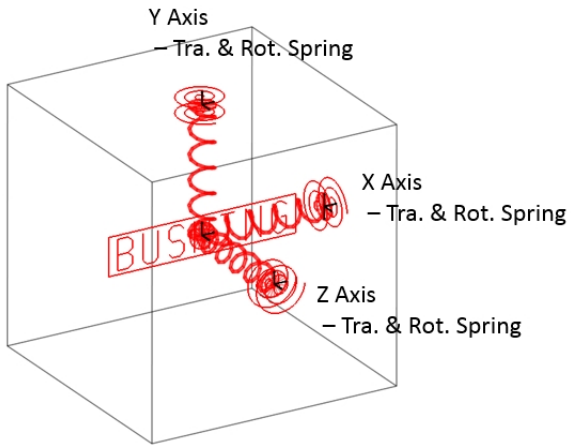


Figure 6.4: Illustration of a bushing force in a center of a cube [21].

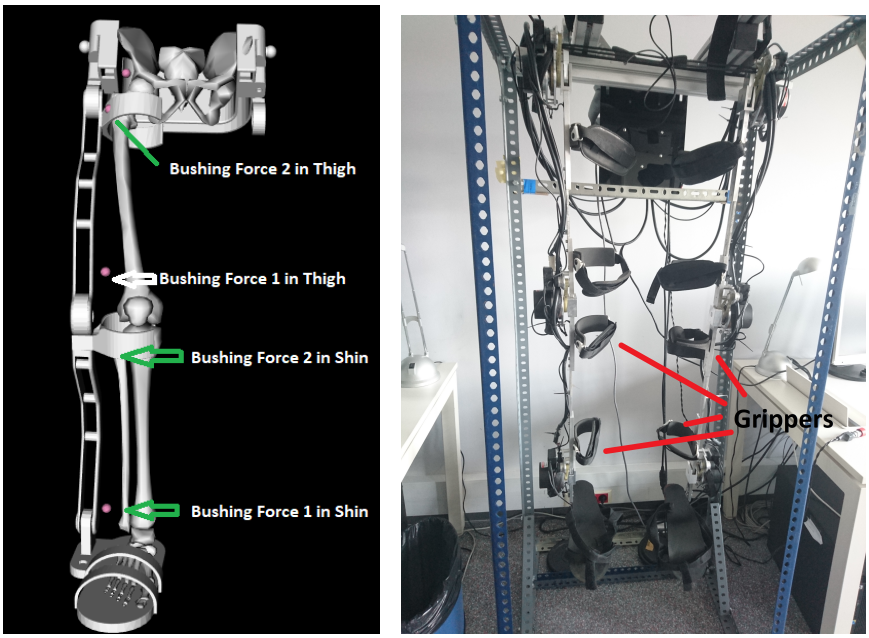
OpenSim has his own method to apply a bushing force between two bodies —listing 6.3. The parameters of this method are: the bodies names; the point of each body where the force will be applied; the values of translational, and rotational damping; and translational and rotational stiffness. The main problem that supposes use this type of force in OpenSim it is that both points has to be aligned —exoskeleton and musculoskeletal, if not, the model becomes to have a strange behaviors such as oscillations and vibrations.

In order to place this points the OpenSim GUI has to be used because it allows to place visual markers and this makes things easier than doing it with OpenSim API.

```
OpenSim::BushingForce::BushingForce( const std::string & body1Name,
                                     const SimTK::Vec3 & point1,
                                     const SimTK::Vec3 & orientation1,
                                     const std::string & body2Name,
                                     const SimTK::Vec3 & point2,
                                     const SimTK::Vec3 & orientation2,
                                     const SimTK::Vec3 & transStiffness,
                                     const SimTK::Vec3 & rotStiffness,
                                     const SimTK::Vec3 & transDamping,
                                     const SimTK::Vec3 & rotDamping)
```

Listing 6.3: This code generates a bushing force between two bodies. [18].

Taking into account that each leg of the H1 Exoskeleton has four grippers to subject a human leg —two for shin, and two more for thigh, four bushing forces have been applied in the OpenSim model. Two of them have been used for linking shins of both bodies, and the last two have been used to link the thigh of the exoskeleton with the thigh of the musculoskeletal body. Finally, one more bushing force has been used for subjecting both model's hips.



(a) Exoskeleton OpenSim Model. (b) H1 Exoskeleton

Figure 6.5: 5(a) shows the bushing force points, and 5(b) shows the H1 grippers.

6.3 Controller

The control-law implemented in this case, it is the same as the exoskeleton without being coupled to any musculoskeletal body, in other words, a PID. Thus , it is implemented the equation 5.1. However, the dynamics of the global system have been changed because right now, the system not only has the joints of the exoskeleton, but also has the joints of the musculoskeletal body. In this way, the dynamic matrix $M(\Theta)$, the centrifugal and Coriolis matrix $C(\Theta, \dot{\Theta})$, and the gravity matrix $N(\Theta, \dot{\Theta})$, it has increased their dimensions taking into account these joints which can be seen in the figure 6.3. It has to remark that WeldJoints are not taking into account in this matrix because they do not have any DoF. Thus, the joints called: pelvisground, patella_r, subtalar_r, and exoBodyJoint_r are WeldJoints. Thus, taking into account that there are seven joints with one DoF each one, the number of torques that it is going to obtain in the dynamic equation are equal to seven.

Knowing the previously said, and considering that there are only three actuators, it has been considered that the thigh of the exoskeleton and the thigh of musculoskeletal model, and so on, are the same body. Thus, the accelerations that affect to each exoskeleton part, also affect to their respectively coupled musculoskeletal body in the same manner. So taking into account this, the desired accelerations obtained in each actuator controller that multiplies in the dynamic equations the components of the exoskeleton, will multiply the musculoskeletal components, too.

As it has been seen in the exoskeleton description, there are three actuators and three methods to compute the desired accelerations:

- `computeDesiredAngularAccelerationHipJoint()`
- `computeDesiredAngularAccelerationKneeJoint()`
- `computeDesiredAngularAccelerationAnkleJoint()`

The method `computeDynamicsTorque()` is in charge of taking the tree desired accelerations and organize each of them in a vector to be multiplied by the corresponding $M(\Theta)$ components. In this way, it is obtained seven torques which will be returned by the method inside a variable of type Vector. — See the listing 6.4.

```
SimTK::Vector PositionController::computeDynamicsTorque(const State &s,
                                                        double* getErrs = nullptr,
                                                        double* getPoses = nullptr) const
{
        double desAccExoHip = computeDesiredAngularAccelerationHipJoint(s,
                                                                        &getErrs[0],
                                                                        &getPoses[0]);
        double desAccExoKnee = computeDesiredAngularAccelerationKneeJoint(s,
                                                                        &getErrs[1],
                                                                        &getPoses[1]);
        double desAccExoAnkle = computeDesiredAngularAccelerationAnkleJoint(s,
                                                                        &getErrs[2],
                                                                        &getPoses[2]);
        SimTK::Vector_<double> desAcc(7);
        desAcc[0] = desAccExoHip; desAcc[1] = desAccExoKnee;
        desAcc[2] = desAccExoKnee; desAcc[3] = desAccExoAnkle;
        desAcc[4] = desAccExoAnkle; desAcc[5] = desAccExoKnee;
        desAcc[6] = desAccExoHip;

        SimTK::Vector torques;
        _model->getMatterSubsystem().multiplyByM(s, desAcc, torques);

        return torques;
}
```

Listing 6.4: This code compute the torques generated by the $M(\Theta)$ matrix multiplied by the desired accelerations.

The gravity, coriolis and centrifugal forces are obtained from the methods

`computeGravityCompensation` and `computeCoriolisCompensation`, respectively. Finally, all the components obtained in each method are reorganized in order to compute the total torque that each actuator of the exoskeleton needs to reach the target.

```
   double getErrs[3];
   double getPoses[3];
   SimTK::Vector dynTorque(computeDynamicsTorque(s, getErrs, getPoses));
   SimTK::Vector grav(computeGravityCompensation(s));
   SimTK::Vector coriolis(computeCoriolisCompensation(s));

   double nTorqueExoHip = (dynTorque.get(0) + dynTorque.get(6)
   + coriolis.get(0) + coriolis.get(6)
   -(grav.get(0) + grav.get(6))
   )/_model->getActuators().get("exoHipActuator_r").getOptimalForce();
   double nTorqueExoKnee = (dynTorque.get(1) + dynTorque.get(2) + dynTorque.get(5)
   + coriolis.get(1) + coriolis.get(2) + coriolis.get(5)
   -(grav.get(1) + grav.get(2) + grav.get(3))
   )/_model->getActuators().get("exoKneeActuator_r").getOptimalForce();
   double nTorqueExoAnkle = (dynTorque.get(3) + dynTorque.get(4)
   + coriolis.get(3) + coriolis.get(4)
   - (grav.get(3) + grav.get(4))
   )/_model->getActuators().get("exoAnkleActuator_r").getOptimalForce();
```

Listing 6.5: This code compute the total torques.

6.4 Results

The first test that has been done is to give a specific position for each joint of the exoskeleton —table 6.1. The parameters of each PID that controls the actuators have been set at the same values of the exoskeleton without being coupled, —table 5.4. In this way, the response of each actuator is:

Table 6.1: Angle target

Joints	Angle Value
ExoHip Joint	-60º
ExoKnee Joint	40º
ExoAnkle Joint	0º

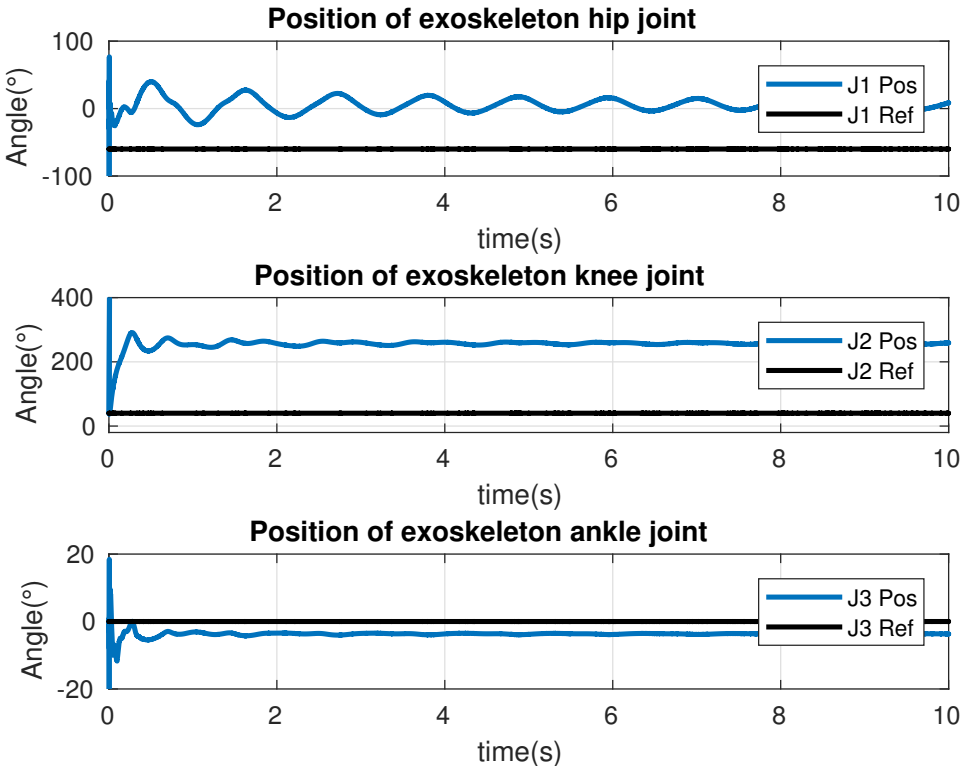


Figure 6.6: Positions of exoskeleton joints respect the reference set.

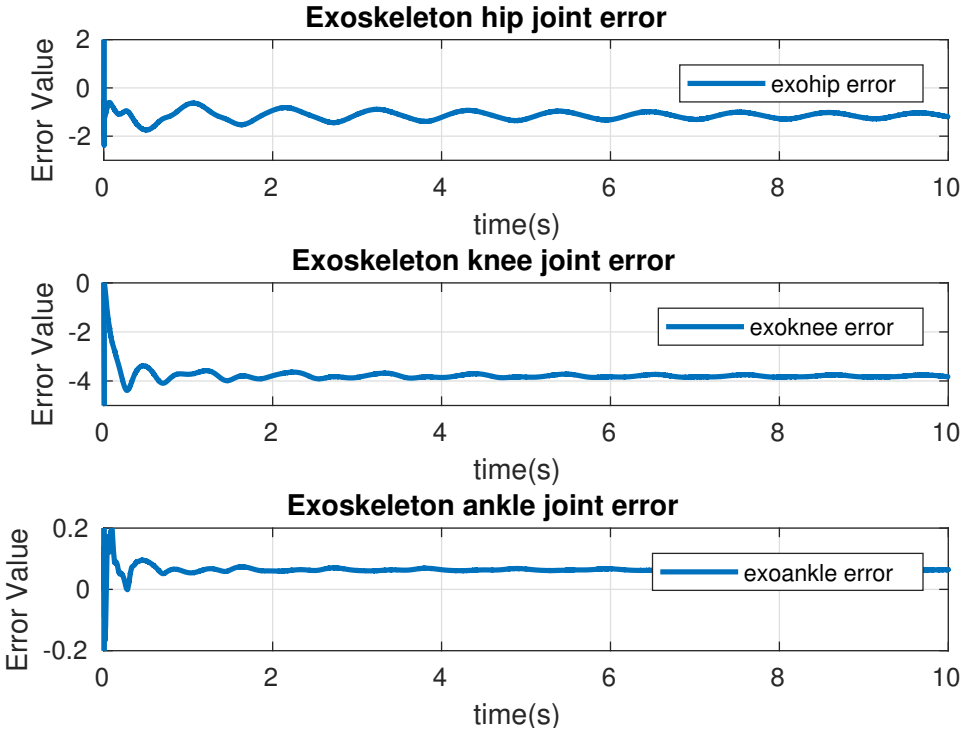


Figure 6.7: Error of exoskeleton joints.

As the figures 6.6, and 6.7 show there are very high overshoots in the response of the three joints. The error obtained at time 8 seconds in hip, knee, and ankle joints of the exoskeleton are -116 %, -380 %, and 6.48 %, respectively. These results indicates that some type of force is affecting to the system and it is not taking into account. In order to understand what is happening, it will be checked the inertial and masses torques, the centrifugal and Coriolis torques, and the torques generated by the gravitational forces.

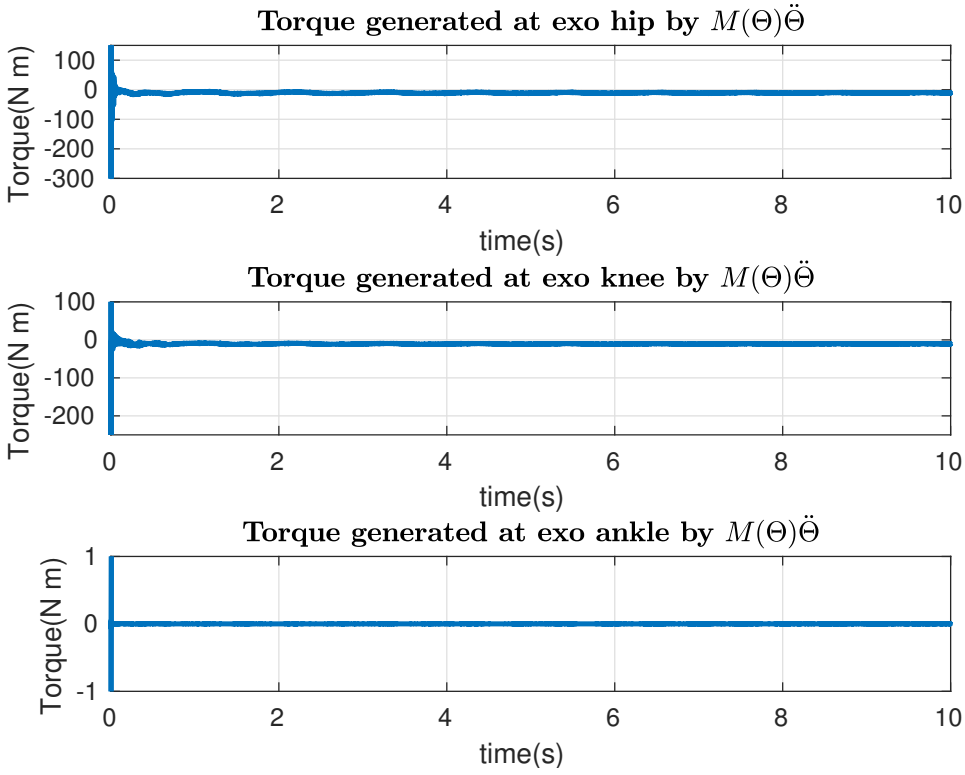


Figure 6.8: Positions of exoskeleton joints respect the reference set.

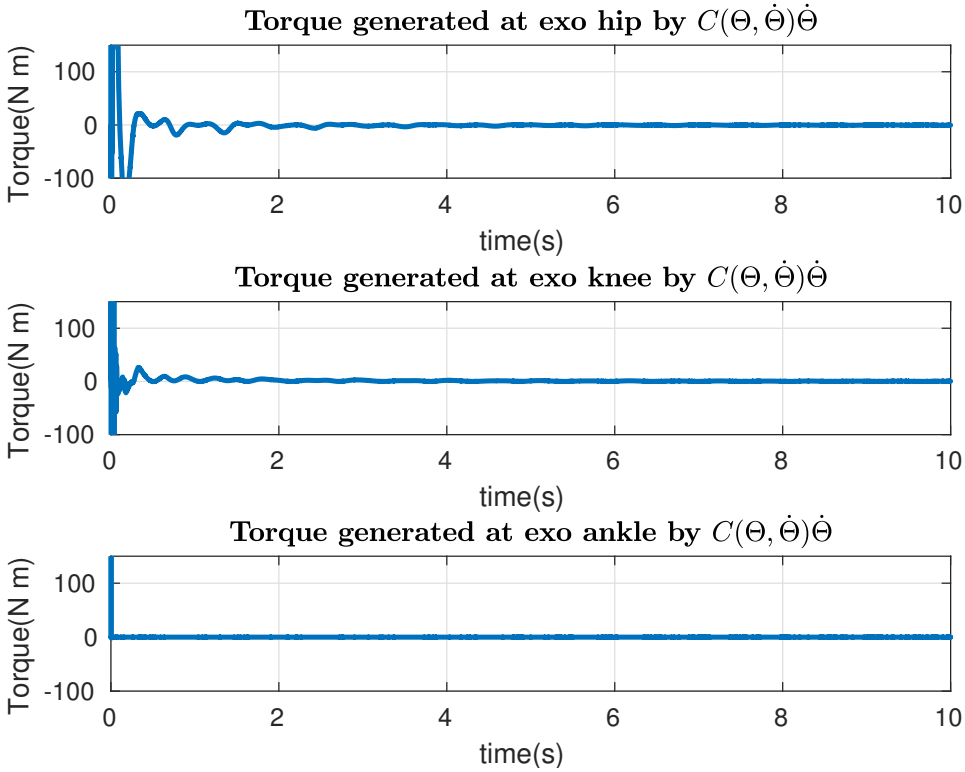


Figure 6.9: Error of exoskeleton joints.

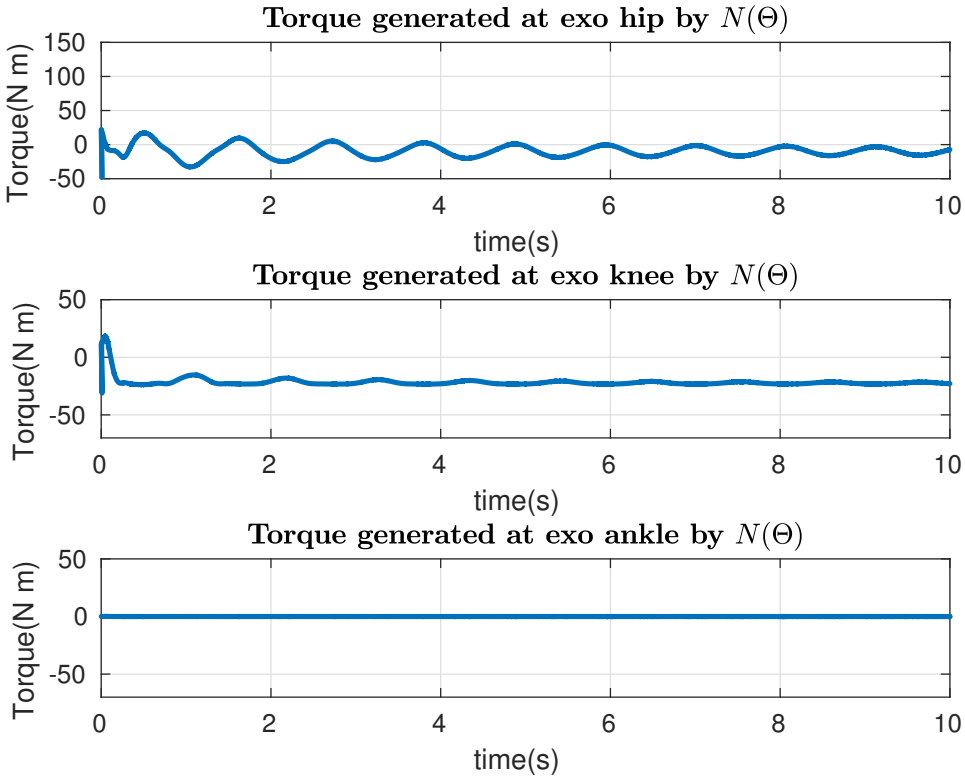


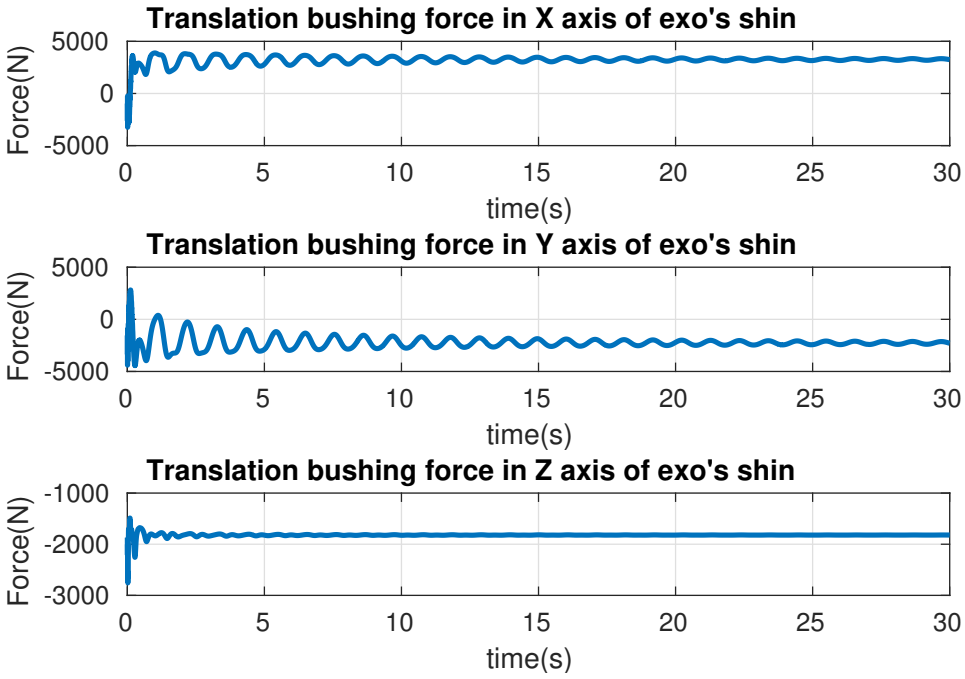
Figure 6.10: Error of exoskeleton joints.

The figures 6.8, and 6.9 indicate each exoskeleton's joint is stabilized at a point but with an offset too high. Thus, the controller it is working but not with the proper manner. In order to understand what is happening another test it is going to carry out. Before of doing it, let us check what values the bushing forces take.

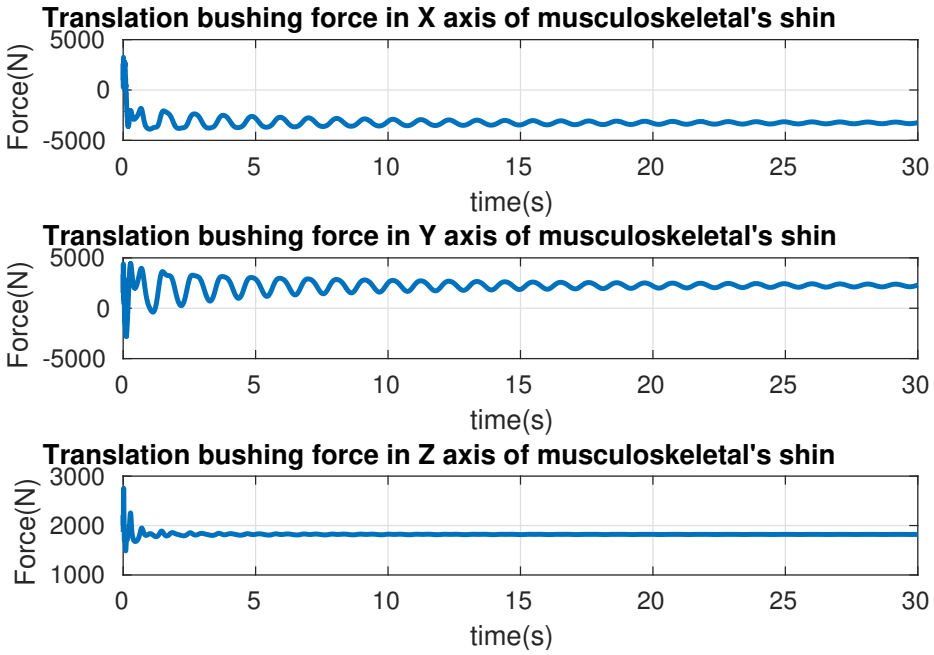
The figure 5(a) shows there are four points where bushing forces have been applied, additionally has been added another one in the hip. Due to each bushing forces gives twelve plots, only one has been chosen —bushing force applied in shin's point 1– to show in the results. The other plots can be found in the annexes.

Translational bushing forces

The first thing that can be observed in the figure 6.11 is the translational forces that actuates to the exoskeleton's shin are the negative of the translational forces that actuates to musculoskeletal's shin. This fact can be observed in the figure 6.12 in which is computed the sum of each translational force, and the result is equal to 0 in each axis.



(a) Exoskeleton's shin bushing translational forces



(b) Musculoskeletal's shin translational forces

Figure 6.11: Exoskeleton's and Musculoskeletal's shins translational bushing forces

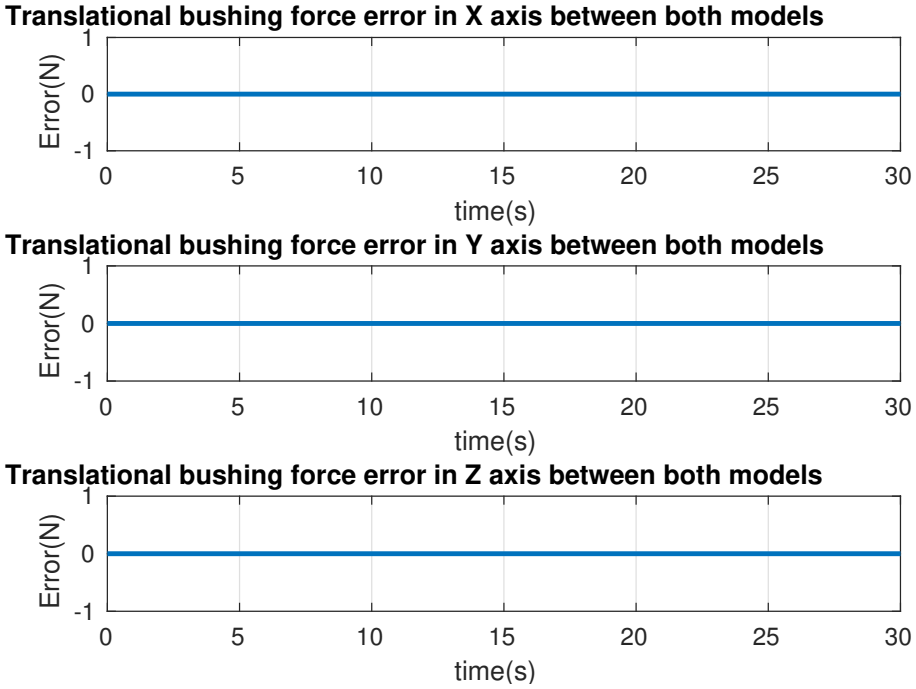
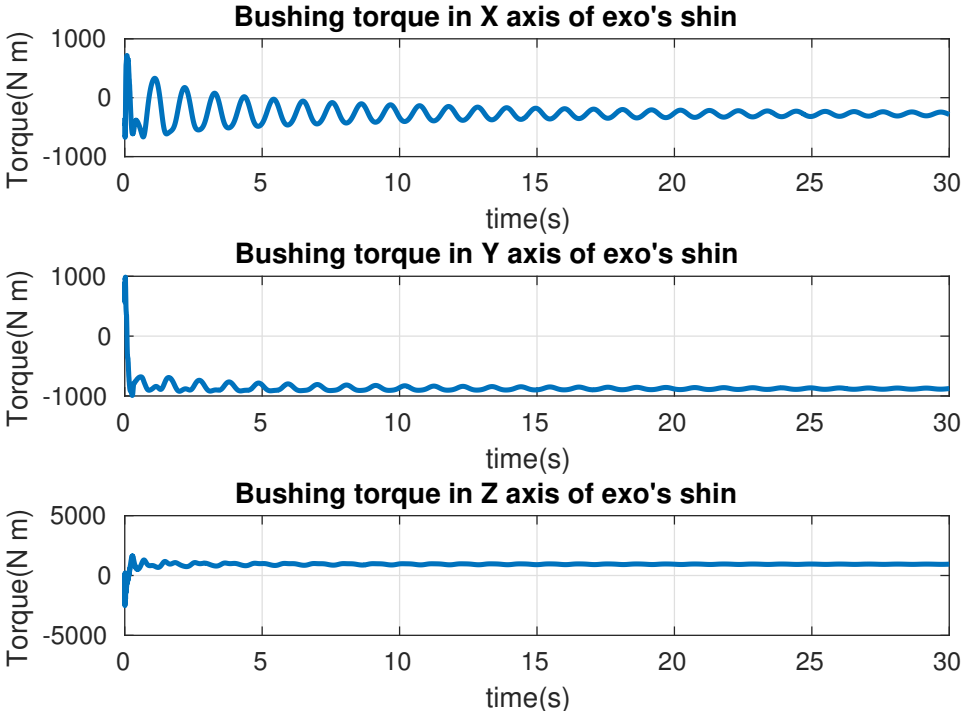


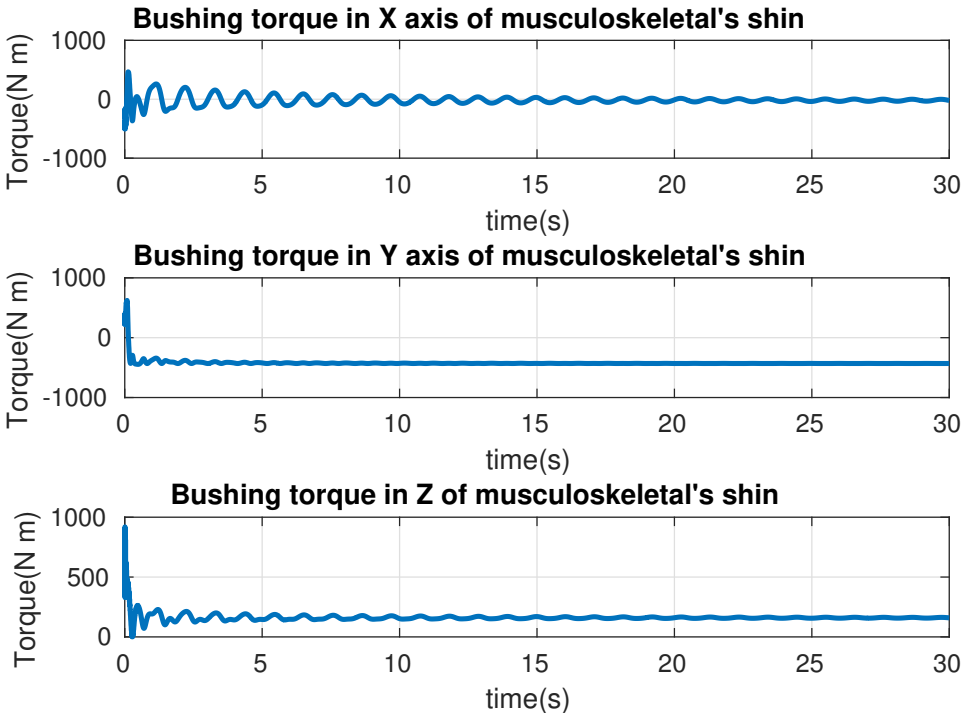
Figure 6.12: Sum of translational bushing forces of exoskeleton's shin and musculoskeletal's shin.

Bushing torque forces

By contrast, If the bushing torques of both models are compared, the value of bushing torques of exoskeleton's shin are not equal to the negative bushing torques of musculoskeletal's shin. These are reflected in the figure 6.13.



(a) Exoskeleton's shin bushing torque forces.



(b) Musculoskeletal's shin torque forces.

Figure 6.13: Exoskeleton's and Musculoskeletal's shins bushing torque forces.

6.4.1 Second Test

This test has consisted in increasing the values of K_p, K_i, and K_v in order to check if scaling the controller, the errors of the system will decrease. Thus, the table 6.2 are the new PID's parameters.

Table 6.2: Exoskeleton PID's paramaters

Joints	K_p	K_i	K_v
ExoHip Joint	40.0	50.0	120.0
ExoKnee Joint	40.0	50.0	120.0
ExoAnkle Joint	1.0	1.0	5.0

Using the same references as the previous test, the results obtained this time are the following:

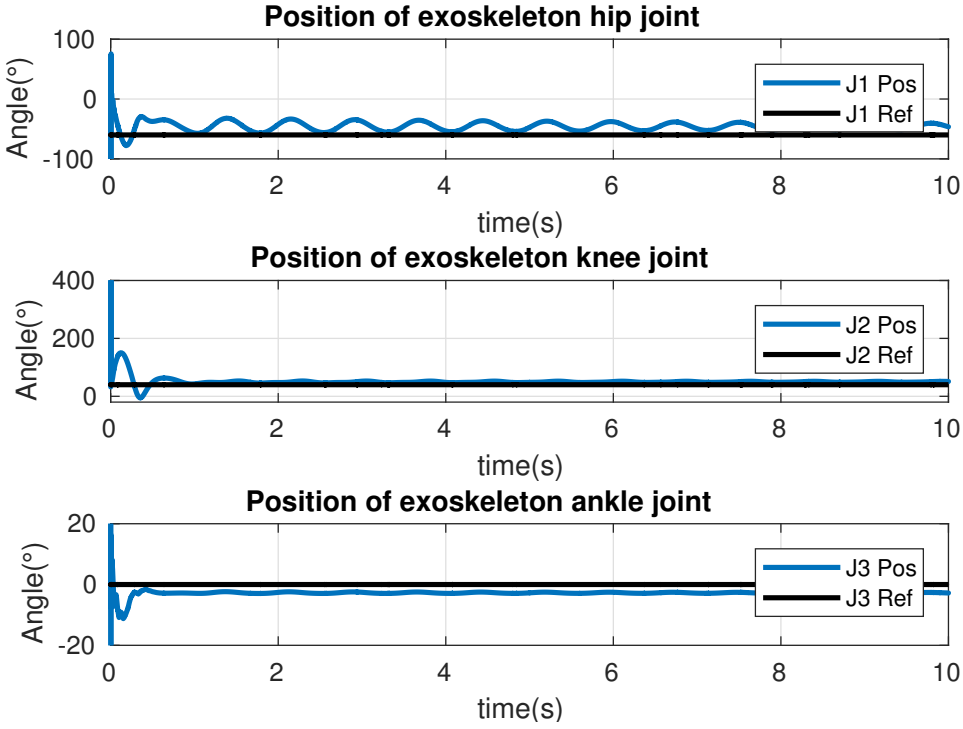


Figure 6.14: Positions of exoskeleton joints respect the reference set.

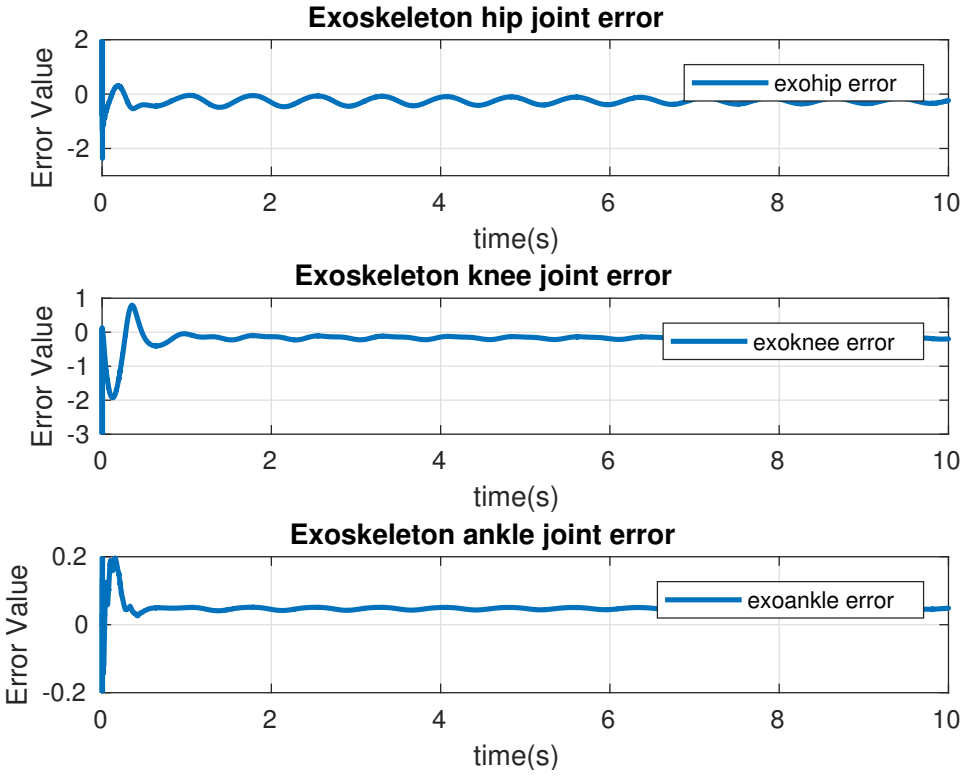
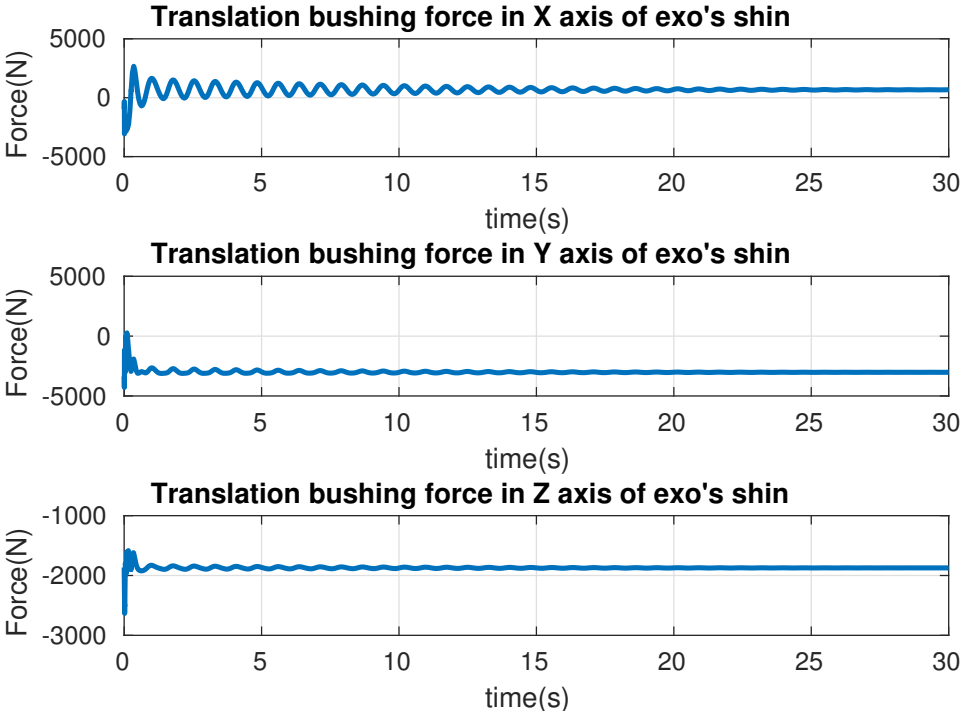


Figure 6.15: Error of exoskeleton joints.

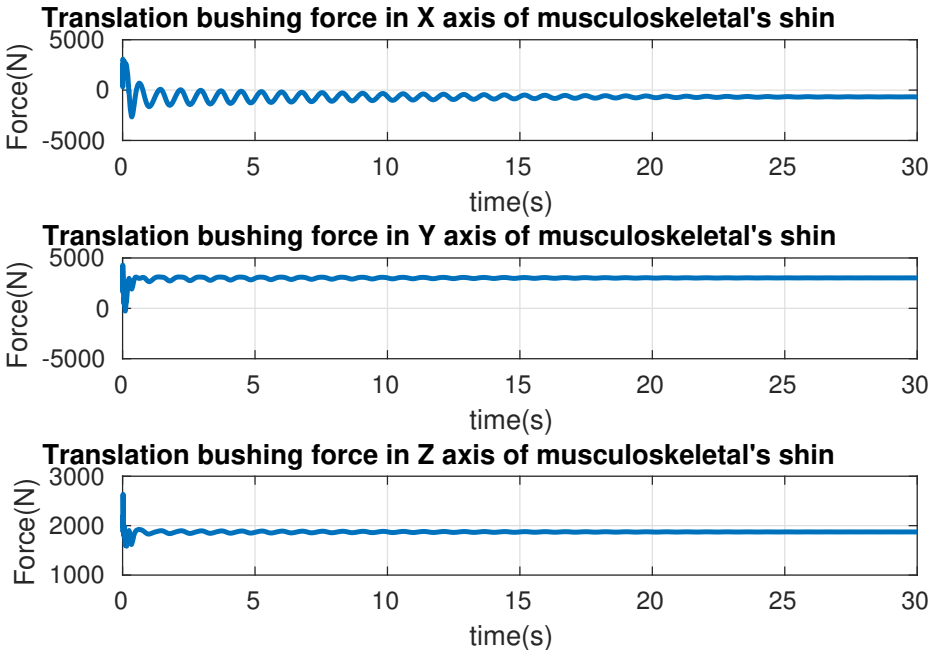
Increasing the K_p parameter has implied that the system becomes faster than the previous test. As a result, the overshoots obtained in this test are higher than test 1. But if it takes a look at error's values of exoskeleton's hip, knee, and ankle are equal to -24.11 % , -17.07 % and 4.71 %, respectively, when time is equal to 8 seconds. Compared with the errors obtained in the test 1, these ones have decreased considerably. But the initial overshoot remains very high.

Translational bushing forces

As in the previous test the translational bushing forces that actuate to exoskeleton's shin is the negative value of the translational bushing forces that actuate to musculoskeletal's shin in each axis — See the figures 6.16 and 6.17.



(a) Exoskeleton's shin bushing translational forces



(b) Musculoskeletal's shin translational forces

Figure 6.16: Exoskeleton's and Musculoskeletal's shins bushing translational forces

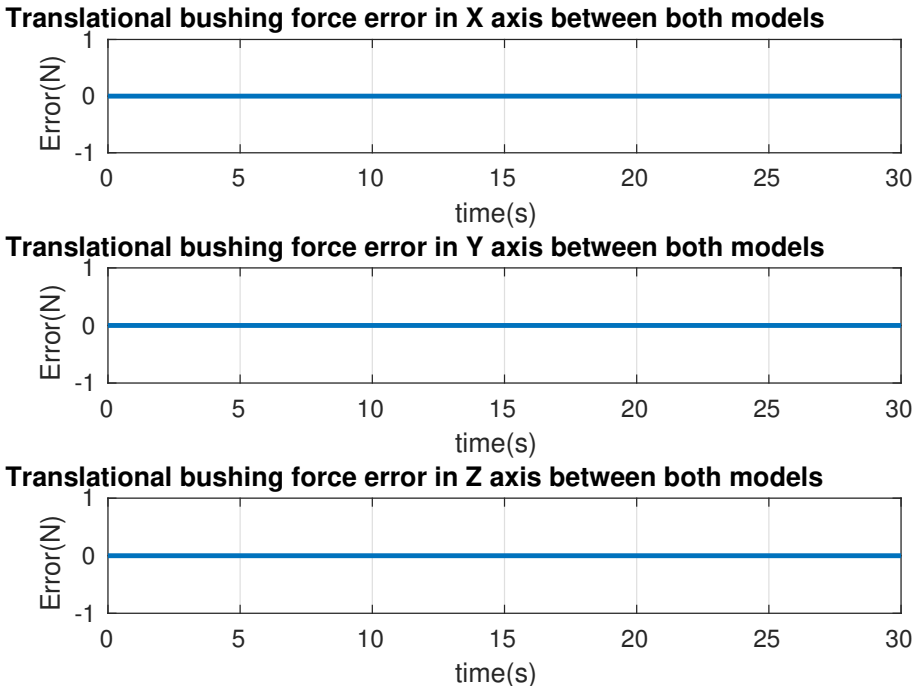
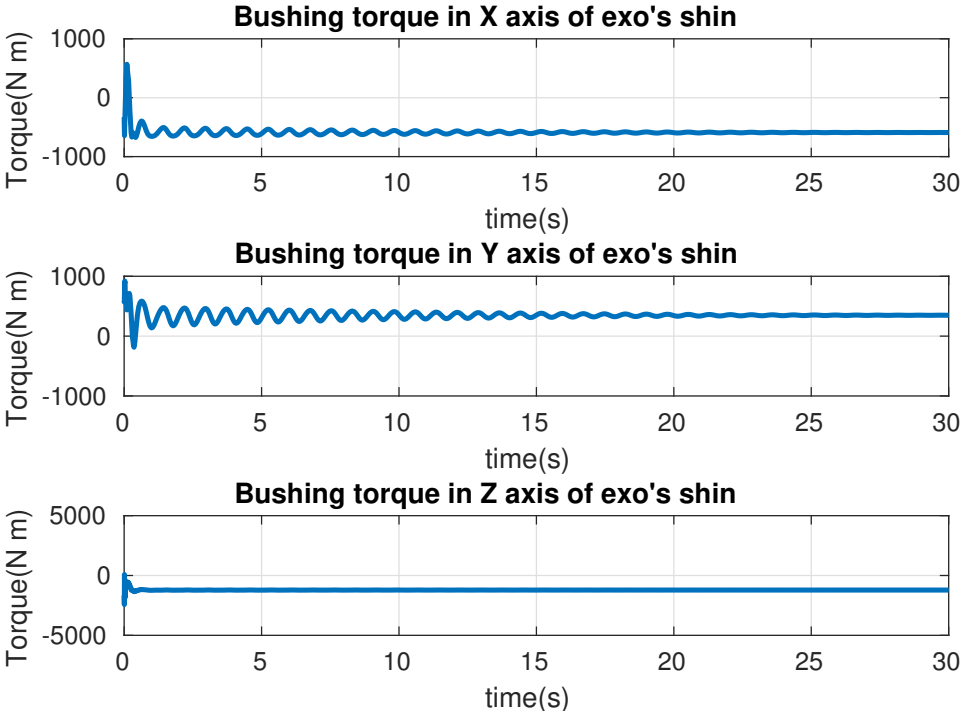


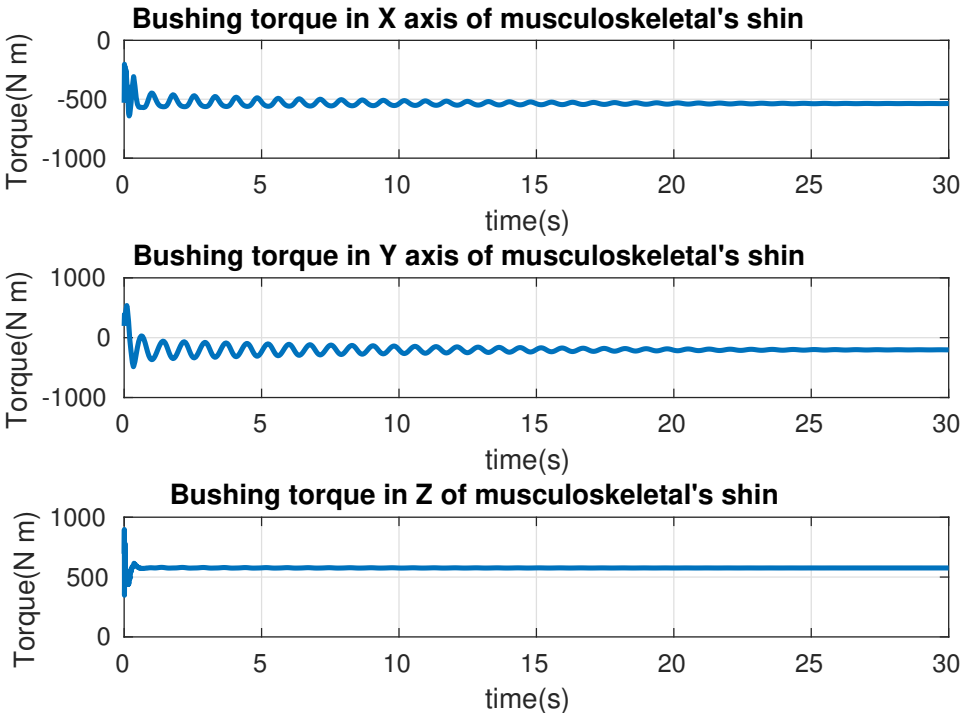
Figure 6.17: Sum of translational forces of exoskeleton's shin and musculoskeletal's shin.

Torque bushing forces

As in the test one, the torques of each model take different values. Thus, the bushing torque of exoskeleton shin's is not the negative of the bushing torque of the musculoskeletal's shin.



(a) Exoskeleton's shin bushing torque forces



(b) Musculoskeletal's shin torque forces

Figure 6.18: Exoskeleton's and Musculoskeletal's shins bushing torque forces

7 Conclusion

The main objective of this project was to test the feasibility of the simulation of a coupled exoskeleton to musculoskeletal OpenSim model. Additionally, the project is also comprised of the many identified objectives that resulted indispensable in order to achieve the main aim of the thesis.

The first one of this sub-objectives was to learn about OpenSim and Simbody. In this matter, all the main steps has been documented with the aim of creating a simple model which has been consisted in a double pendulum model. Not only it has been achieved the creation of a model, but also a controller for it. Additionally, it has been demonstrated that is easy to obtain the dynamics in OpenSim. Thus, this has allowed to compare the dynamics obtained with theoretical equations with the dynamics given by OpenSim. The comparative has shown as a result a little error between both dynamics which is owing to the integrator's resolution.

The second sub-objective was to create an exoskeleton model in OpenSim based on the real model called H1 Exoskeleton. To do it, it was necessary to:

- Implement a visualization of it in the simulation.
- Obtain the dynamics.
- Design a controller.

All of this challenges have been successfully reached. The visualization has been easily implemented by means of imported 3D models generated by an external tool. The dynamics have been obtained using the tools offered by Symbody API. They have been used jointly with a PID control-law in order to control each actuator joint. Even though the design of a optimal controller is not an objective, the results obtained have been satisfactory. However, the controller has to be improved in order to eliminate the overshots in each joint. A possible solution is analyse the response with a mathematical tool as Matlab with the aim of finding optimal PID's parameters.

The achievement of the previous sub-objectives provided the means to develop the main aim of this project. Although coupling the H1 Exoskeleton with the muscoloskeletic "`leg6dof9musc`" in OpenSim has been possible, the utilization of bushing forces to link them has given some problems in the control of it. The main problem arises from the misalignment and placement of the points in which these forces will interact. As it has shown in the results, if the points of each bushing force are not well aligned these forces affect the model's behaviour.

Finally and not less important has been documenting all the previous steps because there is few information related in the implementation of robotic orthosis coupled to musculoskeletal OpenSim models. Therefore, this project will be useful to improve the modeling of H1 Exoskeleton and their controllers, and check if the results obtained in the OpenSim model are similar respect the results of the real one. Moreover, this document aims to become a guide that leads to the creation of new OpenSim projects. In addition, the repository related to this project can be found in GitHub [24].

8 Annexes

This section contains the codes which has been written in the Pendulum control, the exoskeleton implemented in OpenSim, and the exoskeleton coupled to an OpenSim's muscoloskeletal model. Moreover, it is going to find an a library specified that has been written to print information about forces, vectors and export them in files or print it in the console during the execution.

8.1 Pendulum Code

```
/**
* This Project consist in control a double pendulum with OpenSim API.
*
*
*/

#include <OpenSim/OpenSim.h>
using namespace OpenSim;
using namespace std;

class PendulumController : public Controller
{
    OpenSim_DECLARE_CONCRETE_OBJECT(PendulumController, Controller);
public:
    PendulumController(double firstJointAngle, double secondJointAngle) : Controller(),
        r1(firstJointAngle), r2(secondJointAngle)
        {
        }

    double getAngleFromJoints(SimTK::String joint, const SimTK::State &s) const
    {
        double z = _model->getCoordinateSet().get(joint).getValue(s);
        return z;
    }

    double getError(double ref, double aCtAngle) const
    {
        return aCtAngle - ref;
    }

    void getAlphaAndK(double Kp, double Kv, double* getAlphaK) const
    {
        double alpha = (-Kv + sqrt(pow(Kv,2) - 4*Kp)/2);
        double k = 2*Kp/alpha;
        getAlphaK[0] = alpha;
        getAlphaK[1] = k;

    }

    double controlFirstJoint(SimTK::State s,double ref,double k,double alpha, double *getErr=nullptr) const
    {
       double xt_0 = getAngleFromJoints("q1",s); // Get actual Joint's angle
       static double xt_1 = 0;
       double err = getError(ref, xt_0);
       if(getErr != nullptr) // return the error using a pointer, in case of needing it
           *getErr = err;
       double dx = (xt_0 - xt_1)/0.033; // Compute the derivation of the position
       xt_1 = xt_0;
       double torque = alpha*k/2.0*err - (k/2 + alpha)*dx; // Control law
       return torque;
    }

    double controlSecondtJoint(SimTK::State s,double ref,double k,double alpha,double* getErr=nullptr) const
```

```
54      {
55         double xt_0 = getAngleFromJoints("q2",s);
56         static double xt_1 = 0;
57         double err = getError(ref, xt_0);
58         if(getErr != nullptr)
59          *getErr = err;
60         double dx = (xt_0 - xt_1)/0.033;
61         xt_1 = xt_0;
62         double desAcc = alpha*k/2.0*(err) - (k/2 + alpha)*dx;
63         return desAcc;
64      }
65
66      SimTK::Vector gravityCompensation(const SimTK::State &s) const
67      {
68          SimTK::Vector g;
69          _model->getMatterSubsystem().
70                  multiplyBySystemJacobianTranspose(s,
71                                                    _model->getGravityForce().getBodyForces(s),
72                                                    g);
73          return g;
74      }
75
76      void computeControls(const SimTK::State& s, SimTK::Vector &controls) const
77      {
78          double KP1 = 1, KV1 = 10;
79          double KP2 = 0.08, KV2 = 1;
80
81          SimTK::Vector_<double> desAcc(2);
82          desAcc[0] = controlFirstJoint(s,r1,KP1,KV1);
83          desAcc[1] = controlSecondtJoint(s,r2,KP2,KV2);
84
85          SimTK::Vector gravForce(gravityCompensation(s));
86
87          SimTK::Vector desTorque;
88          _model->updMatterSubsystem().multiplyByM(s, desAcc, desTorque);
89
90          //SimTK::Array_<SimTK::MobilizedBodyIndex> onBodyB;
91
92          SimTK::Vector_<double> torque(2);
93          for(int i = 0; i < 2; i++)
94          {
95              torque[i] = (desTorque[i]
96                           + gravForce.get(i)) / _model->getActuators().
97                                                  get(i).getOptimalForce();
98          }
99
100         SimTK::Vector torqueControl_1(1,torque[0]);
101         _model->updActuators().get("firstJointActuator").addInControls(torqueControl_1, controls);
102         SimTK::Vector torqueControl_2(1,torque[1]);
103         _model->updActuators().get("secondJointActuator").addInControls(torqueControl_2, controls);
104
105     }
106
107 private:
108     double r1, r2;
109
110
111 };
112
113 int main()
114 {
115     try
116     {
117         ////////////////////
118         // DEFINE VARIABLES //
119         ////////////////////
120         SimTK::Vec3 gravity(0,-9.8065, 0); // Model's gravity value
121         //SimTK::Vec3 gravity(0); // Model's gravity value
```

```
122         // Time simulation
123         double initTime(0), endTime(40); // In seconds
124 
125         //////////////////
126         // CREATE MODEL //
127         //////////////////
128         Model osimPendulum;
129         osimPendulum.setName("ControlDoublePendulum"); // Set the name of the model
130         osimPendulum.setAuthors("Dídac Coll");         // Set the name of the author
131         osimPendulum.setUseVisualizer(true);
132         osimPendulum.setGravity(gravity);
133 
134         // Get ground
135         OpenSim::Body& ground = osimPendulum.getGroundBody();
136         ground.addDisplayGeometry("checkered_floor.vtp");
137 
138         ///////////////////
139         // CREATE BODIES //
140         ///////////////////
141         // Define Mass properties, dimensions and inertia
142         double cylinderMass = 0.5, cylinderLength = 2.5, cylinderDiameter = 0.30;
143         SimTK::Vec3 cylinderDimensions(cylinderDiameter, cylinderLength, cylinderDiameter);
144         SimTK::Vec3 cylinderMassCenter(0, cylinderLength/2, 0);
145         SimTK::Inertia cylinderInertia = SimTK::Inertia::cylinderAlongY(cylinderDiameter/2.0,
146                                                                         cylinderLength/2.0);
147 
148         OpenSim::Body *firstCylinder = new OpenSim::Body("firstCylinder", cylinderMass, cylinderMassCenter,
149                                     cylinderMass*cylinderInertia);
150         // Set a graphical representation of the cylinder
151         firstCylinder->addDisplayGeometry("cylinder.vtp");
152         // Scale the grahical cylinder(1 m tall, 1 m diameter) to match with body's dimensions
153         GeometrySet& geometrySet = firstCylinder->updDisplayer()->updGeometrySet();
154         DisplayGeometry& newDimCylinder = geometrySet[0];
155         newDimCylinder.setScaleFactors(cylinderDimensions);
156         newDimCylinder.setTransform(Transform(SimTK::Vec3(0.0, cylinderLength/2.0, 0.0)));
157         // Add Sphere in Joint place
158         firstCylinder->addDisplayGeometry("sphere.vtp");
159         // Scale sphere
160         geometrySet[1].setScaleFactors(SimTK::Vec3(0.5));
161 
162 
163 
164         OpenSim::Body *secondCylinder = new OpenSim::Body(*firstCylinder);
165         secondCylinder->setName("secondCylinder");
166 
167         //Create block mass
168         double blockMass = 100.0, blockSideLength = 1.0;
169         SimTK::Vec3 blockMassCenter(0);
170         SimTK::Inertia blockInertia = blockMass*SimTK::Inertia::brick(blockSideLength,
171                                                                       blockSideLength,
172                                                                       blockSideLength);
173 
174         Body *block = new Body("block", blockMass, blockMassCenter, blockInertia);
175         block->addDisplayGeometry("block.vtp");
176         block->updDisplayer()->updGeometrySet()[0].setScaleFactors(SimTK::Vec3(5.0));
177 
178         // Create 1 degree-of-freedom pin joints between ground, first cylinder, and second cyilinder
179         // , weldjoint to attach block to second cyilinder
180         SimTK::Vec3 orientationInGround(0), locationInGround(0),
181                 locationInParent(0.0, cylinderLength, 0.0),
182             orientationInChild(0), locationInChild(0);
183 
184         PinJoint *firstJoint = new PinJoint("firstJoint", ground, locationInGround,
185                                             orientationInGround, *firstCylinder,
186                                             locationInChild, orientationInChild);
187         PinJoint *secondJoint = new PinJoint("secondJoint", *firstCylinder, locationInParent,
188                                              orientationInChild, *secondCylinder,
189                                              locationInChild, orientationInChild);
```

```
190         WeldJoint *endPendulum = new WeldJoint("endPendulum", *secondCylinder,
191                                                locationInParent, orientationInChild,
192                                                *block, locationInChild, orientationInChild);
193
194         // Set range of joints coordinates
195         double range[2] = {-SimTK::Pi*2, SimTK::Pi*2};
196         // Get coordinate set of firstJoint
197         CoordinateSet& fJointCoordinates = firstJoint->upd_CoordinateSet();
198         fJointCoordinates[0].setName("q1");
199         fJointCoordinates[0].setRange(range);
200         // Get coordinate set of secondJoint
201         CoordinateSet& sJointCoordinates = secondJoint->upd_CoordinateSet();
202         sJointCoordinates[0].setName("q2");
203         sJointCoordinates[0].setRange(range);
204
205         // Add Bodies
206         osimPendulum.addBody(firstCylinder);
207         osimPendulum.addBody(secondCylinder);
208         osimPendulum.addBody(block);
209
210         // Creaate Actuators
211         TorqueActuator* firstJointActuator = new TorqueActuator();
212         firstJointActuator->setName("firstJointActuator");
213         firstJointActuator->setBodyA(osimPendulum.getBodySet().get("firstCylinder"));
214         firstJointActuator->setBodyB(osimPendulum.getGroundBody());
215         firstJointActuator->setOptimalForce(20);
216         firstJointActuator->setTorqueIsGlobal(false);
217         TorqueActuator* secondJointActuator = new TorqueActuator();
218         secondJointActuator->setName("secondJointActuator");
219         secondJointActuator->setBodyA(osimPendulum.getBodySet().get("secondCylinder"));
220         secondJointActuator->setBodyB(osimPendulum.getBodySet().get("firstCylinder"));
221         secondJointActuator->setOptimalForce(20);
222         secondJointActuator->setTorqueIsGlobal(false);
223         // Add actuators as forces
224         osimPendulum.addForce(firstJointActuator);
225         osimPendulum.addForce(secondJointActuator);
226
227         // Add a controller
228         PendulumController *pendControl = new PendulumController(0.0, 0.0);
229         pendControl->setActuators(osimPendulum.updActuators());
230         osimPendulum.addController(pendControl);
231
232         // Initialize system
233         SimTK::State &si = osimPendulum.initSystem();
234
235         // Add frictions
236         SimTK::MultibodySystem &system = osimPendulum.updMultibodySystem();
237         SimTK::GeneralForceSubsystem &gForces = osimPendulum.updForceSubsystem();
238         SimTK::Force::MobilityLinearDamper fricFirstJoint (gForces,
239                                                            osimPendulum.updMatterSubsystem().
240                                                            updMobilizedBody(SimTK::MobilizedBodyIndex(1)),
241                                                            SimTK::MobilizerUIndex(0),
242                                                            1);
243
244         SimTK::Force::MobilityLinearDamper fricSecondJoint (gForces,
245                                                             osimPendulum.updMatterSubsystem().
246                                                             updMobilizedBody(SimTK::MobilizedBodyIndex(2)),
247                                                             SimTK::MobilizerUIndex(0),
248                                                             1);
249
250         // Set Up Visualizer
251         osimPendulum.updMatterSubsystem().setShowDefaultGeometry(true);
252         SimTK::Visualizer& viz = osimPendulum.updVisualizer().updSimbodyVisualizer();
253         viz.setBackgroundColor(SimTK::Black);
254
255         si = osimPendulum.initializeState();
256         osimPendulum.printDetailedInfo(si, cout);
257         // Define initial position of each joint
```

```
258         double q1 = -SimTK::Pi;
259         double q2 = -q1 + SimTK::Pi;
260 
261         fJointCoordinates[0].setValue(si,q1);
262         sJointCoordinates[0].setValue(si,q2);
263 
264         osimPendulum.updMultibodySystem().realize(si, SimTK::Stage::Acceleration);
265         // Set integrator
266         SimTK::RungeKuttaMersonIntegrator integrator(osimPendulum.getMultibodySystem());
267         Manager manager(osimPendulum, integrator);
268         manager.setInitialTime(initTime); manager.setFinalTime(endTime);
269         manager.integrate(si);
270 
271     }
272     catch(const OpenSim::Exception& exception)
273     {
274         cout << "Exception occured: " << exception.getMessage() << endl;
275         return 1;
276     }
277 
278 return 0;
279 }
```

8.2 Exoskeleton Code

8.2.1 configuration.h

```
/**
 * This file contain all the #defines that allow to the user configurate
 * the exoskeleton. Some of the features that can be modified easily are
 * disable or enable friction forces, set gravity to 0, and modify the
 * number of exoskeleton parts that acts in the body.
 *
 */
#ifndef CONFIGURATION_H
#define CONFIGURATION_H

/// Uncomment defines to disable the properties

#include <OpenSim/OpenSim.h>


////////////
// MACROS //
////////////

#define degreesToRadians(angleDegrees) (angleDegrees * SimTK::Pi / 180.0)
#define radiansToDegrees(angleRadians) (angleRadians * 180.0 / SimTK::Pi)

/*****************************************************************************/
//////////////////////
// EXO'S BODY PARTS //
//////////////////////

#define ENABLE_EXOFOOT
#define ENABLE_EXOSHIN
#define ENABLE_EXOTHIGH
#define ENABLE_EXOPELVIS

/*****************************************************************************/
////////////////////////////
// EXOSKELETON PARAMETERS //
////////////////////////////

// Set Masses
#define EXOPELVIS_MASS 0.85308774   // kg
#define EXOTHIGH_MASS 0.2138295     // kg
#define EXOSHIN_MASS 0.19567251     // kg
#define EXOFOOT_MASS 0.510668       // kg

// Set Mass Center                              X          Y          Z
#define EXOPELVIS_MASS_CENTER SimTK::Vec3(0.0567373, 0.0433811, -0.00018222) // m
#define EXOTHIGH_MASS_CENTER SimTK::Vec3(0.057235, 0.142446, -0.009)         // m
#define EXOSHIN_MASS_CENTER SimTK::Vec3(-0.009493, 0.13976, 0.00998)         // m
#define EXOFOOT_MASS_CENTER SimTK::Vec3(0.067394, -0.058410, -0.063129)      // m

// Set Inertia
//                                          xx          yy          zz        \
//                                          xy          xz          yz
#define EXOPELVIS_INERTIA SimTK::Inertia(0.33744544, 0.00852229, 0.33785791,    \
                                         -0.02713561, 0.00250198,-0.03328571)
#define EXOTHIGH_INERTIA SimTK::Inertia(0.05890566, 0.00320362, 0.0565293,      \
                                        0.00229535, 0.00058312, -0.01209693)
#define EXOSHIN_INERTIA SimTK::Inertia(0.05890566, 0.00320362, 0.0565293,       \
                                       0.00229535, 0.00058312, -0.01209693)
#define EXOFOOT_INERTIA SimTK::Inertia(0.33744544, 0.00852229, 0.33785791,      \
                                       -0.02713561, 0.00250198,-0.03328571)

// Scale the Body parts
#define EXOPELVIS_SCALE SimTK::Vec3(1.38842608, 1.391052749, 1.3)
```

```
64  #define EXOTHIGH_SCALE SimTK::Vec3(1.3884260802, 1.2178638159, 1.3)
65  #define EXOSHIN_SCALE SimTK::Vec3(1.1884260802, 1.3434147124, 1.3)
66  #define EXOFOOT_SCALE SimTK::Vec3(1.1884260802, 1.2782330916, 1.3)
67
68  // Set Exo hip's ranges min and max in degrees
69  #define EXOHIP_MINRANGE -20 // degrees
70  #define EXOHIP_MAXRANGE 100 // degrees
71
72  // Set Exo Knees ranges min and max in degrees
73  #define EXOKNEE_MINRANGE -5
74  #define EXOKNEE_MAXRANGE 100
75
76  // Set Exo ankles ranges min and max in degrees
77  #define EXOANKLE_MINRANGE -15
78  #define EXOANKLE_MAXRANGE 20
79
80  #define EXOHIP_RANGE {degreesToRadians(EXOHIP_MINRANGE), degreesToRadians(EXOHIP_MAXRANGE)}
81  #define EXOKNEE_RANGE {degreesToRadians(EXOKNEE_MINRANGE), degreesToRadians(EXOKNEE_MAXRANGE)}
82  #define EXOANKLE_RANGE {degreesToRadians(EXOANKLE_MINRANGE), degreesToRadians(EXOANKLE_MAXRANGE)}
83
84  // ExoHip: Range -20 to 100 °, torque -+ 40(Nm)
85  // ExoKnee: Range -5 to 100 °, torque -+ 40(Nm)
86  // ExoAnkle: Range -15 to 20 °, torque -+ 40(Nm)
87
88  /*****************************************************************************/
89  ////////////
90  // FORCES //
91  ////////////
92
93  // Gravity
94  #define ENABLE_GRAVITY // Enable gravity
95  #define GRAVITY_VALUE SimTK::Vec3(0.0, -9.8065, 0.0) // Set the gravity's value
96
97  // Bushing forces
98  #define ENABLE_BUSHINGFORCES // Enable all bushing forces
99
100 // Friction forces
101 #define ENABLE_FRICTIONFORCES // Enable all friction forces
102
103 // Set friction forces value
104 #define EXOHIP_FRICTIONFORCE_VALUE 0
105 #define EXOKNEE_FRICTIONFORCE_VALUE 0
106 #define EXOANKLE_FRICTIONFORCE_VALUE 0
107
108 /*****************************************************************************/
109 ////////////////////////////
110 // INITIAL JOINT POSITIONS //
111 ////////////////////////////
112 #define EXOHIP_INITPOS_DEGREES 0
113 #define EXOKNEE_INITPOS_DEGREES 0
114 #define EXOANKLE_INITPOS_DEGREES 0
115
116 #define EXOHIP_IPOS degreesToRadians(EXOHIP_INITPOS_DEGREES)
117 #define EXOKNEE_IPOS degreesToRadians(EXOKNEE_INITPOS_DEGREES)
118 #define EXOANKLE_IPOS degreesToRadians(EXOANKLE_INITPOS_DEGREES)
119 /*****************************************************************************/
120 ///////////////
121 // ACTUATORS //
122 ///////////////
123
124 // Set Optimal Torque
125 #define OPTIMAL_EXOHIP_TORQUE 20
126 #define OPTIMAL_EXOKNEE_TORQUE 20
127 #define OPTIMAL_EXOANKLE_TORQUE 20
128 /*****************************************************************************/
129 /////////////////
130 // CONTROLLERS //
131 /////////////////
```

```
132
133 #define ENABLE_CONTROLLER // Enable controller
134
135 // Set control parameter values
136 #define KP_EXOHIP 1.25
137 #define KV_EXOHIP 5.0
138 #define KI_EXOHIP 4.0
139
140 #define KP_EXOKNEE 0.75
141 #define KV_EXOKNEE 5.0
142 #define KI_EXOKNEE 4.0
143
144 #define KP_EXOANKLE 0.5
145 #define KV_EXOANKLE 5.0
146 #define KI_EXOANKLE 4.0
147
148 #define K_EXOHIP {KP_EXOHIP,KV_EXOHIP,KI_EXOANKLE}
149 #define K_EXOKNEE {KP_EXOKNEE,KV_EXOKNEE, KI_EXOKNEE}
150 #define K_EXOANKLE {KP_EXOANKLE,KV_EXOANKLE,KI_EXOANKLE}
151 /*********************************************************************************/
152
153
154 #endif //CONFIGURATION_H
```

8.2.2 exoskeletonbody.h

```
1  #ifndef EXOSKELETONBODY_H
2  #define EXOSKELETONBODY_H
3
4  #include <OpenSim/OpenSim.h>
5  #include "configuration.h"
6  #include <OpenSim/Common/PiecewiseConstantFunction.h>
7  using namespace OpenSim;
8  using namespace SimTK;
9
10 class ExoskeletonBody
11 {
12 public:
13     ExoskeletonBody(Model* osimModel, OpenSim::Set<OpenSim::Body>* bodySet,
14                     OpenSim::Set<OpenSim::Joint>* jointSet);
15     void createBodies();
16     void createJoints();
17     void createExoskeleton();
18     void defineExoCoordinates();
19     void defineInitialPosition(SimTK::State& s);
20
21 #ifdef ENABLE_BUSHINGFORCES
22
23     void defineBushingForces(OpenSim::Set<OpenSim::Force>* bushingForces);
24
25 #endif
26
27     void defineFrictionForces();
28     void defineActuators(OpenSim::Set<OpenSim::Actuator>* actuatorSet);
29
30 private:
31     Model* osimModel;
32     OpenSim::Body rFoot, rShin, rThigh, exoHip;
33     OpenSim::Set<OpenSim::Body>* bodySet;
34     OpenSim::Set<OpenSim::Joint>* jointSet;
35
36     // Body variables
37     double rFootMass, rShinMass, rThighMass, hipMass;
38     const Vec3 rFootMassCenter, rShinMassCenter, rThighMassCenter, hipMassCenter;
39     const Vec3 scaleRFoot, scaleRShin, scaleRThigh, sclaeHip;
40     const Inertia rFootInertia, rShinInertia, rThighInertia, hipInertia;
```

```
41
42 };
43
44 #endif // EXOSKELETONBODY_H
```

8.3 exoskeletonbody.cpp

8.3.1 exoskeletonbody.h

```
1  #include "exoskeletonbody.h"
2  /**
3   * @brief ExoskeletonBody::ExoskeletonBody it is the default
4   * constructor.
5   * @param osimModel
6   * @param bodySet
7   * @param jointSet
8   */
9  ExoskeletonBody::ExoskeletonBody(Model* osimModel, OpenSim::Set<OpenSim::Body>* bodySet,
10                                  OpenSim::Set<OpenSim::Joint>* jointSet)
11     : osimModel(osimModel), bodySet(bodySet), jointSet(jointSet),
12     // Initialize Body Mass
13     rFootMass(EXOFOOT_MASS),    // kg
14     rShinMass(EXOSHIN_MASS),    // kg
15     rThighMass(EXOTHIGH_MASS),  // kg
16     hipMass(EXOPELVIS_MASS),    // kg
17     // Initialize Body Mass Centers
18     rFootMassCenter(EXOFOOT_MASS_CENTER),   // m
19     rShinMassCenter(EXOSHIN_MASS_CENTER),   // m
20     rThighMassCenter(EXOTHIGH_MASS_CENTER), // m
21     hipMassCenter(EXOPELVIS_MASS_CENTER),   // m
22     // Initilize Inertia
23     rFootInertia(EXOFOOT_INERTIA),
24     rShinInertia(EXOSHIN_INERTIA),
25     rThighInertia(EXOTHIGH_INERTIA),
26     hipInertia(EXOPELVIS_INERTIA)
27 {
28
29 }
30
31 /**
32  * @brief ExoskeletonBody::createBodies: this method is in charge of creating
33  * all the exoskeleton's bodies. First of all, the bodies has to be created,
34  * then they the joints between them are created and finally the bodies are
35  * added to the OpenSim Model.
36  */
37 void ExoskeletonBody::createBodies()
38 {
39
40 #ifdef ENABLE_EXOPELVIS
41
42     OpenSim::Body hip ("exoHip",
43                        hipMass,
44                        hipMassCenter,
45                        hipInertia);
46     hip.scale(EXOPELVIS_SCALE);
47     hip.addDisplayGeometry("BackExo.obj");
48     exoHip = hip;
49
50     #ifdef ENABLE_EXOTHIGH
51
52     OpenSim::Body rT ("rThigh",
53                       rThighMass,
54                       rThighMassCenter,
55                       rThighInertia);
```

```
56      rT.scale(EXOTHIGH_SCALE);
57      rT.addDisplayGeometry("RightHip.obj");
58      rThigh = rT;
59
60          #ifdef ENABLE_EXOSHIN
61
62          OpenSim::Body rS ("rShin",
63                            rShinMass,
64                            rShinMassCenter,
65                            rShinInertia);
66          rS.scale(EXOSHIN_SCALE);
67          rS.addDisplayGeometry("RightKnee.obj");
68          rShin = rS;
69
70              #ifdef ENABLE_EXOFOOT
71
72               // Create the main exoskeleton bodies
73               OpenSim::Body rF  ("rFoot",          // Body Name
74                                  rFootMass,        // Body Mass
75                                  rFootMassCenter, // Body Mass Center
76                                  rFootInertia      // Body Inertia
77                                  );
78               rF.scale(EXOFOOT_SCALE); // Scale the body
79               rF.addDisplayGeometry("rfoot.obj"); // Add visual geometry to the body
80               rFoot = rF; // Save the object in a private object body
81
82              #endif // ENABLE_EXOFOOT
83
84          #endif // ENABLE_EXOSHIN
85
86      #endif // ENABLE_EXOTHIGH
87
88  #endif // ENABLE_EXOHIP
89
90  }
91
92  /**
93   * @brief ExoskeletonBody::createJoints: thism ethod cal lfirst createBodies
94   * and then all the joints are created. Finally the method adds the joints and
95   * bodies in two different vectors of type Joint and Body, respectively.
96   * Body vector is added in OpenSim Model.
97   */
98  void ExoskeletonBody::createJoints()
99  {
100
101
102 #ifdef ENABLE_EXOPELVIS
103
104     createBodies();
105
106     WeldJoint groundJoint("groundJoint",
107                           osimModel->getGroundBody(),
108                           Vec3(0),
109                           Vec3(0),
110                           exoHip,
111                           Vec3(0),
112                           Vec3(0));
113     jointSet->insert(jointSet->getSize(),groundJoint);
114
115 #ifdef ENABLE_EXOTHIGH
116
117     Vec3 hipLocInParent(0), hipOrInParent(0),
118             hipLocInBody(38e-3, 0.4645, 0), hipOrInBody(0);
119     PinJoint exoHipJ ("exoHipJ",
120                       exoHip,
121                       hipLocInParent,
122                       hipOrInParent,
123                       rThigh,
```

```
                        hipLocInBody,
                        hipOrInBody);
    jointSet->insert(jointSet->getSize(),exoHipJ);

    #ifdef ENABLE_EXOSHIN

        Vec3 kneeLocInParent(0), kneeOrInParent(0),
                kneeLocInBody(72.5e-3, 0.445, 0), kneeOrInBody(0);
        PinJoint exoKneeJ_r ("exoKneeJoint_r",
                             rThigh,
                             kneeLocInParent,
                             kneeOrInParent,
                             rShin,
                             kneeLocInBody,
                             kneeOrInBody);
        jointSet->insert(jointSet->getSize(),exoKneeJ_r);

        #ifdef ENABLE_EXOFOOT

            Vec3 ankLocInParent(0, 0, 0), ankOrInParent(0),
                    ankLocInBody(0, 0, 0), ankOrInBody(0);
            /**
             * @brief exoAnkJ_r: This joint connects the tibia
             * exoskeleton part with the foot
             */
            PinJoint exoAnkJ_r ("exoAnkleJoint_r",
                                rShin,
                                ankLocInParent,
                                ankOrInParent,
                                rFoot,
                                ankLocInBody,
                                ankOrInBody);
            jointSet->insert(jointSet->getSize(),exoAnkJ_r);

        #endif // ENABLE_EXOFOOT

     #endif // ENABLE_EXOSHIN

    #endif // ENABLE_EXOTHIGH

#endif // ENABLE_EXOPELVIS

#ifdef ENABLE_EXOPELVIS

    bodySet->insert(bodySet->getSize(),exoHip);
    osimModel->addBody(&bodySet->get("exoHip"));

    #ifdef ENABLE_EXOTHIGH

        bodySet->insert(bodySet->getSize(),rThigh);
        osimModel->addBody(&bodySet->get("rThigh"));

        #ifdef ENABLE_EXOSHIN

                bodySet->insert(bodySet->getSize(),rShin);
                osimModel->addBody(&bodySet->get("rShin"));

                #ifdef ENABLE_EXOFOOT

                    bodySet->insert(bodySet->getSize(),rFoot);
                    osimModel->addBody(&bodySet->get("rFoot"));

                #endif // ENABLE_EXOFOOT

        #endif //ENABLE_EXOSHIN

    #endif //ENABLE_EXOSHIN

```

```
192 #endif //ENABLE_EXOPELVIS
193
194 }
195
196 /**
197  * @brief ExoskeletonBody::createExoskeleton: calls createJoints method.
198  */
199 void ExoskeletonBody::createExoskeleton()
200 {
201     createJoints();
202 }
203
204 /**
205  * @brief ExoskeletonBody::defineExoCoordinates: define the coordinates names
206  * and ranges.
207  */
208 void ExoskeletonBody::defineExoCoordinates()
209 {
210
211     CoordinateSet& coordinates = osimModel->updCoordinateSet();
212
213 #if defined(ENABLE_EXOPELVIS) && defined(ENABLE_EXOTHIGH)
214
215     double exoHipRange[2] = EXOHIP_RANGE;
216     coordinates[0].setName("exoHipJoint_coord");
217     coordinates[0].setRange(exoHipRange);
218
219     #ifdef ENABLE_EXOSHIN
220
221         double exoKneeRange[2] = EXOKNEE_RANGE;
222         coordinates[1].setName("kneeJoint_r_coord");
223         coordinates[1].setRange(exoKneeRange);
224
225         #ifdef ENABLE_EXOFOOT
226
227             double exoAnkleRange[2] = EXOANKLE_RANGE;
228             coordinates[2].setName("ankleJoint_r_coord");
229             coordinates[2].setRange(exoAnkleRange);
230
231         #endif // ENABLE_EXOFOOT
232
233     #endif // ENABLE_EXOSHIN
234
235 #endif // ENABLE_EXOTHIGH && ENABLE_EXOTHIGH
236
237 }
238 /**
239  * @brief ExoskeletonBody::defineInitialPosition: define the initial position of
240  * the joints.
241  * @param s: the state of the sistem.
242  */
243
244 void ExoskeletonBody::defineInitialPosition(SimTK::State& s)
245 {
246
247 #if defined(ENABLE_EXOPELVIS) && defined(ENABLE_EXOTHIGH)
248
249     CoordinateSet& coordinates = osimModel->updCoordinateSet();
250     coordinates.get("exoHipJoint_coord").setValue(s, EXOHIP_IPOS, true);
251
252     #ifdef ENABLE_EXOSHIN
253
254         coordinates.get("kneeJoint_r_coord").setValue(s, EXOKNEE_IPOS, true);
255
256         #ifdef ENABLE_EXOFOOT
257
258             coordinates.get("ankleJoint_r_coord").setValue(s, EXOANKLE_IPOS,true);
259
```

```
260         #endif // ENABLE_EXOFOOT
261
262     #endif // ENABLE_EXOSHIN
263
264 #endif // ENABLE_EXOTHIGH && ENABLE_EXOTHIGH
265
266 }
267
268
269 #ifdef ENABLE_BUSHINGFORCES // Enables all the bushing forces
270
271 /**
272  * @brief ExoskeletonBody::defineBushingForces: create all the bushing forces are
273  * going to interact between the exoskeleton and the body.
274  * @param bushingForces: vector of type forces
275  */
276
277 void ExoskeletonBody::defineBushingForces(OpenSim::Set<OpenSim::Force>* bushingForces)
278 {
279
280 #if defined(ENABLE_EXOFOOT) && defined(ENABLE_EXOSHIN)
281
282     Vec3 transStiffness(10000), rotStiffness(1000), transDamping(0.1),
283             rotDamping(0);
284     Vec3 tibiaLoc1(-0.01, -0.45, 0.05);
285     Vec3 tibiaOrient(0), ortAnklLoc1(0.04, 0.045, -0.04), ortAnkOrient(0);
286     // Contact forces definitions
287     BushingForce tibiaBushingForce1 ("tibia_r",      // Name of body part
288                                      tibiaLoc1,      // Location
289                                      tibiaOrient,    // Orientation
290                                      "rShin",        // Name of exobody part
291                                      ortAnklLoc1,    // Exobody's location
292                                      ortAnkOrient,   // Exobody orientation
293                                      transStiffness, // Translation stiffness
294                                      rotStiffness,   // Rotation stiffness
295                                      transDamping,   // Translation damping
296                                      rotDamping);    // Orientation damping
297     tibiaBushingForce1.setName("tibiaBushingForce1");// Set the name of bushing force
298
299     Vec3 tibiaLoc2(-0.01, -0.11, 0.05), ortAnklLoc2(0.041, 0.389, -0.038);
300     BushingForce tibiaBushingForce2 ("tibia_r",
301                                      tibiaLoc2,
302                                      tibiaOrient,
303                                      "rShin",
304                                      ortAnklLoc2,
305                                      ortAnkOrient,
306                                      transStiffness,
307                                      rotStiffness,
308                                      transDamping,
309                                      rotDamping);
310     tibiaBushingForce2.setName("tibiaBushingForce2");
311
312     bushingForces->insert(bushingForces->getSize(),tibiaBushingForce1); // Save the force to the vector
313     bushingForces->insert(bushingForces->getSize(),tibiaBushingForce2);
314
315     osimModel->addForce(&bushingForces->get("tibiaBushingForce1")); // Add vector's force to the model
316     osimModel->addForce(&bushingForces->get("tibiaBushingForce2"));
317
318 #ifdef ENABLE_EXOTHIGH
319
320     Vec3 thighLoc1(0.005, -0.4, 0.065), thighOrient(0),
321                     exoThighLoc1(0.025, 0.084, -0.04), exoThighOrient(0);
322     BushingForce thighBushingForce1 ("femur_r",
323                                      thighLoc1,
324                                      thighOrient,
325                                      "rThigh",
326                                      exoThighLoc1,
327                                      exoThighOrient,
```

```
328                                     transStiffness,
329                                     rotStiffness,
330                                     transDamping,
331                                     rotDamping);
332     thighBushingForce1.setName("thighBushingForce1");
333
334     Vec3 thighLoc2(0.005, -0.05, 0.065), exoThighLoc2(0.064, 0.431, -0.0345);
335     BushingForce thighBushingForce2 ("femur_r",
336                                     thighLoc2,
337                                     thighOrient,
338                                     "rThigh",
339                                     exoThighLoc2,
340                                     exoThighOrient,
341                                     transStiffness,
342                                     rotStiffness,
343                                     transDamping,
344                                     rotDamping);
345     thighBushingForce2.setName("thighBushingForce2");
346
347     bushingForces->insert(bushingForces->getSize(),thighBushingForce1);
348     bushingForces->insert(bushingForces->getSize(),thighBushingForce2);
349
350     osimModel->addForce(&bushingForces->get("thighBushingForce1"));
351     osimModel->addForce(&bushingForces->get("thighBushingForce2"));
352
353 #ifdef ENABLE_EXOPELVIS
354
355     Vec3 pelvisLoc1(-0.1, 0.03, 0.12), exoHipLoc1(-0.034, 0.108, -0.059),
356                 pelvisOr(0), exoHipOr(0);
357     BushingForce hipBushingForce1("pelvis",
358                                   pelvisLoc1,
359                                   pelvisOr,
360                                   "exoHip",
361                                   exoHipLoc1,
362                                   exoHipOr,
363                                   transStiffness,
364                                   rotStiffness,
365                                   transDamping,
366                                   rotDamping);
367     hipBushingForce1.setName("hipBushingForce1");
368
369     Vec3 pelvisLoc2(-0.08, 0.02, -0.1), exoHipLoc2(-0.034, 0.108, -0.28);
370     BushingForce hipBushingForce2 ("pelvis",
371                                    pelvisLoc2,
372                                    pelvisOr,
373                                    "exoHip",
374                                    exoHipLoc1,
375                                    exoHipOr,
376                                    transStiffness,
377                                    rotStiffness,
378                                    transDamping,
379                                    rotDamping);
380     hipBushingForce2.setName("hipBushingForce2");
381
382     bushingForces->insert(bushingForces->getSize(),hipBushingForce1);
383     bushingForces->insert(bushingForces->getSize(),hipBushingForce2);
384
385     osimModel->addForce(&bushingForces->get("hipBushingForce1"));
386    // osimModel->addForce(&bushingForces->get("hipBushingForce2"));
387
388 #endif // ENABLE_EXOPELVIS
389
390 #endif // ENABLE_EXOTHIGH
391
392 #endif // ENABLE_EXOFOOT && ENABLE_EXOSHIN
393
394 }
395
```

```
396 #endif // ENABLE_BUSHINGFORCES
397
398 /**
399  * @brief ExoskeletonBody::defineFrictionForces: define joints frictions.
400  */
401
402 void ExoskeletonBody::defineFrictionForces()
403 {
404
405 #if defined(ENABLE_EXOPELVIS) && defined(ENABLE_EXOTHIGH)
406
407     MultibodySystem& system = osimModel->updMultibodySystem();
408     GeneralForceSubsystem& forces = osimModel->updForceSubsystem();
409
410     MobilizedBodyIndex mobIndex(0);
411     const MobilizedBody& mobExoHip =
412             osimModel->getMatterSubsystem().getMobilizedBody(mobIndex);
413     SimTK::Force::MobilityLinearDamper frictionHip (forces,
414                                                     mobExoHip,
415                                                     SimTK::MobilizerUIndex(0),
416                                                     EXOHIP_FRICTIONFORCE_VALUE);
417
418 #ifdef ENABLE_EXOSHIN
419
420     mobIndex++;
421     const MobilizedBody& mobExoKnee =
422             osimModel->getMatterSubsystem().getMobilizedBody(mobIndex);
423
424     SimTK::Force::MobilityLinearDamper frictionKnee (forces,
425                                                      mobExoKnee,
426                                                      SimTK::MobilizerUIndex(0),
427                                                      EXOKNEE_FRICTIONFORCE_VALUE);
428 #ifdef ENABLE_EXOFOOT
429
430     mobIndex++;
431     const MobilizedBody& mobExoAnkle =
432             osimModel->getMatterSubsystem().getMobilizedBody(mobIndex);
433     SimTK::Force::MobilityLinearDamper frictionAnkle(forces,
434                                                      mobExoAnkle,
435                                                      SimTK::MobilizerUIndex(0),
436                                                      EXOANKLE_FRICTIONFORCE_VALUE);
437
438 #endif
439
440 #endif // ENABLE_EXOSHIN
441
442 #endif // ENABLE_EXOPELVIS && ENABLE_EXOTHIGH
443
444 }
445
446 /**
447  * @brief ExoskeletonBody::defineActuators: define joint force actuators
448  * @param actuatorSet
449  */
450
451 void ExoskeletonBody::defineActuators(OpenSim::Set<OpenSim::Actuator>* actuatorSet)
452 {
453 #if defined(ENABLE_EXOPELVIS) && defined(ENABLE_EXOTHIGH)
454
455     TorqueActuator* exoHipActuator_r = new TorqueActuator();
456     exoHipActuator_r->setName("exoHipActuator_r");
457     exoHipActuator_r->setBodyA(bodySet->get("exoHip"));
458     exoHipActuator_r->setBodyB(bodySet->get("rThigh"));
459     exoHipActuator_r->setOptimalForce(OPTIMAL_EXOHIP_TORQUE);
460     exoHipActuator_r->setTorqueIsGlobal(false);
461
462     actuatorSet->insert(actuatorSet->getSize(), *exoHipActuator_r);
463     osimModel->addForce(&actuatorSet->get("exoHipActuator_r"));
```

```
464
465 #ifdef ENABLE_EXOSHIN
466
467     TorqueActuator* exoKneeActuator_r = new TorqueActuator();
468     exoKneeActuator_r->setName("exoKneeActuator_r");
469     exoKneeActuator_r->setBodyA(bodySet->get("rThigh"));
470     exoKneeActuator_r->setBodyB(bodySet->get("rShin"));
471     exoKneeActuator_r->setOptimalForce(OPTIMAL_EXOKNEE_TORQUE);
472     exoKneeActuator_r->setTorqueIsGlobal(false);
473
474     actuatorSet->insert(actuatorSet->getSize(), *exoKneeActuator_r);
475     osimModel->addForce(&actuatorSet->get("exoKneeActuator_r"));
476
477 #ifdef ENABLE_EXOFOOT
478
479     TorqueActuator* exoAnkleActuator_r = new TorqueActuator();
480     exoAnkleActuator_r->setName("exoAnkleActuator_r");
481     exoAnkleActuator_r->setBodyA(bodySet->get("rFoot"));
482     exoAnkleActuator_r->setBodyB(bodySet->get("rShin"));
483     exoAnkleActuator_r->setOptimalForce(OPTIMAL_EXOANKLE_TORQUE);
484     exoAnkleActuator_r->setTorqueIsGlobal(false);
485
486     actuatorSet->insert(actuatorSet->getSize(), *exoAnkleActuator_r);
487     osimModel->addForce(&actuatorSet->get("exoAnkleActuator_r"));
488
489 #endif // ENABLE_EXOFOOT
490
491 #endif // ENABLE_EXOSHIN
492
493 #endif // ENBABLE_EXOHIP && ENABLE_EXOTHIGH
494
495 }
```

8.3.2 positioncontroller.h

```
1  #ifndef POSITIONCONTROLLER_H
2  #define POSITIONCONTROLLER_H
3
4  #include <OpenSim/OpenSim.h>
5  #include <math.h>
6  #include "printOpenSimInformation.h"
7  #include "configuration.h"
8
9  using namespace OpenSim;
10 using namespace SimTK;
11 using namespace std;
12
13 class PositionController : public Controller
14 {
15 OpenSim_DECLARE_CONCRETE_OBJECT(PositionController, Controller);
16
17 public:
18     // This section contains methods that can be called in this controller class.
19     public:
20         /**
21          * Constructor
22          *
23          * @param[in] aModel
24          * Model to be controlled
25          * @param[in] aKp
26          * Position gain by which the position error will be multiplied
27          */
28         PositionController(double hipJointRefAngle, double kneeJointRefAngle,
29                            double ankleJointRefAngle, Model* aModel);
30
31         void getComponentsOfTwoVec3(SimTK::Vec3 initialPoint,
```

```
                                    SimTK::Vec3 finalPoint,
                                    SimTK::Vec3* vectorResult) const;
        double getVectorModule(SimTK::Vec3 &vector) const;
        double computeDesiredAngularAccelerationHipJoint(const SimTK::State &s,
                                                         double* getErr,
                                                         double* getPos) const;
        double computeDesiredAngularAccelerationKneeJoint(const SimTK::State &s,
                                                          double* getErr,
                                                          double* getPos) const;
        double computeDesiredAngularAccelerationAnkleJoint(const SimTK::State &s,
                                                           double* getErr,
                                                           double* getPos) const;

        SimTK::Vector computeDynamicsTorque(const SimTK::State &s,
                                            double* getErrs,
                                            double* getPoses) const;
        /**
        * @brief computeGravityCompensation obtain all the
        * gravity value that affects each joint
        * @param[in] s
        *  actual state
        * @return Return all the values of each body affected by the gravity
         */
        SimTK::Vector computeGravityCompensation(const SimTK::State &s) const;
        SimTK::Vector computeCoriolisCompensation(const SimTK::State &s) const;


        void computeControls(const SimTK::State& s, SimTK::Vector &controls) const;


    // This section contains the member variables of this controller class.
    private:
        /** Position gain for this controller */
        double k_exohip[3], k_exoknee[3], k_exoankle[3];
        /** Velocity gain for this controller */
        double hipJointRefAngle, kneeJointRefAngle, ankleJointRefAngle;

        // Model
        Model* aModel;

        // Mass
        double rFemurMass, rTibiaMass, rPatellaMass, rTalusMass, rCalcnMass,
               rToesMass, rFootMass, rShinMass, rThighMass, exoHipMass;

        double massFemThigh_r, massTibShin_r, massToeFoot_r;

};

#endif
```

8.3.3 positioncontroller.cpp

```
#include "positioncontroller.h"


PositionController::PositionController(double hipJointRefAngle, double kneeJointRefAngle,
                                       double ankleJointRefAngle, Model* aModel) :
    Controller(),k_exohip K_EXOHIP,
    k_exoknee K_EXOKNEE,
    k_exoankle K_EXOANKLE,
    hipJointRefAngle(hipJointRefAngle),
    kneeJointRefAngle(kneeJointRefAngle),
    ankleJointRefAngle(ankleJointRefAngle)
{

}
```

```
15
16
17  /**
18   * @brief PositionController::getComponentsOfTwoVec3: compute the main components between two points
19   * @param initialPoint
20   * @param finalPoint
21   * @param vectorResult
22   */
23  void PositionController::getComponentsOfTwoVec3(SimTK::Vec3 initialPoint, SimTK::Vec3 finalPoint, SimTK::Vec3* vectorResult)
24  {
25
26      SimTK::Vec3 vectResult(finalPoint[0]-initialPoint[0],finalPoint[1]-initialPoint[1],finalPoint[2]-initialPoint[2]);
27      *vectorResult = vectResult;
28
29  }
30
31  double PositionController::getVectorModule(SimTK::Vec3& vector) const
32  {
33      double module = sqrt(pow(vector[0],2) + pow(vector[1],2) + pow(vector[2],2));
34      return module;
35  }
36
37  double PositionController::computeDesiredAngularAccelerationHipJoint(const State &s,
38                                                                        double* getErr = nullptr,
39                                                                        double* getPos = nullptr) const
40  {
41      // Obtain the coordinateset of the joint desired
42      const Coordinate& coord = _model->getCoordinateSet().get("exoHipJoint_coord");
43      double xt = coord.getValue(s); // Actual angle x(t)
44      *getPos = xt;
45      static double errt1 = hipJointRefAngle;
46      double err = hipJointRefAngle - xt; // Error
47      double iErr = (err + errt1)*0.033; // Compute integration
48      double dErr = (err - errt1)/0.033; // Compute the derivative of xt
49      errt1 = err;
50      *getErr = err;
51     // cout << "Error ExoHipJoint = " << err << endl;
52      double  kp = k_exohip[0], kv = k_exohip[1], ki = k_exohip[2];
53      double desAcc = kp*err + kv*dErr + ki*iErr;
54      return desAcc;
55  }
56
57  double PositionController::computeDesiredAngularAccelerationKneeJoint(const State &s,
58                                                                         double* getErr = nullptr,
59                                                                         double* getPos = nullptr) const
60  {
61      const Coordinate& coord = _model->getCoordinateSet().get("kneeJoint_r_coord");
62      double xt = coord.getValue(s);
63      *getPos = xt;
64      static double errt1 = kneeJointRefAngle;
65      double err = kneeJointRefAngle - xt; // Error
66      double iErr = (err + errt1)*0.033; // Compute integration
67      double dErr = (err - errt1)/0.033; // Compute the derivative of xt
68      errt1 = err;
69      *getErr = err;
70      //cout << "Error ExoKneeJoint = " << err << endl;
71      double  kp = k_exoknee[0], kv = k_exoknee[1], ki = k_exoknee[2];
72      double desAcc = kp*err + kv*dErr + ki*iErr;
73      return desAcc;
74  }
75
76
77  double PositionController::computeDesiredAngularAccelerationAnkleJoint(const State &s,
78                                                                          double* getErr = nullptr,
79                                                                          double* getPos = nullptr) const
80  {
81      const Coordinate& coord = _model->getCoordinateSet().get("ankleJoint_r_coord");
82      double xt = coord.getValue(s);
```

```
    *getPos = xt;
    static double xt1 = 0; // Angle position one previous step x(t-1)
    static double errt1 = ankleJointRefAngle;
    double err = ankleJointRefAngle - xt; // Error
    double iErr = (err + errt1)*0.033; // Compute integration
    double dErr = (err - errt1)/0.033; // Compute the derivative of xt
    errt1 = err;
    *getErr = err;
   // cout << "Error ExoAnkleJoint = " << err << endl;
    double  kp = k_exoankle[0], kv = k_exoankle[1], ki = k_exoankle[2];
    double desAcc = kp*err + kv*dErr + ki*iErr;
    return desAcc;
}

SimTK::Vector PositionController::computeDynamicsTorque(const State &s,
                                                         double* getErrs = nullptr,
                                                         double* getPoses = nullptr) const
{
    double desAccExoHip = computeDesiredAngularAccelerationHipJoint(s, &getErrs[0], &getPoses[0]);
    double desAccExoKnee = computeDesiredAngularAccelerationKneeJoint(s, &getErrs[1], &getPoses[1]);
    double desAccExoAnkle = computeDesiredAngularAccelerationAnkleJoint(s, &getErrs[2], &getPoses[2]);
    SimTK::Vector_<double> desAcc(3);
    desAcc[0] = desAccExoHip;
    desAcc[1] = desAccExoKnee;
    desAcc[2] = desAccExoAnkle;
    SimTK::Vector torques;
    _model->getMatterSubsystem().multiplyByM(s, desAcc, torques);

    return torques;
}

SimTK::Vector PositionController::computeGravityCompensation(const SimTK::State &s) const
{
    SimTK::Vector g;
    _model->getMatterSubsystem().
            multiplyBySystemJacobianTranspose(s,
                                              _model->getGravityForce().getBodyForces(s),
                                              g);
    return g;
}

SimTK::Vector PositionController::computeCoriolisCompensation(const SimTK::State &s) const
{
    SimTK::Vector c;
    _model->getMatterSubsystem().
            calcResidualForceIgnoringConstraints(s,
                                                 SimTK::Vector(0),
                                                 SimTK::Vector_<SpatialVec>(0),
                                                 SimTK::Vector(0), c);
    return c;
}



void PositionController::computeControls(const SimTK::State& s, SimTK::Vector &controls) const
{
   double getErrs[3];
   double getPoses[3];
   SimTK::Vector dynTorque(computeDynamicsTorque(s, getErrs, getPoses));
   SimTK::Vector grav(computeGravityCompensation(s));
   SimTK::Vector coriolis(computeCoriolisCompensation(s));



   double nTorqueExoHip = (dynTorque.get(0)
                           + coriolis.get(0)
                           - (grav.get(0))
                           )/_model->getActuators().get("exoHipActuator_r").getOptimalForce();
```

```
151     double nTorqueExoKnee = (dynTorque.get(1)
152                              + coriolis.get(1)
153                              - grav.get(1)
154                              )/_model->getActuators().get("exoKneeActuator_r").getOptimalForce();
155     double nTorqueExoAnkle = (//desAnkleJointTorq
156                               dynTorque.get(2)
157                               + coriolis.get(2)
158                               - grav.get(2)
159                               )/_model->getActuators().get("exoAnkleActuator_r").getOptimalForce();
160 
161 //    cout << "Torque 1 = " << nTorqueExoHip << endl;
162 //    cout << "Torque 2 = " << nTorqueExoKnee << endl;
163 //    cout << "Torque 3 = " << nTorqueExoAnkle << endl;
164 
165     vector<string> nameData = {"Error J1", "Error J2", "Error J3",
166                                "Position J1", "Position J2", "Position J3",
167                                "desTorque1","desTorque2", "desTorque3",
168                                "coriolis1","coriolis2", "coriolis3",
169                                "grav1","grav2", "grav3",
170                                "time"};
171     vector<double> dataToPrint(nameData.size());
172     for(int i = 0; i < 3; i++)
173     {
174         dataToPrint[i] = getErrs[i]; // Save errors
175         dataToPrint[3 + i] = getPoses[i]; // Save positions
176         dataToPrint[6 + i] = dynTorque.get(i); // Save Dynamics Torques
177         dataToPrint[9 + i] = coriolis.get(i); // Save coriolis and centrifugal torques
178         dataToPrint[12 + i] = grav.get(i); // Save
179         dataToPrint[15] = s.getTime();
180     }
181      // Print all data to file
182      static PrintToFile printFile("DataToPlot.txt");
183      printFile.PrintDataToFile(nameData, dataToPrint);
184 
185     Vector torqueControl(1, -nTorqueExoHip);
186     _model->updActuators().get("exoHipActuator_r").addInControls(torqueControl, controls);
187     Vector torqueControl2(1, -nTorqueExoKnee);
188     _model->updActuators().get("exoKneeActuator_r").addInControls(torqueControl2, controls);
189     Vector torqueControl3(1, nTorqueExoAnkle);
190     _model->updActuators().get("exoAnkleActuator_r").addInControls(torqueControl3, controls);
191 }
```

8.3.4 exoMain.cpp

```
1  #include <OpenSim/OpenSim.h>
2  #include "printOpenSimInformation.h"
3  #include "exoskeletonbody.h"
4  #include "configuration.h"
5  #include <OpenSim/Common/PiecewiseConstantFunction.h>
6  #include "positioncontroller.h"
7  #include <iostream>
8  #include <fstream>
9  #include <string>
10 
11 using namespace OpenSim;
12 using namespace SimTK;
13 using namespace std;
14 
15 int main()
16 {
17     double initTime = 0.0, finalTime = 60.0;
18     Model osimModel;
19     osimModel.setName("exowithoutbody");
20 
21     // Set gravity value
22 #ifdef ENABLE_GRAVITY
```

```
23
24      const Vec3 gravity(GRAVITY_VALUE);
25
26  #else
27
28      const Vec3 gravity(0);
29
30  #endif
31
32      osimModel.setGravity(gravity);
33
34      // Set visualization mode
35      osimModel.setUseVisualizer(true);
36      OpenSim::Set<OpenSim::Body> bodySet;
37      OpenSim::Set<OpenSim::Joint> jointSet;
38      OpenSim::Set<OpenSim::Force> bushingForces;
39      OpenSim::Set<OpenSim::Actuator> actuatorSet;
40
41      ExoskeletonBody exoskeletonBody(&osimModel, &bodySet, &jointSet);
42      exoskeletonBody.createExoskeleton();
43      exoskeletonBody.defineExoCoordinates();
44      exoskeletonBody.defineActuators(&actuatorSet);
45
46  #ifdef ENABLE_CONTROLLER
47
48      double hipRef = degreesToRadians(50),
49             kneeRef = degreesToRadians(-60) - hipRef/1.5,
50             ankleRef = degreesToRadians(20);
51
52      PositionController* posController = new PositionController(hipRef,
53                                                                 kneeRef,
54                                                                 ankleRef,
55                                                                 &osimModel);
56      posController->setActuators(osimModel.updActuators());
57      osimModel.addController(posController);
58
59  #endif
60
61
62      ////////////////////////////
63      // PERFORM A SIMULATION //
64      ////////////////////////////
65      // Initialize System
66      State& si = osimModel.initSystem();
67      osimModel.printDetailedInfo(si, std::cout);
68      exoskeletonBody.defineFrictionForces();
69      osimModel.print("exowithoutbody.osim");
70
71      si = osimModel.initializeState();
72      exoskeletonBody.defineInitialPosition(si);
73
74      ////////////////////////
75      // DISABLE ACTUACTORS //
76      ////////////////////////
77      osimModel.updMatterSubsystem().setShowDefaultGeometry(true);
78      Visualizer& viz = osimModel.updVisualizer().updSimbodyVisualizer();
79      viz.setBackgroundColor(Black);
80
81      // Simulate
82      RungeKuttaMersonIntegrator integrator (osimModel.updMultibodySystem());
83      Manager manager(osimModel, integrator);
84      manager.setInitialTime(initTime); manager.setFinalTime(finalTime);
85      manager.integrate(si);
86
87      return 0;
88  }
```

8.4 Code of Exoskeleton Coupled to Musculoskeletal OpenSim Model

8.4.1 configuration.h

```
/**
 * This file contain all the #defines that allow to the user configurate
 * the exoskeleton. Some of the features that can be modified easily are
 * disable or enable friction forces, set gravity to 0, and modify the
 * number of exoskeleton parts that acts in the body.
 *
 */
#ifndef CONFIGURATION_H
#define CONFIGURATION_H

/// Uncomment defines to disable the properties

#include <OpenSim/OpenSim.h>


////////////
// MACROS //
////////////

#define degreesToRadians(angleDegrees) (angleDegrees * SimTK::Pi / 180.0)
#define radiansToDegrees(angleRadians) (angleRadians * 180.0 / SimTK::Pi)


//////////////////////
// EXO'S BODY PARTS //
//////////////////////

#define ENABLE_EXOFOOT
#define ENABLE_EXOSHIN
#define ENABLE_EXOTHIGH
#define ENABLE_EXOHIP


////////////
// FORCES //
////////////

// Gravity
#define ENABLE_GRAVITY // Enable gravity
#define GRAVITY_VALUE SimTK::Vec3(0.0, -9.8065, 0.0) // Set the gravity's value

// Bushing forces
#define ENABLE_BUSHINGFORCES // Enable all bushing forces

// Friction forces
#define ENABLE_FRICTIONFORCES // Enable all friction forces

// Set friction forces value
#define EXOHIP_FRICTIONFORCE_VALUE 5
#define EXOKNEE_FRICTIONFORCE_VALUE 5
#define EXOANKLE_FRICTIONFORCE_VALUE 5

///////////////
// ACTUATORS //
///////////////

// Set Optimal Torque
#define OPTIMAL_EXOHIP_TORQUE 20
#define OPTIMAL_EXOKNEE_TORQUE 20
#define OPTIMAL_EXOANKLE_TORQUE 20

/////////////////
// CONTROLLERS //
```

```
64  //////////////////
65
66  #define ENABLE_CONTROLLER // Enable controller
67
68      // Set control parameter values
69      #define KP_EXOHIP 50.0
70      #define KV_EXOHIP 40.0
71      #define KI_EXOHIP 120.0
72
73      #define KP_EXOKNEE 50.0
74      #define KV_EXOKNEE 40.0
75      #define KI_EXOKNEE 120.0
76
77      #define KP_EXOANKLE 5.0
78      #define KV_EXOANKLE 10.0
79      #define KI_EXOANKLE 10.0
80
81      #define K_EXOHIP {KP_EXOHIP,KV_EXOHIP,KI_EXOANKLE}
82      #define K_EXOKNEE {KP_EXOKNEE,KV_EXOKNEE, KI_EXOKNEE}
83      #define K_EXOANKLE {KP_EXOANKLE,KV_EXOANKLE,KI_EXOANKLE}
84
85  ////////////////////////////
86  // EXOSKELETON PARAMETERS //
87  ////////////////////////////
88
89  // Set Exo hip's ranges min and max in degrees
90  #define EXOHIP_MINRANGE -20 // degrees
91  #define EXOHIP_MAXRANGE 100 // degrees
92
93  // Set Exo Knees ranges min and max in degrees
94  #define EXOKNEE_MINRANGE -5
95  #define EXOKNEE_MAXRANGE 100
96
97  // Set Exo ankles ranges min and max in degrees
98  #define EXOANKLE_MINRANGE -15
99  #define EXOANKLE_MAXRANGE 20
100
101 #define EXOHIP_RANGE {degreesToRadians(EXOHIP_MINRANGE), degreesToRadians(EXOHIP_MAXRANGE)}
102 #define EXOKNEE_RANGE {degreesToRadians(EXOKNEE_MINRANGE), degreesToRadians(EXOKNEE_MAXRANGE)}
103 #define EXOANKLE_RANGE {degreesToRadians(EXOANKLE_MINRANGE), degreesToRadians(EXOANKLE_MAXRANGE)}
104
105 // ExoHip: Range -20 to 100 °, torque -+ 40(Nm)
106 // ExoKnee: Range -5 to 100 °, torque -+ 40(Nm)
107 // ExoAnkle: Range -15 to 20 °, torque -+ 40(Nm)
108
109 // Initial Position of each joint
110 #define EXOANKLE_INITIALPOS degreesToRadians(0)
111 #define EXOKNEE_INITIALPOS degreesToRadians(37.864)
112 #define EXOHIP_INITIALPOS degreesToRadians(-16.689)
113
114 #endif //CONFIGURATION_H
```

8.4.2 exoskeletonbody.h

```
1  #ifndef EXOSKELETONBODY_H
2  #define EXOSKELETONBODY_H
3
4  #include <OpenSim/OpenSim.h>
5  #include "configuration.h"
6  #include <OpenSim/Common/PiecewiseConstantFunction.h>
7  using namespace OpenSim;
8  using namespace SimTK;
9
10 class ExoskeletonBody
11 {
12 public:
```

```
13      ExoskeletonBody(Model* osimModel, OpenSim::Set<OpenSim::Body>* bodySet,
14                      OpenSim::Set<OpenSim::Joint>* jointSet);
15      void createBodies();
16      void createJoints();
17      void createExoskeleton();
18      void defineExoCoordinates();
19      void defineInitialPosition(SimTK::State& s);
20
21  #ifdef ENABLE_BUSHINGFORCES
22
23      void defineBushingForces(OpenSim::Set<OpenSim::Force>* bushingForces);
24
25  #endif
26
27      void defineFrictionForces();
28      void defineActuators(OpenSim::Set<OpenSim::Actuator>* actuatorSet);
29
30  private:
31      Model* osimModel;
32      OpenSim::Body rFoot, rShin, rThigh, exoHip;
33      OpenSim::Set<OpenSim::Body>* bodySet;
34      OpenSim::Set<OpenSim::Joint>* jointSet;
35
36      // Body variables
37      double rFootMass, rShinMass, rThighMass, hipMass;
38      const Vec3 rFootMassCenter, rShinMassCenter, rThighMassCenter, hipMassCenter;
39      const Vec3 scaleRFoot, scaleRShin, scaleRThigh, sclaeHip;
40      const Inertia rFootInertia, rShinInertia, rThighInertia, hipInertia;
41
42  };
43
44  #endif // EXOSKELETONBODY_H
```

8.5 exoskeletonbody.cpp

8.5.1 exoskeletonbody.h

```
1   #include "exoskeletonbody.h"
2   /**
3    * @brief ExoskeletonBody::ExoskeletonBody it is the default
4    * constructor.
5    * @param osimModel
6    * @param bodySet
7    * @param jointSet
8    */
9   ExoskeletonBody::ExoskeletonBody(Model* osimModel, OpenSim::Set<OpenSim::Body>* bodySet,
10                                   OpenSim::Set<OpenSim::Joint>* jointSet)
11      : osimModel(osimModel), bodySet(bodySet), jointSet(jointSet),
12      // Initialize Body Mass
13      rFootMass(0.510668),    // kg
14      rShinMass(0.19567251),  // kg
15      rThighMass(0.2138295),  // kg
16      hipMass(0.85308774),   // kg
17      // Initialize Body Mass Centers
18  //    rFootMassCenter(0.0282501, -0.523347, 0.116142),  // m
19  //    rShinMassCenter(0.0649924, -0.80779, 0.0808896),  // m
20  //    rThighMassCenter(0.0805624, -0.191386, 0.117911), // m
21  //    hipMassCenter(0.0567373, 0.0433811, -0.00018222),// m
22      rFootMassCenter(0.067394, -0.058410, -0.063129),  // m
23      rShinMassCenter(-0.009493, 0.13976, 0.00998),  // m
24      rThighMassCenter(0.057235, 0.142446, -0.009), // m
25      hipMassCenter(0.0567373, 0.0433811, -0.00018222),// m
26      // Initilize Inertia
27      //                xx          yy          zz
```

```
28      //                xy          xz          yz
29      rFootInertia(0.33744544, 0.00852229, 0.33785791,
30                   -0.02713561, 0.00250198,-0.03328571),
31      rShinInertia(0.05890566, 0.00320362, 0.0565293,
32                    0.00229535, 0.00058312, -0.01209693),
33      rThighInertia(0.01556931, 0.004679, 0.01394983,
34                     -0.00314352, 0.00203429, -0.00518597),
35      hipInertia(0.01254384, 0.01446753, 0.00695939,
36                  0.00217966, 0.00011335, -4.28e-006)
37  {
38
39  }
40
41  /**
42   * @brief ExoskeletonBody::createBodies: this method is in charge of creating
43   * all the exoskeleton's bodies. First of all, the bodies has to be created,
44   * then they the joints between them are created and finally the bodies are
45   * added to the OpenSim Model.
46   */
47  void ExoskeletonBody::createBodies()
48  {
49
50  #ifdef ENABLE_EXOFOOT
51
52      // Create the main exoskeleton bodies
53      OpenSim::Body rF  ("rFoot",         // Body Name
54                         rFootMass,       // Body Mass
55                         rFootMassCenter, // Body Mass Center
56                         rFootInertia     // Body Inertia
57                         );
58      rF.scale(Vec3(1.1884260802,1.2782330916,1.3)); // Scale the body
59      rF.addDisplayGeometry("rfoot.obj"); // Add visual geometry to the body
60      rFoot = rF; // Save the object in a private object body
61
62      #ifdef ENABLE_EXOSHIN
63
64      OpenSim::Body rS ("rShin",
65                        rShinMass,
66                        rShinMassCenter,
67                        rShinInertia);
68      rS.scale(Vec3(1.1884260802,1.3434147124,1.3));
69      rS.addDisplayGeometry("RightKnee.obj");
70      rShin = rS;
71
72          #ifdef ENABLE_EXOTHIGH
73
74               OpenSim::Body rT ("rThigh",
75                                 rThighMass,
76                                 rThighMassCenter,
77                                 rThighInertia);
78               rT.scale(Vec3(1.3884260802,1.2178638159,1.3));
79               rT.addDisplayGeometry("RightHip.obj");
80               rThigh = rT;
81
82              #ifdef ENABLE_EXOHIP
83
84                  OpenSim::Body hip ("exoHip",
85                                     hipMass,
86                                     hipMassCenter,
87                                     hipInertia);
88                  hip.scale(Vec3(1.38842608, 1.391052749, 1.3));
89                  hip.addDisplayGeometry("BackExo.obj");
90                  exoHip = hip;
91
92              #endif // ENABLE_EXOHIP
93
94          #endif // ENABLE_EXOTHIGH
95
```

```
96      #endif // ENABLE_EXOSHIN
97
98  #endif // ENABLE_EXOFOOT
99
100 }
101
102 /**
103  * @brief ExoskeletonBody::createJoints: thism ethod cal lfirst createBodies
104  * and then all the joints are created. Finally the method adds the joints and
105  * bodies in two different vectors of type Joint and Body, respectively.
106  * Body vector is added in OpenSim Model.
107  */
108 void ExoskeletonBody::createJoints()
109 {
110
111 #ifdef ENABLE_EXOFOOT
112
113     createBodies();
114
115     // Create a weldjoint behind skeleton and exoskeleton foot.
116     const Vec3 exoBodyLocInParent(-0.15, 0.07, 0.1), exoBodyOrInParent(0),
117             exoBodyLocInBody(0, 0, 0), exoBodyOrInBody(0);
118     WeldJoint exoBodyJ_r ("exoBodyJoint_r",
119                         osimModel->updBodySet().get(7), // toe_r
120                         exoBodyLocInParent,
121                         exoBodyOrInParent,
122                         rFoot,
123                         exoBodyLocInBody,
124                         exoBodyOrInBody);
125     jointSet->insert(jointSet->getSize(),exoBodyJ_r);
126     bodySet->insert(bodySet->getSize(),rFoot);
127
128     #ifdef ENABLE_EXOSHIN
129
130         Vec3 ankLocInParent(0, 0, 0), ankOrInParent(0),
131                 ankLocInBody(0, 0, 0), ankOrInBody(0);
132         /**
133          * @brief exoAnkJ_r: This joint connects the tibia
134          * exoskeleton part with the foot
135          */
136         PinJoint exoAnkJ_r ("exoAnkleJoint_r",
137                             rFoot,
138                             ankLocInParent,
139                             ankOrInParent,
140                             rShin,
141                             ankLocInBody,
142                             ankOrInBody);
143         jointSet->insert(jointSet->getSize(),exoAnkJ_r);
144         bodySet->insert(bodySet->getSize(),rShin);
145
146         #ifdef ENABLE_EXOTHIGH
147
148             Vec3 kneeLocInParent(72.5e-3, 0.445, 0), kneeOrInParent(0),
149                     kneeLocInBody(0), kneeOrInBody(0);
150             PinJoint exoKneeJ_r ("exoKneeJoint_r",
151                                 rShin,
152                                 kneeLocInParent,
153                                 kneeOrInParent,
154                                 rThigh,
155                                 kneeLocInBody,
156                                 kneeOrInBody);
157             jointSet->insert(jointSet->getSize(),exoKneeJ_r);
158             bodySet->insert(bodySet->getSize(),rThigh);
159
160             #ifdef ENABLE_EXOHIP
161
162                 Vec3 hipLocInParent(38e-3, 0.4645, 0), hipOrInParent(0),
163                         hipLocInBody(0), hipOrInBody(0);
```

```
164                 PinJoint exoHipJ ("exoHipJ",
165                                   rThigh,
166                                   hipLocInParent,
167                                   hipOrInParent,
168                                   exoHip,
169                                   hipLocInBody,
170                                   hipOrInBody);
171                 jointSet->insert(jointSet->getSize(),exoHipJ);
172                 bodySet->insert(bodySet->getSize(),exoHip);
173 
174             #endif // ENABLE_EXOHIP
175 
176         #endif // ENABLE_EXOTHIGH
177 
178     #endif // ENABLE_EXOSHIN
179 
180 #endif // ENABLE_EXOFOOT
181 
182 #ifdef ENABLE_EXOFOOT
183 
184         osimModel->addBody(&bodySet->get("rFoot"));
185 
186     #ifdef ENABLE_EXOSHIN
187 
188             osimModel->addBody(&bodySet->get("rShin"));
189 
190         #ifdef ENABLE_EXOTHIGH
191 
192                 osimModel->addBody(&bodySet->get("rThigh"));
193 
194                 #ifdef ENABLE_EXOHIP
195 
196                     osimModel->addBody(&bodySet->get("exoHip"));
197 
198                 #endif
199 
200         #endif //ENABLE_EXOTHIGH
201 
202     #endif //ENABLE_EXOSHIN
203 
204 #endif //ENABLE_EXOFOOT
205 
206 
207 
208 }
209 
210 /**
211  * @brief ExoskeletonBody::createExoskeleton: calls createJoints method.
212  */
213 void ExoskeletonBody::createExoskeleton()
214 {
215     createJoints();
216 }
217 
218 /**
219  * @brief ExoskeletonBody::defineExoCoordinates: define the coordinates names
220  * and ranges.
221  */
222 void ExoskeletonBody::defineExoCoordinates()
223 {
224 
225     CoordinateSet& coordinates = osimModel->updCoordinateSet();
226 
227 #if defined(ENABLE_EXOFOOT) && defined(ENABLE_EXOSHIN)
228 
229     double exoHipRange[2] = EXOHIP_RANGE;
230     coordinates[4].setName("ankleJoint_r_coord");
231     coordinates[4].setRange(exoHipRange);
```

```
232
233     #ifdef ENABLE_EXOTHIGH
234
235         double exoKneeRange[2] = EXOKNEE_RANGE;
236         coordinates[5].setName("kneeJoint_r_coord");
237         coordinates[5].setRange(exoKneeRange);
238
239         #ifdef ENABLE_EXOHIP
240
241             double exoAnkleRange[2] = EXOANKLE_RANGE;
242             coordinates[6].setName("exoHipJoint_coord");
243             coordinates[6].setRange(exoAnkleRange);
244
245         #endif
246
247     #endif // ENABLE_EXOTHIGH
248
249 #endif // ENABLE_EXOSHIN
250
251 }
252 /**
253  * @brief ExoskeletonBody::defineInitialPosition: define the initial position of
254  * the joints.
255  * @param s: the state of the sistem.
256  */
257
258 void ExoskeletonBody::defineInitialPosition(SimTK::State& s)
259 {
260
261 #if defined(ENABLE_EXOFOOT) && defined(ENABLE_EXOSHIN)
262
263     CoordinateSet& coordinates = osimModel->updCoordinateSet();
264     coordinates.get("ankleJoint_r_coord").setValue(s, EXOANKLE_INITIALPOS, true);
265
266     #ifdef ENABLE_EXOTHIGH
267
268         coordinates.get("kneeJoint_r_coord").setValue(s, EXOKNEE_INITIALPOS, true);
269
270         #ifdef ENABLE_EXOHIP
271
272             coordinates.get("exoHipJoint_coord").setValue(s, EXOHIP_INITIALPOS, true);
273
274         #endif // ENABLE_EXOHIP
275
276     #endif // ENABLE_EXOTHIGH
277
278 #endif // ENABLE_EXOFOOT && ENABLE_EXOSHIN
279
280 }
281
282
283 #ifdef ENABLE_BUSHINGFORCES // Enables all the bushing forces
284
285 /**
286  * @brief ExoskeletonBody::defineBushingForces: create all the bushing forces are
287  * going to interact between the exoskeleton and the body.
288  * @param bushingForces: vector of type forces
289  */
290
291 void ExoskeletonBody::defineBushingForces(OpenSim::Set<OpenSim::Force>* bushingForces)
292 {
293
294 #if defined(ENABLE_EXOFOOT) && defined(ENABLE_EXOSHIN)
295
296     Vec3 transStiffness(100000), rotStiffness(10000), transDamping(0.1),
297             rotDamping(0);
298     Vec3 tibiaLoc1(-0.01, -0.45, 0.05);
299     Vec3 tibiaOrient(0), ortAnklLoc1(0.04, 0.045, -0.04), ortAnkOrient(0);
```

```
300     // Contact forces definitions
301     BushingForce tibiaBushingForce1 ("tibia_r",      // Name of body part
302                                      tibiaLoc1,      // Location
303                                      tibiaOrient,    // Orientation
304                                      "rShin",        // Name of exobody part
305                                      ortAnklLoc1,    // Exobody's location
306                                      ortAnkOrient,   // Exobody orientation
307                                      transStiffness, // Translation stiffness
308                                      rotStiffness,   // Rotation stiffness
309                                      transDamping,   // Translation damping
310                                      rotDamping);    // Orientation damping
311     tibiaBushingForce1.setName("tibiaBushingForce1");// Set the name of bushing force
312
313     Vec3 tibiaLoc2(-0.01, -0.11, 0.05), ortAnklLoc2(0.041, 0.389, -0.038);
314     BushingForce tibiaBushingForce2 ("tibia_r",
315                                      tibiaLoc2,
316                                      tibiaOrient,
317                                      "rShin",
318                                      ortAnklLoc2,
319                                      ortAnkOrient,
320                                      transStiffness,
321                                      rotStiffness,
322                                      transDamping,
323                                      rotDamping);
324     tibiaBushingForce2.setName("tibiaBushingForce2");
325
326     bushingForces->insert(bushingForces->getSize(),tibiaBushingForce1); // Save the force to the vector
327     bushingForces->insert(bushingForces->getSize(),tibiaBushingForce2);
328
329     osimModel->addForce(&bushingForces->get("tibiaBushingForce1")); // Add vector's force to the model
330     osimModel->addForce(&bushingForces->get("tibiaBushingForce2"));
331
332 #ifdef ENABLE_EXOTHIGH
333
334     Vec3 thighLoc1(0.005, -0.4, 0.065), thighOrient(0),
335                    exoThighLoc1(0.025, 0.084, -0.04), exoThighOrient(0);
336     BushingForce thighBushingForce1 ("femur_r",
337                                      thighLoc1,
338                                      thighOrient,
339                                      "rThigh",
340                                      exoThighLoc1,
341                                      exoThighOrient,
342                                      transStiffness,
343                                      rotStiffness,
344                                      transDamping,
345                                      rotDamping);
346     thighBushingForce1.setName("thighBushingForce1");
347
348     Vec3 thighLoc2(0.005, -0.05, 0.065), exoThighLoc2(0.064, 0.431, -0.0345);
349     BushingForce thighBushingForce2 ("femur_r",
350                                      thighLoc2,
351                                      thighOrient,
352                                      "rThigh",
353                                      exoThighLoc2,
354                                      exoThighOrient,
355                                      transStiffness,
356                                      rotStiffness,
357                                      transDamping,
358                                      rotDamping);
359     thighBushingForce2.setName("thighBushingForce2");
360
361     bushingForces->insert(bushingForces->getSize(),thighBushingForce1);
362     bushingForces->insert(bushingForces->getSize(),thighBushingForce2);
363
364     osimModel->addForce(&bushingForces->get("thighBushingForce1"));
365     osimModel->addForce(&bushingForces->get("thighBushingForce2"));
366
367 #ifdef ENABLE_EXOHIP
```

```
368
369     Vec3 pelvisLoc1(-0.1, 0.03, 0.12), exoHipLoc1(-0.034, 0.108, -0.059),
370                 pelvisOr(0), exoHipOr(0);
371     BushingForce hipBushingForce1("pelvis",
372                                   pelvisLoc1,
373                                   pelvisOr,
374                                   "exoHip",
375                                   exoHipLoc1,
376                                   exoHipOr,
377                                   transStiffness,
378                                   rotStiffness,
379                                   transDamping,
380                                   rotDamping);
381     hipBushingForce1.setName("hipBushingForce1");
382
383     Vec3 pelvisLoc2(-0.08, 0.02, -0.1), exoHipLoc2(-0.034, 0.108, -0.28);
384     BushingForce hipBushingForce2 ("pelvis",
385                                    pelvisLoc2,
386                                    pelvisOr,
387                                    "exoHip",
388                                    exoHipLoc1,
389                                    exoHipOr,
390                                    transStiffness,
391                                    rotStiffness,
392                                    transDamping,
393                                    rotDamping);
394     hipBushingForce2.setName("hipBushingForce2");
395
396     bushingForces->insert(bushingForces->getSize(),hipBushingForce1);
397     bushingForces->insert(bushingForces->getSize(),hipBushingForce2);
398
399     osimModel->addForce(&bushingForces->get("hipBushingForce1"));
400    // osimModel->addForce(&bushingForces->get("hipBushingForce2"));
401
402 #endif // ENABLE_EXOHIP
403
404 #endif // ENABLE_EXOTHIGH
405
406 #endif // ENABLE_EXOFOOT && ENABLE_EXOSHIN
407
408 }
409
410 #endif // ENABLE_BUSHINGFORCES
411
412 /**
413  * @brief ExoskeletonBody::defineFrictionForces: define joints frictions.
414  */
415
416 void ExoskeletonBody::defineFrictionForces()
417 {
418
419 #if defined(ENABLE_EXOFOOT) && defined(ENABLE_EXOSHIN)
420
421     MobilizedBodyIndex mobIndex(8);
422     const MobilizedBody& mobExoAnkle =
423             osimModel->getMatterSubsystem().getMobilizedBody(mobIndex);
424
425     MultibodySystem& system = osimModel->updMultibodySystem();
426     GeneralForceSubsystem& forces = osimModel->updForceSubsystem();
427
428     SimTK::Force::MobilityLinearDamper frictionAnkle(forces,
429                                                      mobExoAnkle,
430                                                      SimTK::MobilizerUIndex(0),
431                                                      EXOANKLE_FRICTIONFORCE_VALUE);
432
433 #ifdef ENABLE_EXOTHIGH
434
435     mobIndex++;
```

```
436     const MobilizedBody& mobExoKnee =
437             osimModel->getMatterSubsystem().getMobilizedBody(mobIndex);
438
439     SimTK::Force::MobilityLinearDamper frictionKnee (forces,
440                                                      mobExoKnee,
441                                                      SimTK::MobilizerUIndex(0),
442                                                      EXOKNEE_FRICTIONFORCE_VALUE);
443 #ifdef ENABLE_EXOHIP
444
445     mobIndex++;
446     const MobilizedBody& mobExoHip =
447             osimModel->getMatterSubsystem().getMobilizedBody(mobIndex);
448     SimTK::Force::MobilityLinearDamper frictionHip (forces,
449                                                     mobExoHip,
450                                                     SimTK::MobilizerUIndex(0),
451                                                     EXOHIP_FRICTIONFORCE_VALUE);
452
453 #endif
454
455 #endif // ENABLE_EXOTHIGH
456
457 #endif // ENABLE_EXOFOOT && ENABLE_EXOSHIN
458
459 }
460
461 /**
462  * @brief ExoskeletonBody::defineActuators: define joint force actuators
463  * @param actuatorSet
464  */
465
466 void ExoskeletonBody::defineActuators(OpenSim::Set<OpenSim::Actuator>* actuatorSet)
467 {
468 #if (defined(ENABLE_EXOHIP) && defined(ENABLE_EXOTHIGH))
469
470     TorqueActuator* exoHipActuator_r = new TorqueActuator();
471     exoHipActuator_r->setName("exoHipActuator_r");
472     exoHipActuator_r->setBodyA(bodySet->get("exoHip"));
473     exoHipActuator_r->setBodyB(bodySet->get("rThigh"));
474     exoHipActuator_r->setOptimalForce(OPTIMAL_EXOHIP_TORQUE);
475     exoHipActuator_r->setTorqueIsGlobal(false);
476
477     actuatorSet->insert(actuatorSet->getSize(), *exoHipActuator_r);
478     osimModel->addForce(&actuatorSet->get("exoHipActuator_r"));
479
480 #endif
481
482 #if (defined(ENABLE_EXOTHIGH) && defined(ENABLE_EXOSHIN))
483
484     TorqueActuator* exoKneeActuator_r = new TorqueActuator();
485     exoKneeActuator_r->setName("exoKneeActuator_r");
486     exoKneeActuator_r->setBodyA(bodySet->get("rThigh"));
487     exoKneeActuator_r->setBodyB(bodySet->get("rShin"));
488     exoKneeActuator_r->setOptimalForce(OPTIMAL_EXOKNEE_TORQUE);
489     exoKneeActuator_r->setTorqueIsGlobal(false);
490
491     actuatorSet->insert(actuatorSet->getSize(), *exoKneeActuator_r);
492     osimModel->addForce(&actuatorSet->get("exoKneeActuator_r"));
493
494 #endif
495
496 #if defined(ENABLE_EXOFOOT) && defined(ENABLE_EXOSHIN)
497
498     TorqueActuator* exoAnkleActuator_r = new TorqueActuator();
499     exoAnkleActuator_r->setName("exoAnkleActuator_r");
500     exoAnkleActuator_r->setBodyA(bodySet->get("rFoot"));
501     exoAnkleActuator_r->setBodyB(bodySet->get("rShin"));
502     exoAnkleActuator_r->setOptimalForce(OPTIMAL_EXOANKLE_TORQUE);
503     exoAnkleActuator_r->setTorqueIsGlobal(false);
```

```
504
505     actuatorSet->insert(actuatorSet->getSize(), *exoAnkleActuator_r);
506     osimModel->addForce(&actuatorSet->get("exoAnkleActuator_r"));
507
508 #endif
509 }
```

8.5.2 positioncontroller.h

```
1  #ifndef POSITIONCONTROLLER_H
2  #define POSITIONCONTROLLER_H
3
4  #include<OpenSim/OpenSim.h>
5  #include <math.h>
6  #include "printOpenSimInformation.h"
7  #include "configuration.h"
8
9  using namespace OpenSim;
10 using namespace SimTK;
11 using namespace std;
12
13 class PositionController : public Controller
14 {
15 OpenSim_DECLARE_CONCRETE_OBJECT(PositionController, Controller);
16
17 public:
18     // This section contains methods that can be called in this controller class.
19     public:
20         /**
21          * Constructor
22          *
23          * @param[in] aModel
24          * Model to be controlled
25          * @param[in] aKp
26          * Position gain by which the position error will be multiplied
27          */
28         PositionController(double hipJointRefAngle, double kneeJointRefAngle,
29                            double ankleJointRefAngle, Model* aModel);
30
31         void getComponentsOfTwoVec3(SimTK::Vec3 initialPoint, SimTK::Vec3 finalPoint, SimTK::Vec3* vectorResult) const;
32         double getVectorModule(SimTK::Vec3 &vector) const;
33         double computeDesiredAngularAccelerationHipJoint(const SimTK::State &s,
34                                                          double* getErr,
35                                                          double* getPos) const;
36         double computeDesiredAngularAccelerationKneeJoint(const SimTK::State &s,
37                                                           double* getErr,
38                                                           double* getPos) const;
39         double computeDesiredAngularAccelerationAnkleJoint(const SimTK::State &s,
40                                                            double* getErr,
41                                                            double* getPos) const;
42         SimTK::Vector computeDynamicsTorque(const SimTK::State &s,
43                                             double* getErrs,
44                                             double* getPoses) const;
45         /**
46         * @brief computeGravityCompensation obtain all the
47         * gravity value that affects each joint
48         * @param[in] s
49         *  actual state
50         * @return Return all the values of each body affected by the gravity
51          */
52         SimTK::Vector computeGravityCompensation(const SimTK::State &s) const;
53         SimTK::Vector computeCoriolisCompensation(const SimTK::State &s) const;
54
55
56         void computeControls(const SimTK::State& s, SimTK::Vector &controls) const;
57
```

```
58
59      // This section contains the member variables of this controller class.
60      private:
61          /** Position gain for this controller */
62          double k_exohip[3], k_exoknee[3], k_exoankle[3];
63          /** Velocity gain for this controller */
64          double hipJointRefAngle, kneeJointRefAngle, ankleJointRefAngle;
65
66          // Model
67          Model* aModel;
68
69          // Mass
70          double rFemurMass, rTibiaMass, rPatellaMass, rTalusMass, rCalcnMass,
71                 rToesMass, rFootMass, rShinMass, rThighMass, exoHipMass;
72
73          double massFemThigh_r, massTibShin_r, massToeFoot_r;
74
75  };
76
77  #endif
```

8.5.3 positioncontroller.cpp

```
1   #include "positioncontroller.h"
2
3
4   PositionController::PositionController(double hipJointRefAngle, double kneeJointRefAngle,
5                                          double ankleJointRefAngle, Model* aModel) :
6       Controller(),k_exohip K_EXOHIP,
7       k_exoknee K_EXOKNEE,
8       k_exoankle K_EXOANKLE,
9       hipJointRefAngle(hipJointRefAngle),
10      kneeJointRefAngle(kneeJointRefAngle),
11      ankleJointRefAngle(ankleJointRefAngle)
12  {
13
14  }
15
16
17  /**
18   * @brief PositionController::getComponentsOfTwoVec3: compute the main components between two points
19   * @param initialPoint
20   * @param finalPoint
21   * @param vectorResult
22   */
23  void PositionController::getComponentsOfTwoVec3(SimTK::Vec3 initialPoint, SimTK::Vec3 finalPoint, SimTK::Vec3* vectorResult) const
24  {
25
26      SimTK::Vec3 vectResult(finalPoint[0]-initialPoint[0],finalPoint[1]-initialPoint[1],finalPoint[2]-initialPoint[2]);
27      *vectorResult = vectResult;
28
29  }
30
31  double PositionController::getVectorModule(SimTK::Vec3& vector) const
32  {
33      double module = sqrt(pow(vector[0],2) + pow(vector[1],2) + pow(vector[2],2));
34      return module;
35  }
36
37  double PositionController::computeDesiredAngularAccelerationHipJoint(const State &s,
38                                                                        double* getErr = nullptr,
39                                                                        double* getPos = nullptr) const
40  {
41      // Obtain the coordinateset of the joint desired
42      const Coordinate& coord = _model->getCoordinateSet().get("exoHipJoint_coord");
43      double xt = coord.getValue(s); // Actual angle x(t)
```

```
44      double tsample = 0.033; // Time sample
45      if(getPos != nullptr)
46         *getPos = xt; // Save position
47      static double errt1 = hipJointRefAngle;
48      double err = hipJointRefAngle - xt; // Error
49      double iErr = (err + errt1)*tsample; // Compute integration
50      double dErr = (err - errt1)/tsample; // Compute the derivative of xt
51      errt1 = err;
52      if(getErr != nullptr)
53          *getErr = err;
54      // cout << "Error ExoHipJoint = " << err << endl;
55      double  kp = k_exohip[0], kv = k_exohip[1], ki = k_exohip[2];
56      double desAcc = kp*err + kv*dErr + ki*iErr;
57      return desAcc;
58  }
59
60  double PositionController::computeDesiredAngularAccelerationKneeJoint(const State &s,
61                                                                         double* getErr = nullptr,
62                                                                         double* getPos = nullptr) const
63  {
64      const Coordinate& coord = _model->getCoordinateSet().get("kneeJoint_r_coord");
65      double xt = coord.getValue(s);
66      double tsample = 0.033;
67      if(getPos != nullptr)
68         *getPos = xt;
69      static double errt1 = kneeJointRefAngle;
70      double err = kneeJointRefAngle - xt; // Error
71      double iErr = (err + errt1)*tsample; // Compute integration
72      double dErr = (err - errt1)/tsample; // Compute the derivative of xt
73      errt1 = err;
74      if(getErr != nullptr)
75          *getErr = err;
76      //cout << "Error ExoKneeJoint = " << err << endl;
77      double  kp = k_exoknee[0], kv = k_exoknee[1], ki = k_exoknee[2];
78      double desAcc = kp*err + kv*dErr + ki*iErr;
79      return desAcc;
80  }
81
82  double PositionController::computeDesiredAngularAccelerationAnkleJoint(const State &s,
83                                                                          double* getErr = nullptr,
84                                                                          double* getPos = nullptr) const
85  {
86       const Coordinate& coord = _model->getCoordinateSet().get("ankleJoint_r_coord");
87       double xt = coord.getValue(s);
88       double tsample = 0.033;
89       if(getPos != nullptr)
90          *getPos = xt;
91       static double xt1 = 0; // Angle position one previous step x(t-1)
92       static double errt1 = ankleJointRefAngle;
93       double err = ankleJointRefAngle - xt; // Error
94       double iErr = (err + errt1)*tsample; // Compute integration
95       double dErr = (err - errt1)/tsample; // Compute the derivative of xt
96       errt1 = err;
97       if(getErr != nullptr)
98           *getErr = err;
99       // cout << "Error ExoAnkleJoint = " << err << endl;
100      double  kp = k_exoankle[0], kv = k_exoankle[1], ki = k_exoankle[2];
101      double desAcc = kp*err + kv*dErr + ki*iErr;
102      return desAcc;
103 }
104
105 SimTK::Vector PositionController::computeDynamicsTorque(const State &s,
106                                                          double* getErrs = nullptr,
107                                                          double* getPoses = nullptr) const
108 {
109     double desAccExoHip = computeDesiredAngularAccelerationHipJoint(s, &getErrs[0], &getPoses[0]);
110     double desAccExoKnee = computeDesiredAngularAccelerationKneeJoint(s, &getErrs[1], &getPoses[1]);
111     double desAccExoAnkle = computeDesiredAngularAccelerationAnkleJoint(s, &getErrs[2], &getPoses[2]);
```

```
    SimTK::Vector_<double> desAcc(7);
    desAcc[0] = desAccExoHip; desAcc[1] = desAccExoKnee;
    desAcc[2] = desAccExoKnee; desAcc[3] = desAccExoAnkle;
    desAcc[4] = desAccExoAnkle; desAcc[5] = desAccExoKnee;
    desAcc[6] = desAccExoHip;

    SimTK::Vector torques;
    _model->getMatterSubsystem().multiplyByM(s, desAcc, torques);

    return torques;
}

SimTK::Vector PositionController::computeGravityCompensation(const SimTK::State &s) const
{
    SimTK::Vector g;
    _model->getMatterSubsystem().
            multiplyBySystemJacobianTranspose(s,
                                              _model->getGravityForce().getBodyForces(s),
                                              g);
    return g;
}

SimTK::Vector PositionController::computeCoriolisCompensation(const SimTK::State &s) const
{
    SimTK::Vector c;
    _model->getMatterSubsystem().
            calcResidualForceIgnoringConstraints(s,
                                                 SimTK::Vector(0),
                                                 SimTK::Vector_<SpatialVec>(0),
                                                 SimTK::Vector(0), c);
    return c;
}



void PositionController::computeControls(const SimTK::State& s, SimTK::Vector &controls) const
{
   double getErrs[3];
   double getPoses[3];
   SimTK::Vector dynTorque(computeDynamicsTorque(s, getErrs, getPoses));
   SimTK::Vector grav(computeGravityCompensation(s));
   SimTK::Vector coriolis(computeCoriolisCompensation(s));

   double nTorqueExoHip = (dynTorque.get(0) + dynTorque.get(6)
                           + coriolis.get(0) + coriolis.get(6)
                           -(grav.get(0) + grav.get(6))
                           )/_model->getActuators().get("exoHipActuator_r").getOptimalForce();
   double nTorqueExoKnee = (dynTorque.get(1) + dynTorque.get(2) + dynTorque.get(5)
                            + coriolis.get(1) + coriolis.get(2) + coriolis.get(5)
                            -(grav.get(1) + grav.get(2) + grav.get(3))
                            )/_model->getActuators().get("exoKneeActuator_r").getOptimalForce();
   double nTorqueExoAnkle = (dynTorque.get(3) + dynTorque.get(4)
                             + coriolis.get(3) + coriolis.get(4)
                            - (grav.get(3) + grav.get(4))
                             )/_model->getActuators().get("exoAnkleActuator_r").getOptimalForce();
//   for(int i = 0; i < 3; i++)
//       cout << "Error " << i+1 << " = " << getErrs[i] << endl;
   vector<string> nameData = {"Error J1", "Error J2", "Error J3",
                              "Position J1", "Position J2", "Position J3",
                              "desTorque1","desTorque2", "desTorque3",
                              "coriolis1","coriolis2", "coriolis3",
                              "grav1","grav2", "grav3",
                              "time"};
   vector<double> dataToPrint(nameData.size());
   for(int i = 0; i < 3; i++)
   {
       dataToPrint[i] = getErrs[i]; // Save errors
       dataToPrint[3 + i] = getPoses[i]; // Save positions
```

```
        dataToPrint[6 + i] = dynTorque.get(i); // Save Dynamics Torques
        dataToPrint[9 + i] = coriolis.get(i); // Save coriolis and centrifugal torques
        dataToPrint[12 + i] = grav.get(i); // Save
        dataToPrint[15] = s.getTime();
    }
   // Print all data to file
   static PrintToFile printFile("DataToPlot.txt");
   printFile.PrintDataToFile(nameData, dataToPrint);
   Vector torqueControl(1, nTorqueExoHip);
   _model->updActuators().get("exoHipActuator_r").addInControls(torqueControl, controls);
   Vector torqueControl2(1, nTorqueExoKnee);
   _model->updActuators().get("exoKneeActuator_r").addInControls(torqueControl2, controls);
   Vector torqueControl3(1, -nTorqueExoAnkle);
   _model->updActuators().get("exoAnkleActuator_r").addInControls(torqueControl3, controls);
}
```

8.5.4 exoskeletonProgram.cpp

```
// Explain All the code

#include <OpenSim/OpenSim.h>
#include "printOpenSimInformation.h"
#include "exoskeletonbody.h"
#include "configuration.h"
#include <OpenSim/Common/PiecewiseConstantFunction.h>
#include "positioncontroller.h"
#include <iostream>
#include <fstream>
#include <string>

using namespace OpenSim;
using namespace SimTK;
using namespace std;


int main()
{
    try
    {
        double initTime = 0.0, finalTime = 30.0;
        Model osimModel("skeleton.osim");
        osimModel.setName("exoskeleton");

        // Set gravity value
#ifdef ENABLE_GRAVITY

        const Vec3 gravity(GRAVITY_VALUE);

#else

        const Vec3 gravity(0,0,0);

#endif

        osimModel.setGravity(gravity);

        // Set visualization mode
        osimModel.setUseVisualizer(true);
        OpenSim::Set<OpenSim::Body> bodySet;
        OpenSim::Set<OpenSim::Joint> jointSet;
        OpenSim::Set<OpenSim::Force> bushingForces;
        OpenSim::Set<OpenSim::Actuator> actuatorSet;

        const Vec3 locInParent(0), orInParent(0), locInBody(0), orInBody(0);
        WeldJoint* pelvisGround = new WeldJoint("pelvisground",
                                                osimModel.getGroundBody(),
```

```
                                                locInParent,
                                                orInParent,
                                                osimModel.updBodySet().get(1),
                                                locInBody,
                                                orInBody);

        ExoskeletonBody exoskeletonBody(&osimModel, &bodySet, &jointSet);
        exoskeletonBody.createExoskeleton();
        exoskeletonBody.defineExoCoordinates();

#ifdef ENABLE_BUSHINGFORCES

        exoskeletonBody.defineBushingForces(&bushingForces);

#endif

        exoskeletonBody.defineActuators(&actuatorSet);

#ifdef ENABLE_CONTROLLER

        double hipRef = degreesToRadians(-60),
               kneeRef = degreesToRadians(40),
               ankleRef = degreesToRadians(0);

        PositionController* posController = new PositionController(hipRef,
                                                                    kneeRef,
                                                                    ankleRef,
                                                                    &osimModel);
        posController->setActuators(osimModel.updActuators());
        osimModel.addController(posController);

#endif

        ////////////////////////////
        // PERFORM A SIMULATION //
        ////////////////////////////
        // Initialize System
        State& si = osimModel.initSystem();
        osimModel.printDetailedInfo(si, std::cout);

        ///////////////////////
        exoskeletonBody.defineFrictionForces();


        ForceReporter *forces = new ForceReporter(&osimModel);
        osimModel.updAnalysisSet().adoptAndAppend(forces);
        //PrintForceToFile printBushingForce("forces.txt","tibiaBushingForce1.tibia_r",&osimModel);

        si = osimModel.initializeState();
        exoskeletonBody.defineInitialPosition(si);
        //osimModel.getMultibodySystem().realize(si,Stage::Acceleration);
        ////////////////////////
        // DISABLE ACTUACTORS //
        ////////////////////////
        osimModel.updMatterSubsystem().setShowDefaultGeometry(true);
        Visualizer& viz = osimModel.updVisualizer().updSimbodyVisualizer();
        viz.setBackgroundColor(Black);


        // Simulate
        RungeKuttaMersonIntegrator integrator (osimModel.updMultibodySystem());
        Manager manager(osimModel, integrator);
        manager.setInitialTime(initTime); manager.setFinalTime(finalTime);
        manager.integrate(si);
        // Save results
        forces->getForceStorage().print("allForces.csv");
        //printBushingForce.printForces();

```

```
117         }
118         catch(OpenSim::Exception ex)
119         {
120             std::cout << ex.getMessage() << std::endl;
121             return 1;
122         }
123 
124         return 0;
125     }
```

8.6 Library Print OpenSim Information

This part shows the code created in order to print some information in the console or in files.

8.6.1 printOpenSimInformation.h

```
#ifndef PRINTOPENSIMINFORMATION_H
    #define PRINTOPENSIMINFORMATION_H

#include <OpenSim/OpenSim.h>
#include <iostream>
#include <array>
#include <vector>
#include <fstream>
#include <string>

using namespace OpenSim;
using namespace SimTK;
using namespace std;


namespace OpenSim {
    class PrintInformation {
    public:
        PrintInformation(std::ostream& stream) : stream(stream), endl('\n'){}
        PrintInformation() : stream(cout), endl('\n') {}

        void printVector(const SimTK::Vector &vec);
        void printVec3(const SimTK::Vec3& vec);
        void printInertia(const SimTK::Inertia& inert);
        void printSpatialMat(const SimTK::SpatialMat &spatialMat, bool matlab = true);

    protected:
        std::ostream& stream;
        const char endl;
    };


    class PrintToFile  {
    public:
        PrintToFile(const char* fileName) : fileName(fileName),
            file(fileName, std::ios::out | std::ios::trunc)
        {
        }
        void PrintDataToFile(std::vector<string> nameData, std::vector<double> data);
        void PrintDataToFile(string* nameData, double* data, int size);

    protected:

        const char* fileName;
        ofstream file;


    };

    class PrintModelInformation : public PrintInformation
    {
    public:
        PrintModelInformation(const Model& osimModel, std::ostream& stream) :
        PrintInformation(stream),aModel(osimModel)
        {

        }

        void printCoordinateNames();
```

```
60
61      private:
62          const Model& aModel;
63
64      };
65
66      class PrintJointInformation : public PrintInformation
67      {
68      public:
69          PrintJointInformation(const JointSet& joints, std::ostream& stream) :
70              PrintInformation(stream), joints(joints) {
71              joints.getNames(arrayOfJointNames);
72          }
73
74          void printJointNames();
75          void printJointParentLocation(int index);
76          void printJointChildLocation(int index);
77          void printJointParentOrientation(int index);
78          void printAllJointParentLocations();
79          void printAllJointParentOrientation();
80
81      private:
82          JointSet joints;
83          Array<std::string> arrayOfJointNames;
84      };
85
86      class PrintForceToFile : public PrintToFile {
87      public:
88          PrintForceToFile(const char* fileName, string forceName, Model* osimModel)
89              : PrintToFile(fileName), forceName(forceName), osimModel(osimModel),
90                  forceSet(osimModel->getForceSet()) {
91
92              forceReporter = new ForceReporter(osimModel);
93              osimModel->addAnalysis(forceReporter);
94
95
96          }
97
98          void getForces();
99          void getForceNames();
100         void printForces();
101
102     private:
103         string forceName;
104         Model* osimModel;
105         Array<Array<double> > rData;
106         ForceReporter* forceReporter;
107         ForceSet forceSet;
108         Array<string> forceNames;
109
110     };
111
112     class AngularAccelerationReporter : public PeriodicEventReporter {
113     public:
114         AngularAccelerationReporter(const MultibodySystem& system, const MobilizedBody& mobod,
115                     const string moBodyName, Real reportInerval)
116             : PeriodicEventReporter(reportInerval), system(system), mobod(mobod), moBodyName(moBodyName) {}
117         void handleEvent(const State& state) const {
118             system.realize(state, Stage::Acceleration);
119             Vec3 angAcc = mobod.getBodyAngularAcceleration(state);
120             cout << "Angular Acceleration of " << moBodyName << " = { x = " << angAcc[0] << ", y = "
121                  << angAcc[1] << ", z = " << angAcc[2] << " }" << endl;
122         }
123
124     private:
125         const MultibodySystem& system;
126         const MobilizedBody& mobod;
127         const string moBodyName;
```

```
128     };
129
130     class PositionReporter : public PeriodicEventReporter {
131     public:
132         PositionReporter(const MultibodySystem &system, const MobilizedBody &mobod,
133                          const string moBodyName, Real reportInterval)
134             :   PeriodicEventReporter(reportInterval), system(system), mobod(mobod), moBodyName(moBodyName) {}
135         void handleEvent(const State &state) const {
136             system.realize(state, Stage::Position);
137             Vec3 pos = mobod.getBodyOriginLocation(state);
138             cout << "Position of "<< moBodyName << "= { x = " << pos[0] << ", y = " << pos[1]
139                  << ", z = "<< pos[2] << " }" << endl;
140         }
141     private:
142         const MultibodySystem& system;
143         const MobilizedBody& mobod;
144         const string moBodyName;
145     };
146
147     class VelocityReporter : public PeriodicEventReporter {
148     public:
149         VelocityReporter(const MultibodySystem &system, const MobilizedBody &mobod,
150                          const string moBodyName, Real reportInterval)
151             : PeriodicEventReporter(reportInterval), system(system), mobod(mobod), moBodyName(moBodyName) {}
152         void handleEvent(const State& state) const {
153             system.realize(state, Stage::Velocity);
154             Vec3 vel = mobod.getBodyOriginVelocity(state);
155             cout << "Velocity of "<< moBodyName << "= { x = " << vel[0] << ", y = " << vel[1]
156                  << ", z = "<< vel[2] << " }" << endl;
157         }
158     private:
159         const MultibodySystem& system;
160         const MobilizedBody& mobod;
161         const string moBodyName;
162
163     };
164
165     class AngleReporter : public PeriodicEventReporter {
166     public:
167         AngleReporter(string coordinateName, Model* model, const MultibodySystem &system, Real reportInterval) :
168             PeriodicEventReporter(reportInterval), model(model),
169             coordinateName(coordinateName), system(system) {}
170         void handleEvent(const State& state) const {
171             system.realize(state, Stage::Position);
172             const Coordinate& coord = model->getCoordinateSet().get(coordinateName);
173             double angle = coord.getValue(state);
174             cout << "angle position: " << angle << endl;
175
176         }
177
178     private:
179         string coordinateName;
180         Model* model;
181         const MultibodySystem& system;
182     };
183
184 }
185 #endif // PRINTOPENSIMINFORMATION_H
```

8.6.2 printOpenSimInformation.cpp

```
1  #include "printOpenSimInformation.h"
2
3  namespace OpenSim {
4
5      // PrintToFile
```

```
6   void PrintToFile::PrintDataToFile(std::vector<string> nameData, std::vector<double> data)
7   {
8       static bool realized = false;
9       if(!realized)
10      {
11          for(int i = 0; i < nameData.size(); i++)
12              file << nameData[i] << "\t";
13          file << endl;
14          realized = true;
15      }
16      for(int i = 0; i < data.size(); i++)
17          file << data[i] << "\t";
18      file << endl;
19  }
20
21  void PrintToFile::PrintDataToFile(string* nameData, double* data, int size)
22  {
23      static bool realized = false;
24      if(!realized)
25      {
26          for(int i = 0; i < size; i++)
27              file << nameData[i] << "\t";
28          file << endl;
29          realized = true;
30      }
31      for(int i = 0; i < size; i++)
32          file << data[i] << "\t";
33      file << endl;
34
35  }
36
37  // PrintInformation Class
38  void PrintInformation::printVector(const SimTK::Vector &vec)
39  {
40    stream << vec.toString() << endl;
41  }
42
43  void PrintInformation::printVec3(const SimTK::Vec3& vec)
44  {
45
46      stream << "[" << vec[0]
47               << ", " << vec[1]
48               << ", " << vec[2]
49               << "]" << endl;
50  }
51
52  void PrintInformation::printInertia(const SimTK::Inertia &inertia)
53  {
54      SimTK::Mat33 fMInertia(inertia.toMat33());
55      stream << "[" << fMInertia[0][0] <<", " << fMInertia[0][1] << ", " << fMInertia[0][2] << ";" << "\n"
56             << fMInertia[1][0] <<", " << fMInertia[1][1] << ", " << fMInertia[1][2] << ";" << "\n"
57             << fMInertia[2][0] <<", " << fMInertia[2][1] << ", " << fMInertia[2][2] << "]" << endl;
58
59  }
60
61  void PrintInformation::printSpatialMat(const SimTK::SpatialMat &spatialMat, bool matlab)
62  {
63      /**
64       *
65       *
66       *     | m0000, m0001, m0002, m0100, m0101, m0102 |
67       *     | m0010, m0011, m0012, m0110, m0111, m0112 |
68       * m = | m0020, m0021, m0022, m0120, m0121, m0122 |
69       *     | m1000, m1001, m1002, m1100, m1101, m1102 |
70       *     | m1010, m1011, m1012, m1110, m1111, m1112 |
71       *     | m1020, m1021, m1022, m1120, m1121, m1122 |
72       *
73       *
```

```
74          */
75          const SimTK::SpatialMat &m(spatialMat);
76          if (matlab) {
77          stream << "["
78             << m[0][0][0][0] << "," << m[0][0][0][1] << "," << m[0][0][0][2] << ","
79                  << m[0][1][0][0] << "," << m[0][1][0][1] << "," << m[0][1][0][2] << ";" << "\n" // First matrix row
80
81             << m[0][0][1][0] << "," << m[0][0][1][1] << "," << m[0][0][1][2] << ","
82                  << m[0][1][1][0] << "," << m[0][1][1][1] << "," << m[0][1][1][2] << ";" << "\n" // Second matrix row
83
84             << m[0][0][2][0] << "," << m[0][0][2][1] << "," << m[0][0][2][2] << ","
85                  << m[0][1][2][0] << "," << m[0][1][2][1] << "," << m[0][1][2][2] << ";" << "\n" // Third matrix row
86
87             << m[1][0][0][0] << "," << m[1][0][0][1] << "," << m[1][0][0][2] << ","
88                 << m[1][1][0][0] << "," << m[1][1][0][1] << "," << m[1][1][0][2] << ";" << "\n" // Fourth matrix row
89
90             << m[1][0][1][0] << "," << m[1][0][1][1] << "," << m[1][0][1][2] << ","
91                 << m[1][1][1][0] << "," << m[1][1][1][1] << "," << m[1][1][1][2] << ";" << "\n" // Fith matrix row
92
93             << m[1][0][2][0] << "," << m[1][0][2][1] << "," << m[1][0][2][2] << ","
94                 << m[1][1][2][0] << "," << m[1][1][2][1] << "," << m[1][1][2][2] << "]" << endl;// Sith matrix row
95          }
96          else
97          {
98              stream << m.toString() << endl;
99          }
100
101     }
102
103     // PrintModelInformation Class
104     void PrintModelInformation::printCoordinateNames()
105     {
106         const CoordinateSet &coordinates(aModel.getCoordinateSet());
107         Array<string> names;
108         coordinates.getNames(names);
109         stream << "COORDINATES (" << names.getSize() << ")" <<endl;
110         for(int i = 0; i < names.getSize(); i++)
111             stream << "coordinate[" << i << "] = "
112                    << names[i] << endl;
113
114     }
115
116     // PrintJointInformation Class
117
118     void PrintJointInformation::printJointNames()
119     {
120         for(int i = 0; i < arrayOfJointNames.getSize(); i++)
121             stream << i << "- " << arrayOfJointNames[i] << endl;
122     }
123
124     void PrintJointInformation::printJointParentLocation(int index)
125     {
126         SimTK::Vec3 locationInParent;
127         joints.get(index).getLocationInParent(locationInParent);
128         stream << index << "- " << arrayOfJointNames[index] << "'s location in parent: ";
129         printVec3(locationInParent);
130     }
131
132     void PrintJointInformation::printJointChildLocation(int index)
133     {
134         SimTK::Vec3 locationInChild;
135         locationInChild = joints.get(index).getLocationInChild();
136         stream << index << "- " << arrayOfJointNames[index] << "'s location in child: ";
137         printVec3(locationInChild);
138     }
139
140     void PrintJointInformation::printJointParentOrientation(int index)
141     {
```

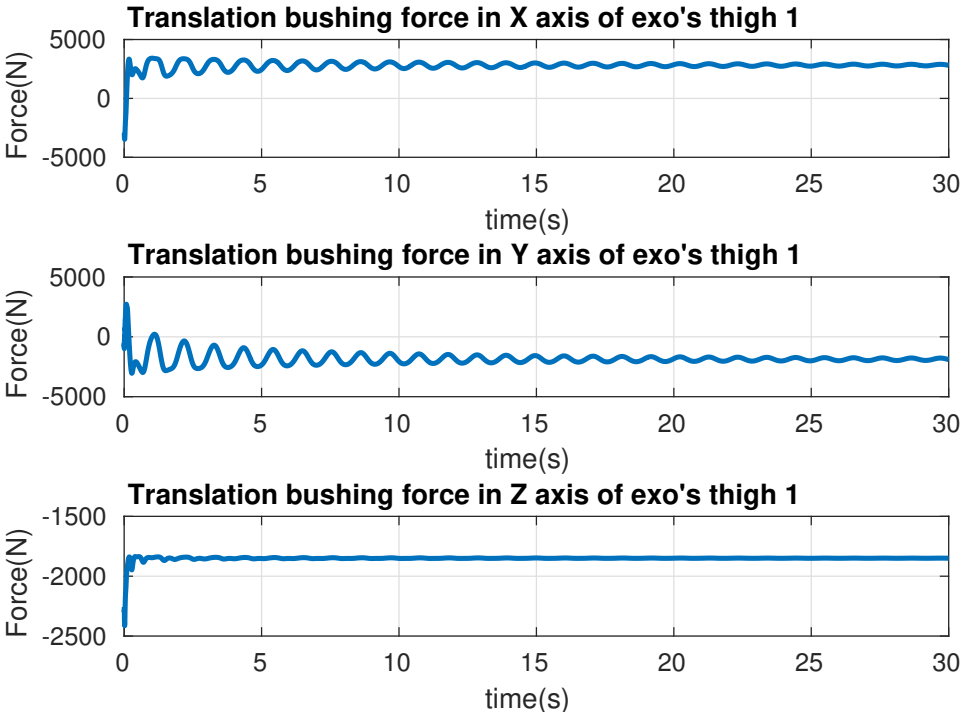
```
142         SimTK::Vec3 orientationInParent;
143         joints.get(index).getOrientationInParent(orientationInParent);
144         stream << index << "- " << arrayOfJointNames[index] << "'s orientation in parent: ";
145         printVec3(orientationInParent);
146     }
147
148     void PrintJointInformation::printAllJointParentLocations()
149     {
150         SimTK::Vec3 locationInParent;
151         for (int index = 0; index < joints.getSize(); index++)
152         {
153             joints.get(index).getLocationInParent(locationInParent);
154             stream << index << "- " << arrayOfJointNames[index] <<"'s location in parent: ";
155             printVec3(locationInParent);
156         }
157     }
158
159     void PrintJointInformation::printAllJointParentOrientation()
160     {
161         SimTK::Vec3 orientationInParent;
162         for (int index = 0; index < joints.getSize(); index++)
163         {
164             joints.get(index).getOrientationInParent(orientationInParent);
165             stream << index << "- " << arrayOfJointNames[index] <<"'s location in parent: ";
166             printVec3(orientationInParent);
167         }
168
169     }
170
171     // PrintForceToFile Class
172     void PrintForceToFile::getForces() {
173         forceReporter->getForceStorage().getDataForIdentifier(forceName, rData);
174     }
175
176     void PrintForceToFile::getForceNames() {
177         forceNames = forceReporter->getForceStorage().getColumnLabels();
178     }
179
180     void PrintForceToFile::printForces() {
181         getForces();
182         getForceNames();
183         int firstIndex, lastIndex;
184         bool firstTime = false, forceHasBeenFound = false;
185         file << "This file print the force that are interested on:" << '\n';
186         for(int i = 0; i < forceNames.getSize(); i++) {
187             size_t found = forceNames[i].find(forceName);
188             if (found != string::npos) {
189                 if(!firstTime) {
190                     firstIndex = i;
191                     forceHasBeenFound = true;
192                     firstTime = false;
193                 }
194                 file << forceNames[i] << '\t';
195             }
196             else if(forceHasBeenFound) {
197                 file << '\n';
198                 forceHasBeenFound = false;
199                 lastIndex = i;
200             }
201         }
202         for(int j = 0; j < rData[0].getSize(); j++) {
203             for(int i = 0; i < rData.getSize(); i++) {
204                 file << std::fixed << std::setprecision(3) << rData[i][j] << '\t';
205             }
206             file << '\n';
207         }
208
209     file.close();
```

```
210     }
211
212 }
213
214 //int main()
215 //{
216
217 //    return 0;
218 //}
```

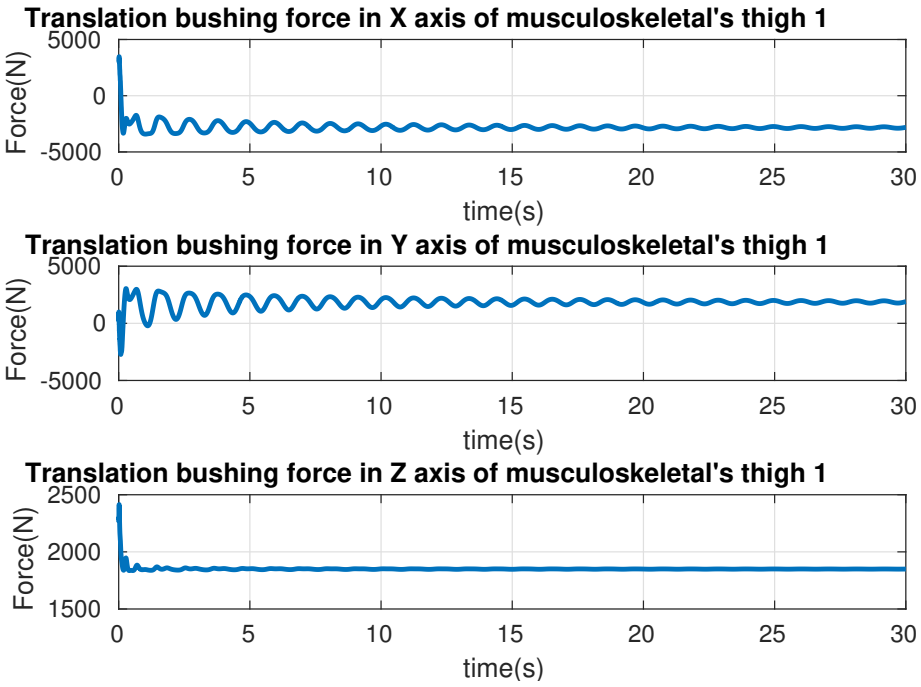
8.7 Plots of the exoskeleton coupled to muscoloskeletal model

This section will show all the bushing forces of test one and two.

8.7.1 Plots of test1's bushing forces

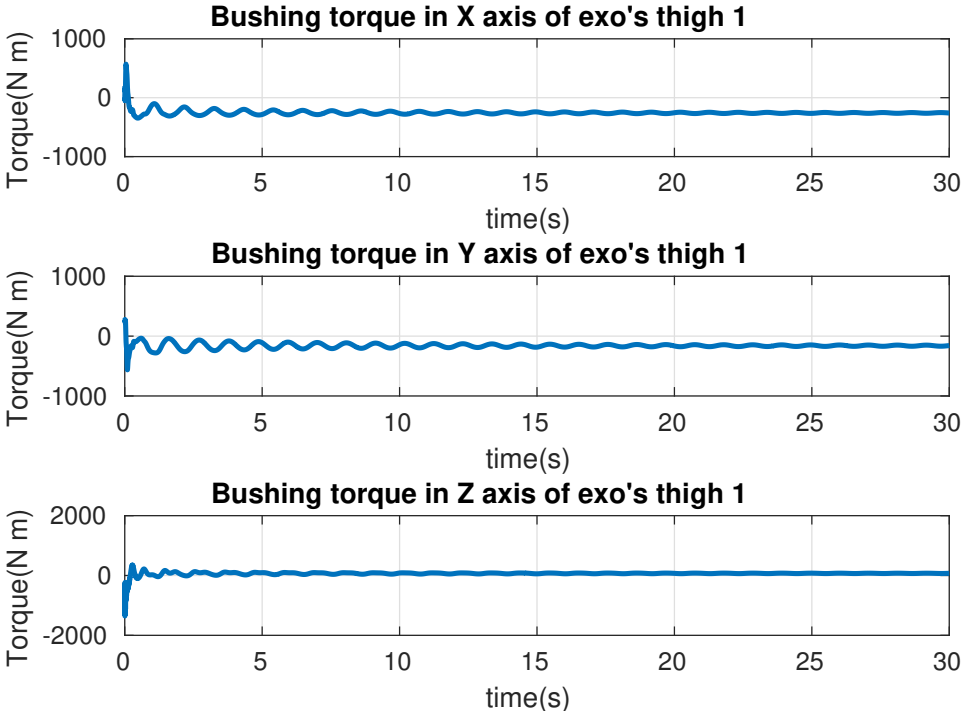


(a) Translation bushing forces of exo's thigh 1

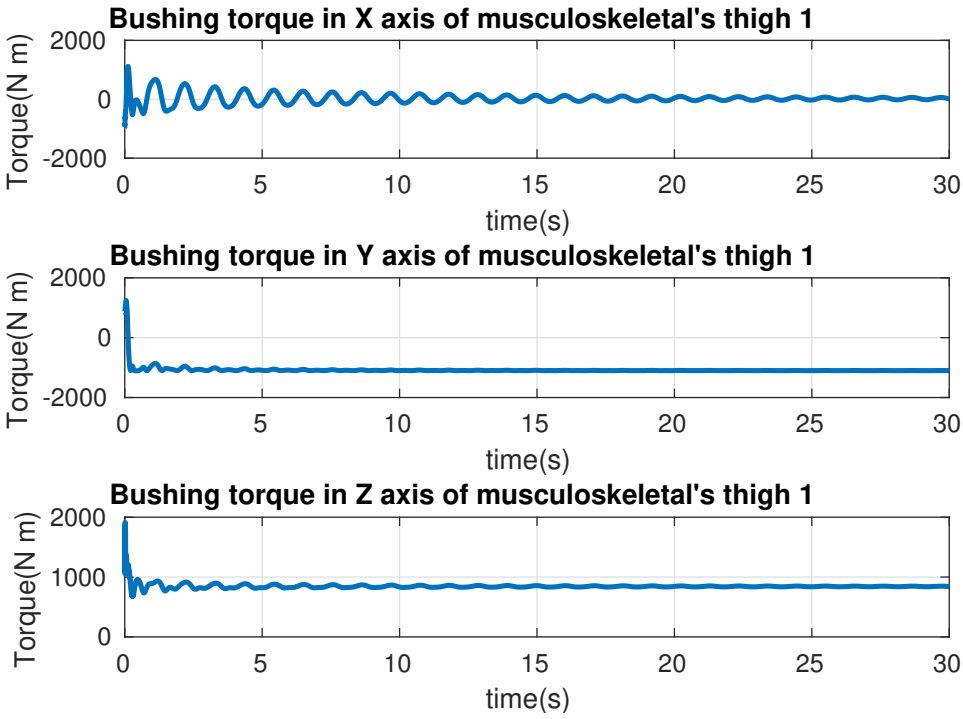


(b) Translation bushing forces of musculoskeletal's thigh 1

Figure 8.1: Translation bushing forces of exo's and musculoskeletal's thigh 1

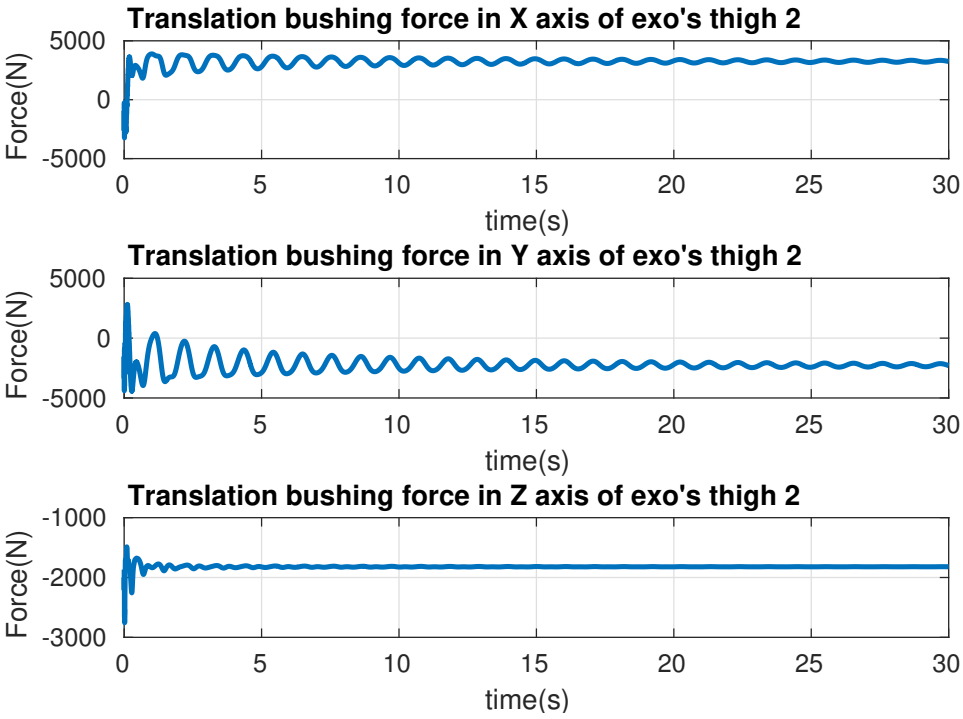


(a) Bushing torque of exo's thigh 1

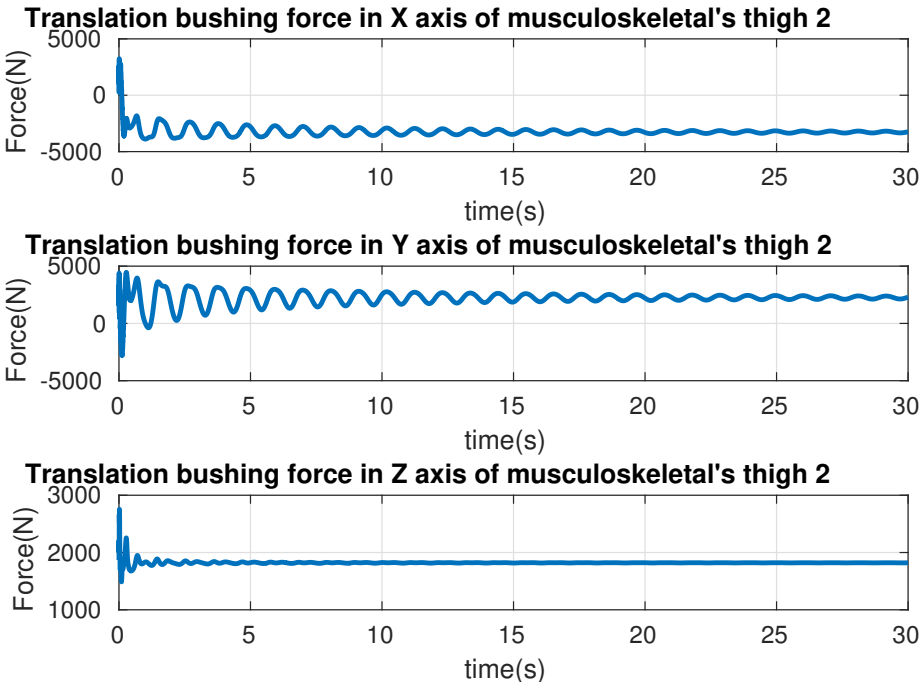


(b) Bushing torque of musculoskeletal's thigh 1

Figure 8.2: Bushing torque of exo's and musculoskeletal's thigh 1

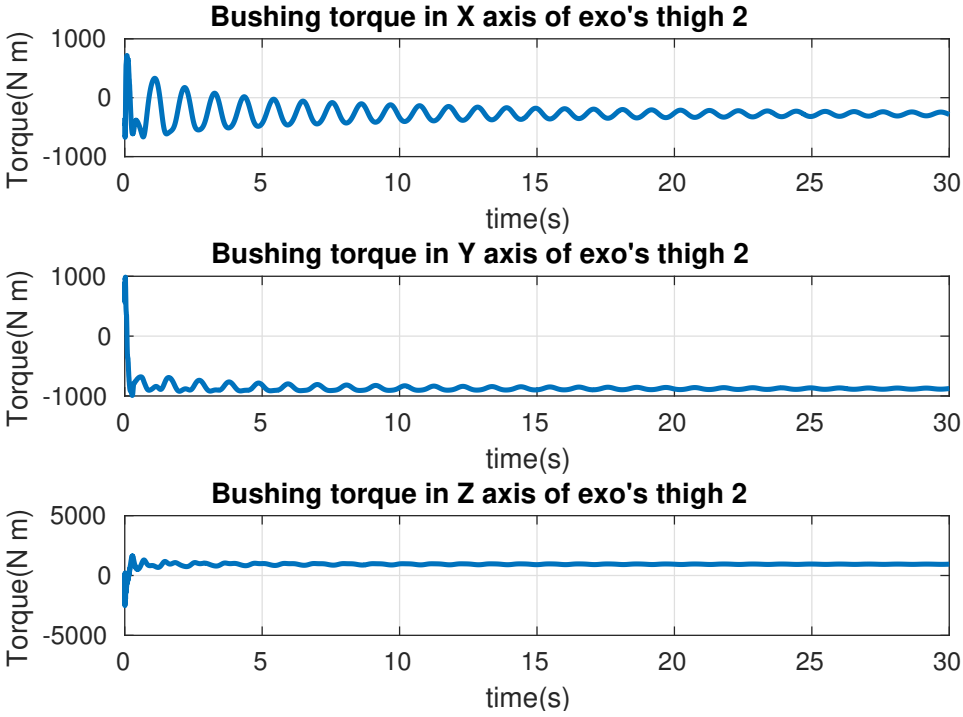


(a) Translation bushing forces of exo's thigh 2

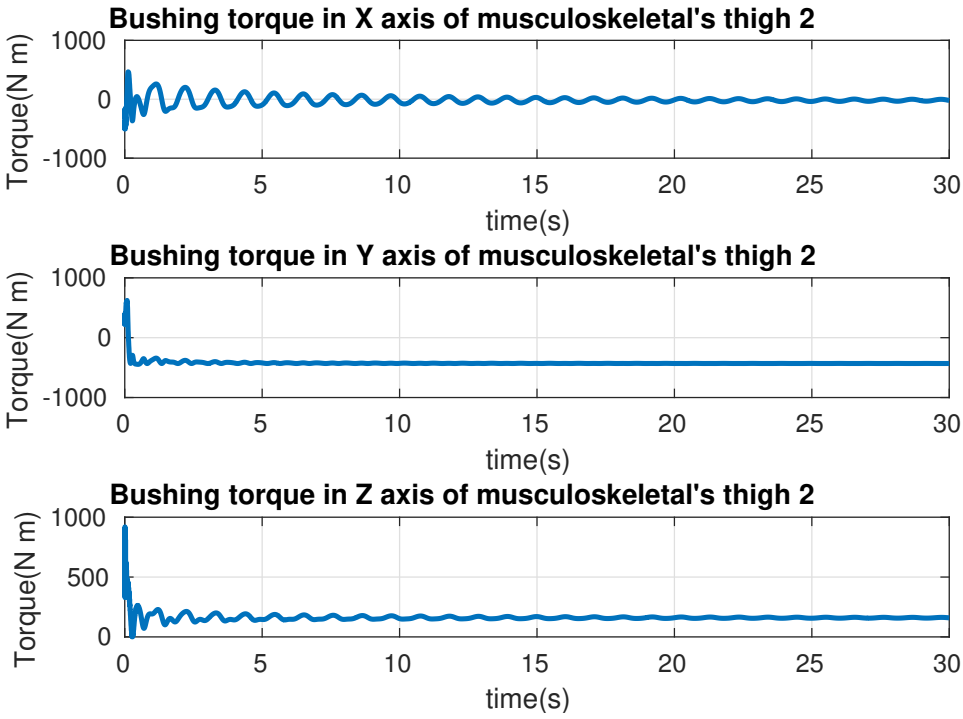


(b) Translation bushing forces of musculoskeletal's thigh 2

Figure 8.3: Translation bushing forces of exo's and musculoskeletal's thigh 2

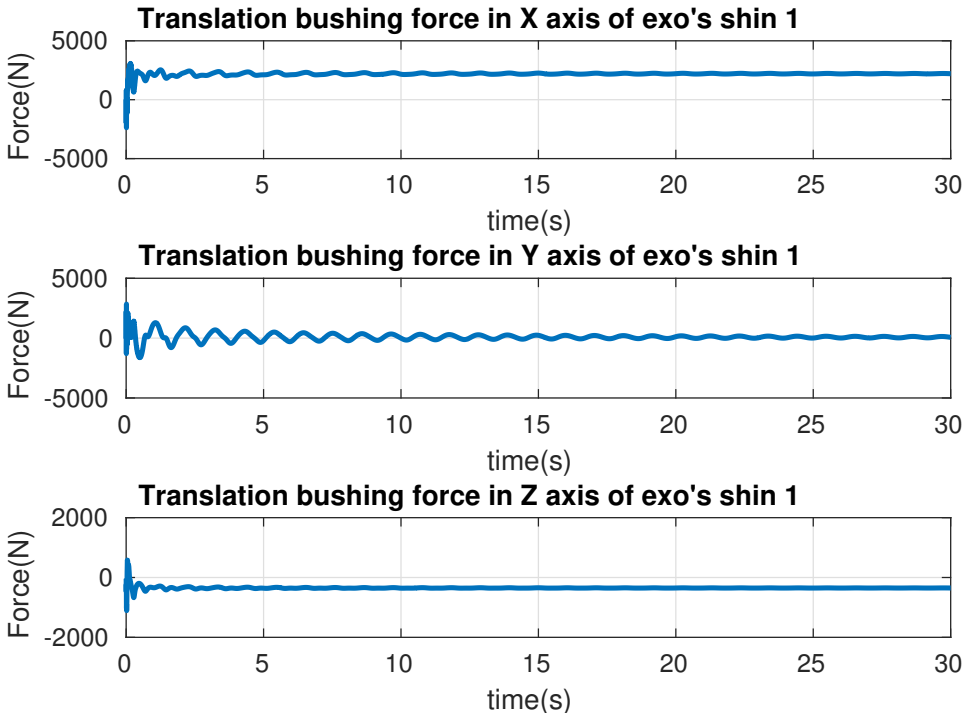


(a) Bushing torque of exo's thigh 2

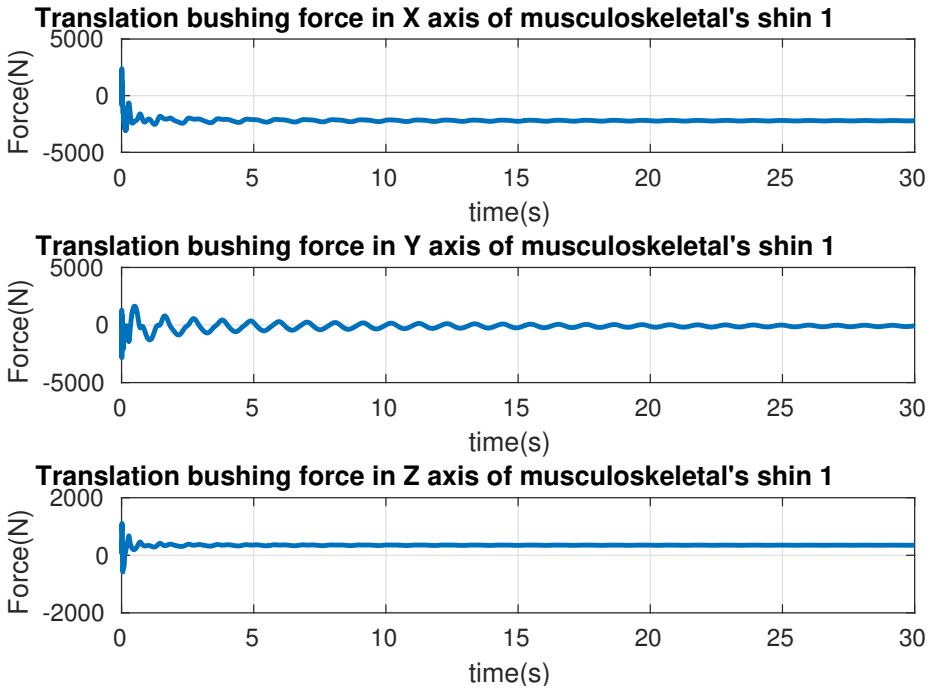


(b) Bushing torque of musculoskeletal's thigh 2

Figure 8.4: Bushing torque of exo's and musculoskeletal's thigh 2

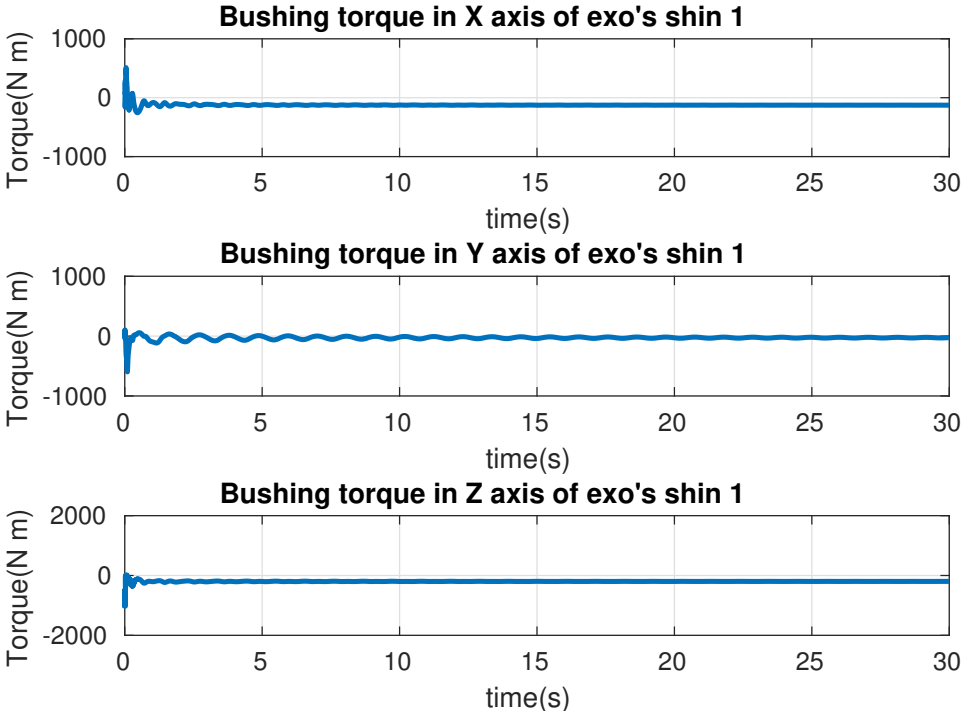


(a) Translation bushing forces of exo's shin 1

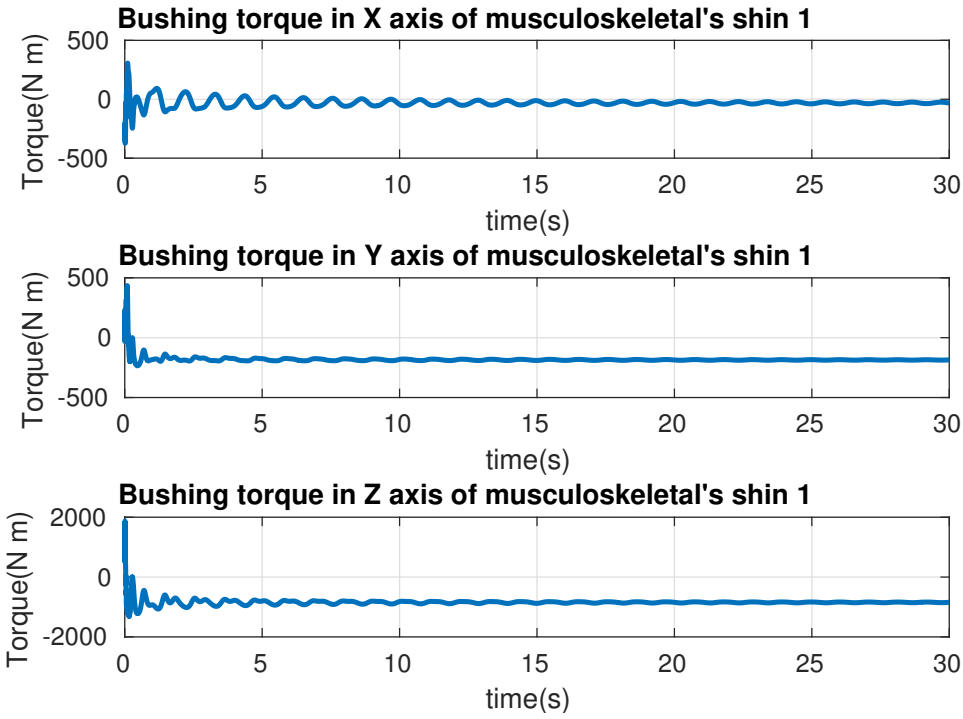


(b) Translation bushing forces of musculoskeletal's shin 1

Figure 8.5: Translation bushing forces of exo's and musculoskeletal's shin 1

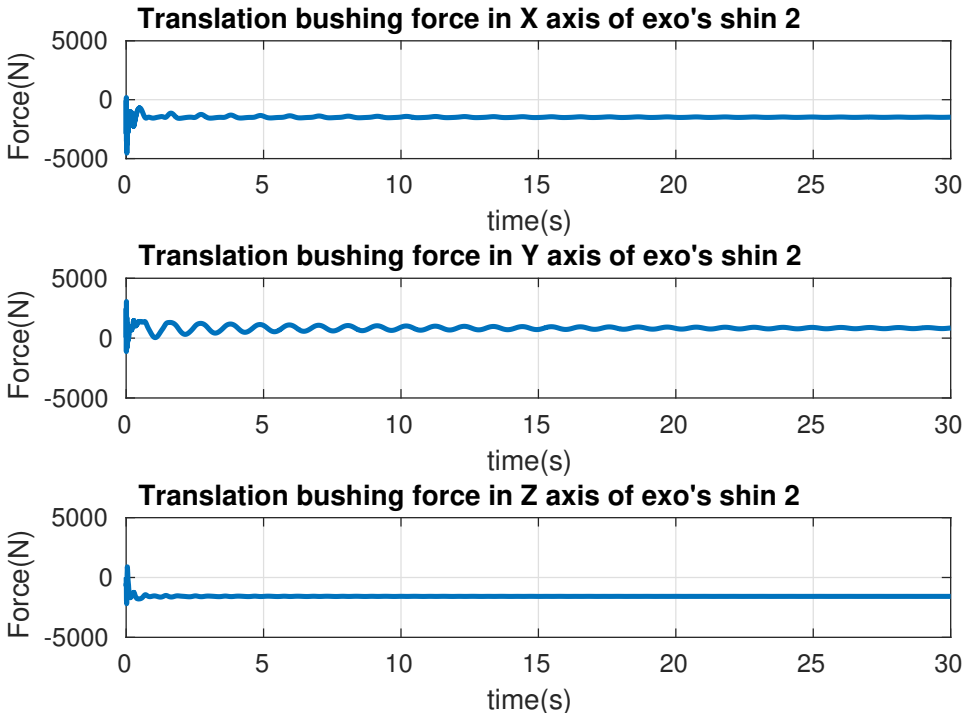


(a) Bushing torque of exo's shin 1

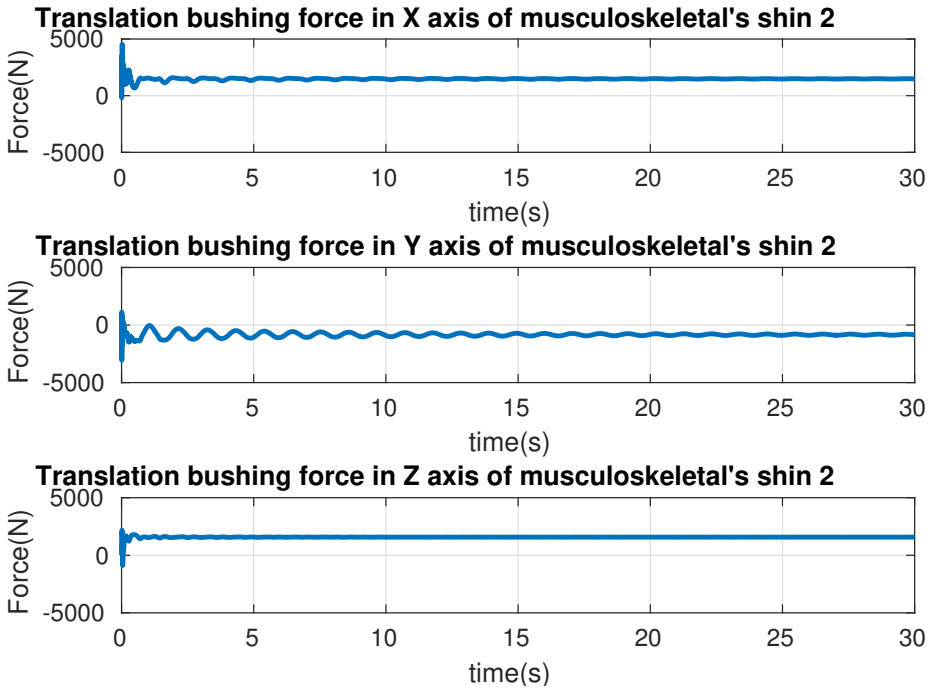


(b) Bushing torque of musculoskeletal's shin 1

Figure 8.6: Bushing torque of exo's and musculoskeletal's shin 1

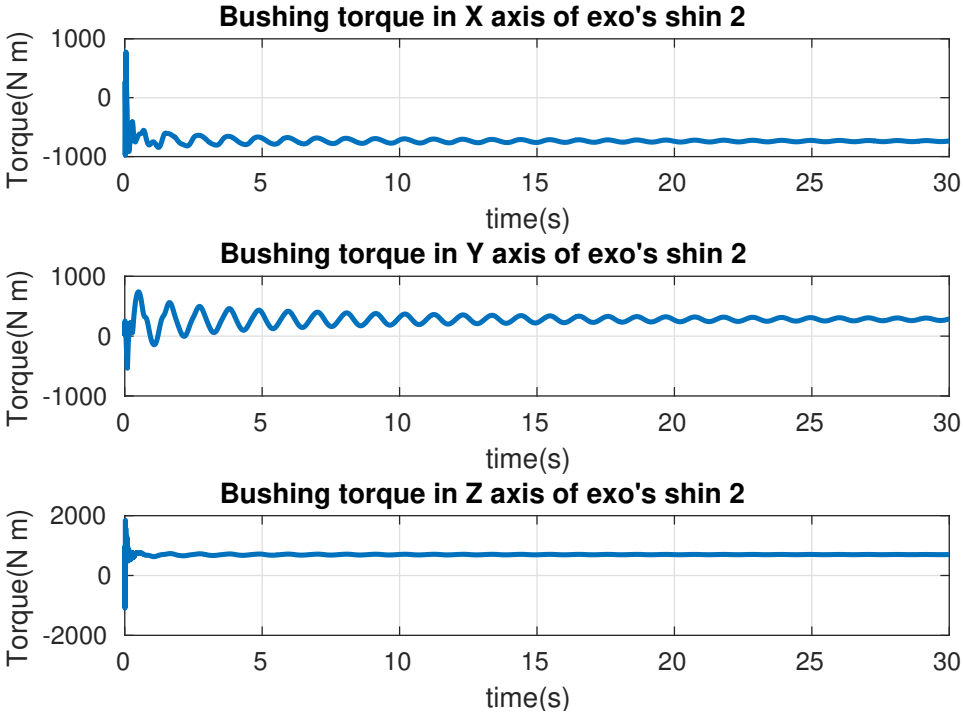


(a) Translation bushing forces of exo's shin 2

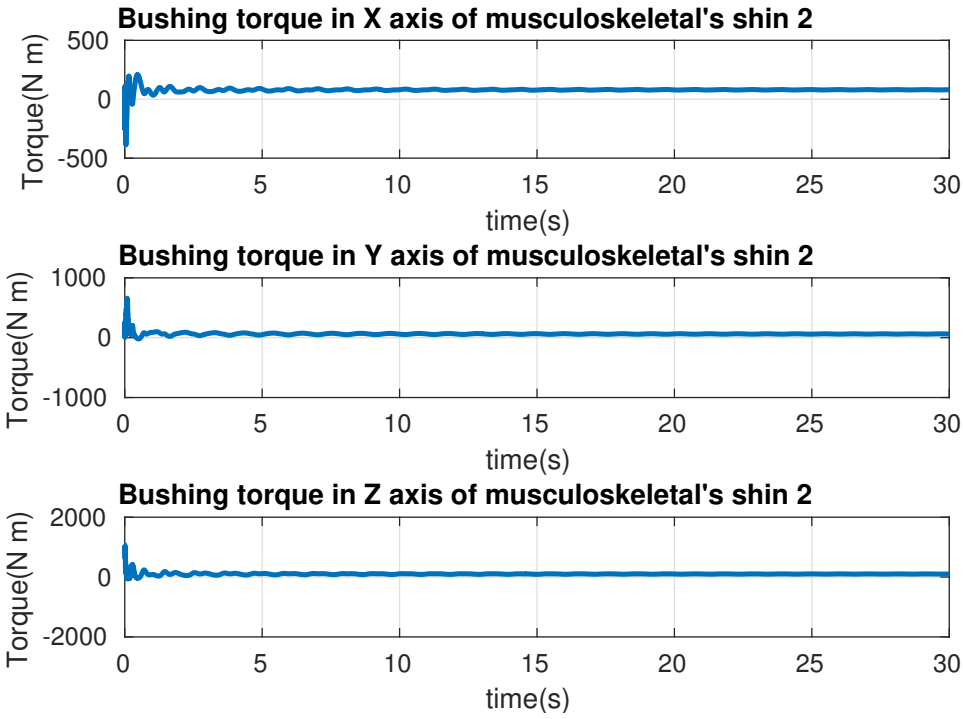


(b) Translation bushing forces of musculoskeletal's shin 2

Figure 8.7: Translation bushing forces of exo's and musculoskeletal's shin 2

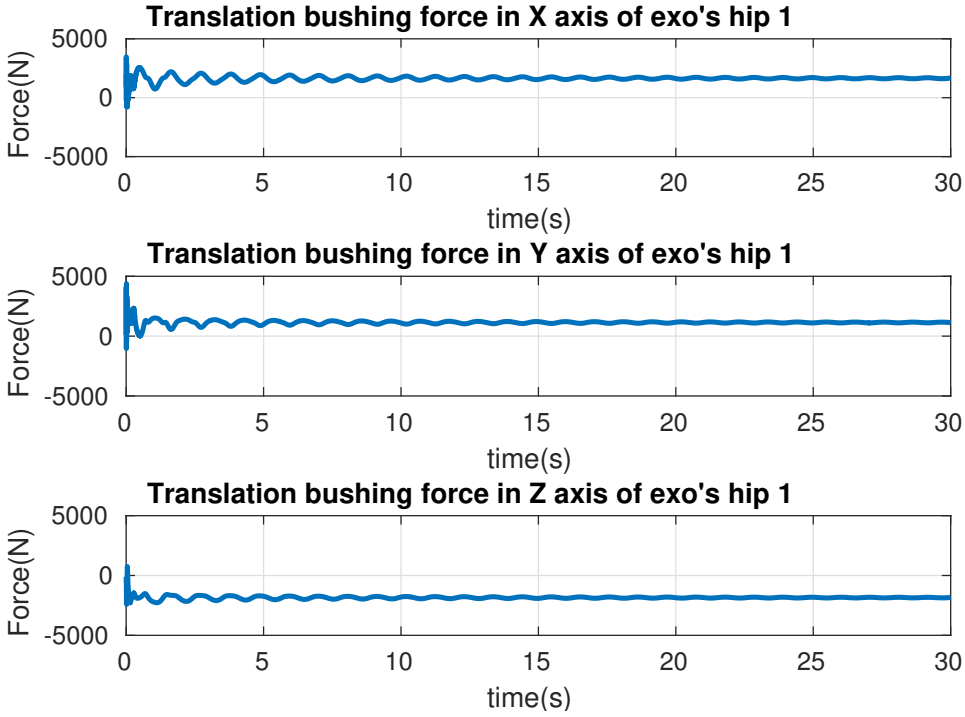


(a) Bushing torque of exo's shin 2

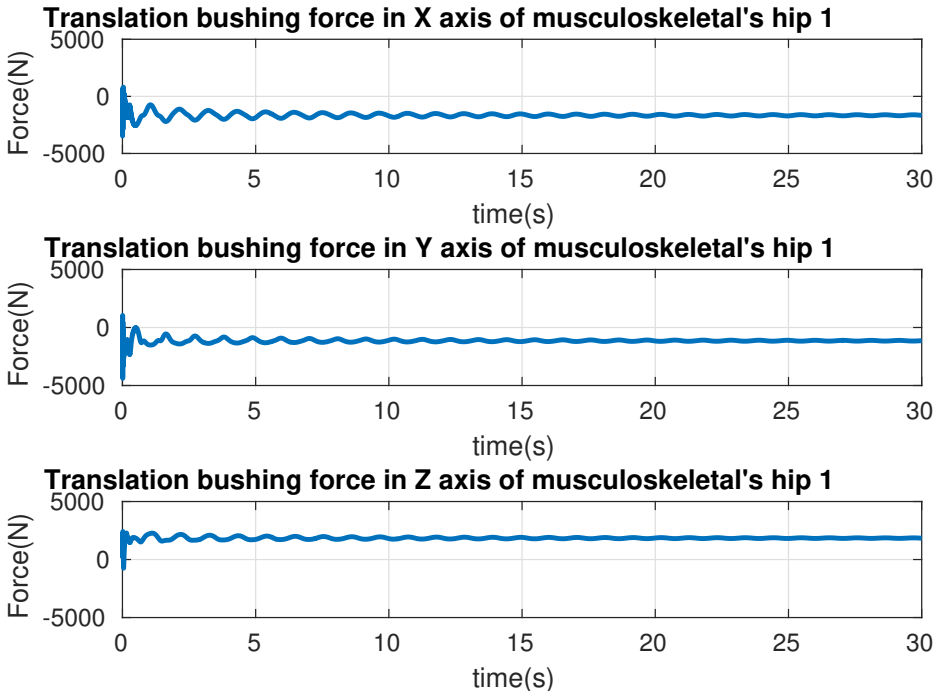


(b) Bushing torque of musculoskeletal's shin 2

Figure 8.8: Bushing torque of exo's and musculoskeletal's shin 2

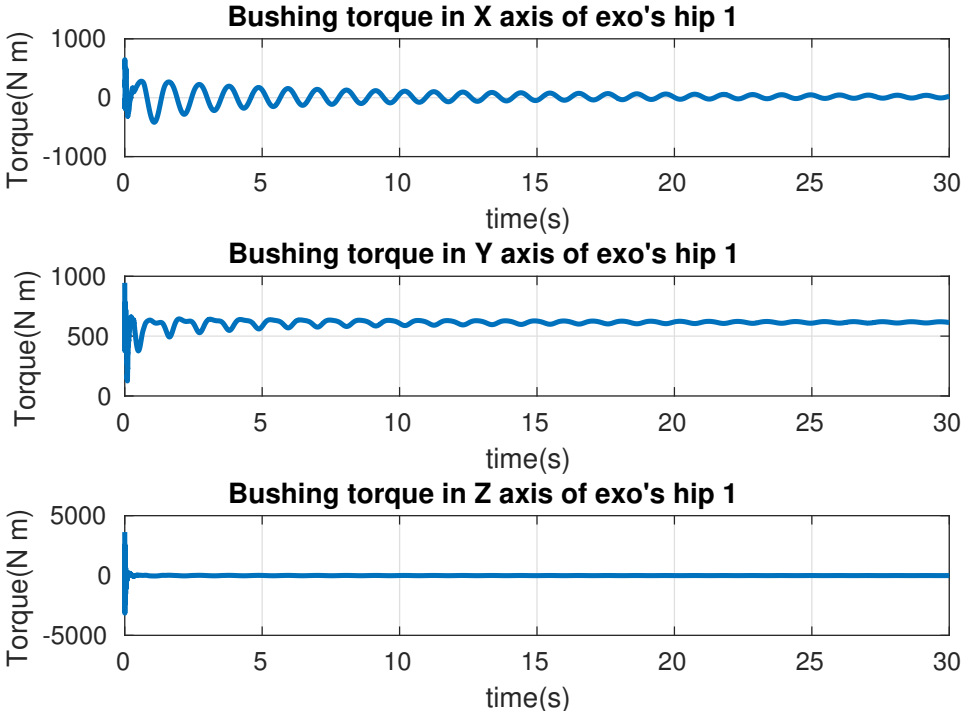


(a) Translation bushing forces of exo's hip 1

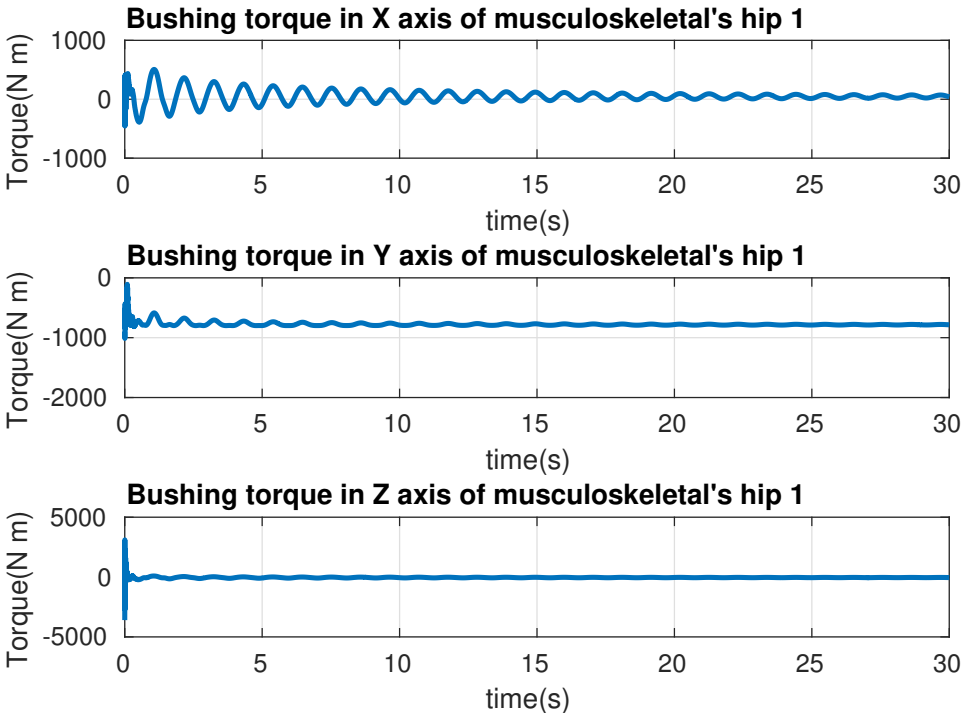


(b) Translation bushing forces of musculoskeletal's hip 1

Figure 8.9: Translation bushing forces of exo's and musculoskeletal's hip 1



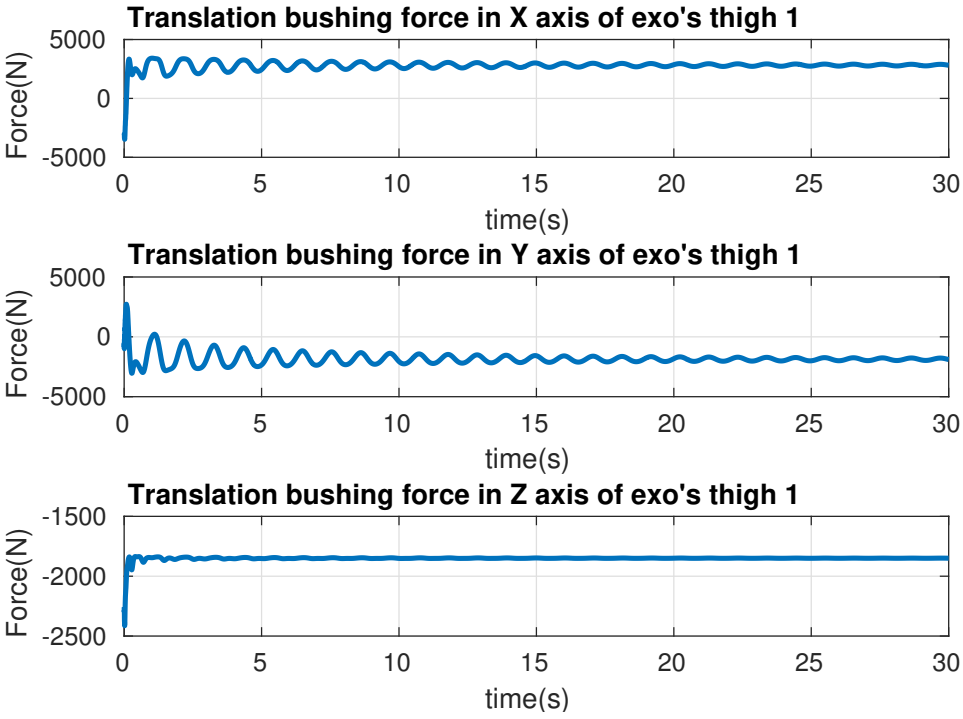
(a) Bushing torque of exo's hip 1



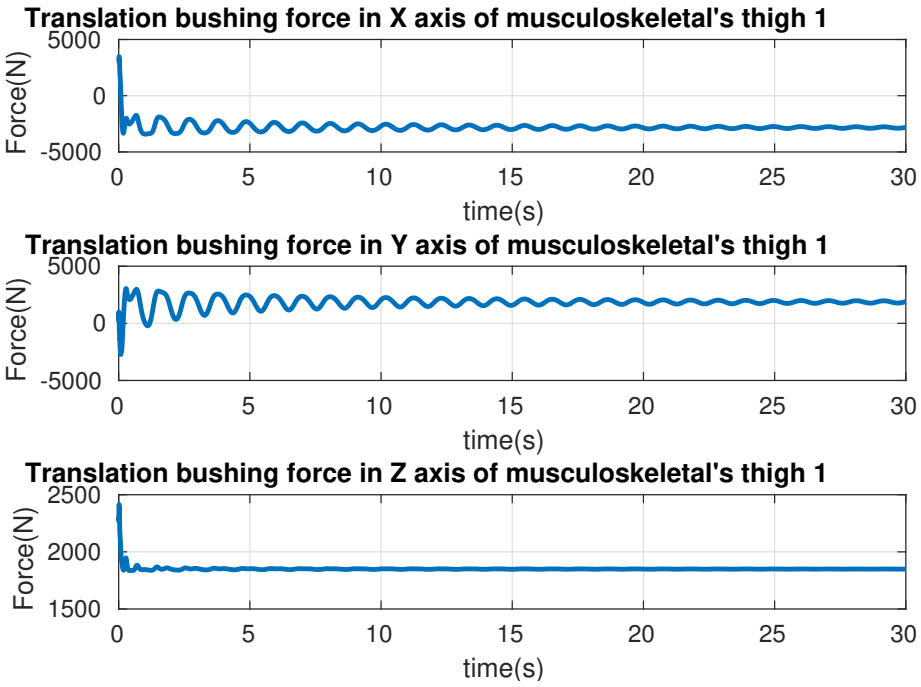
(b) Bushing torque of musculoskeletal's hip 1

Figure 8.10: Bushing torque of exo's and musculoskeletal's hip 1

8.7.2 Plots of test2's bushing forces

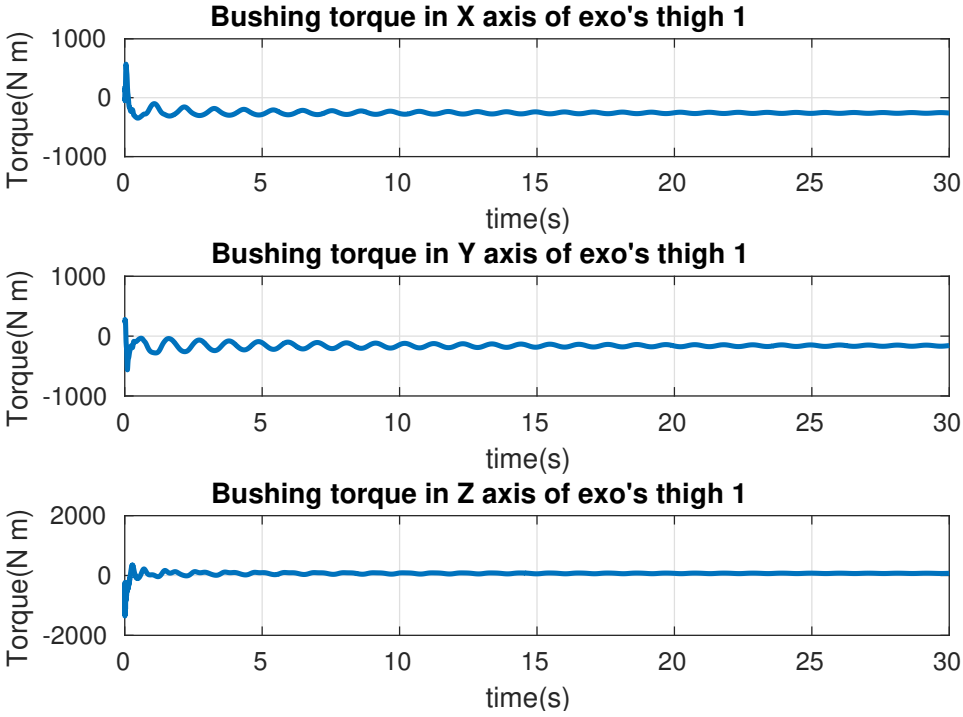


(a) Translation bushing forces of exo's thigh 1

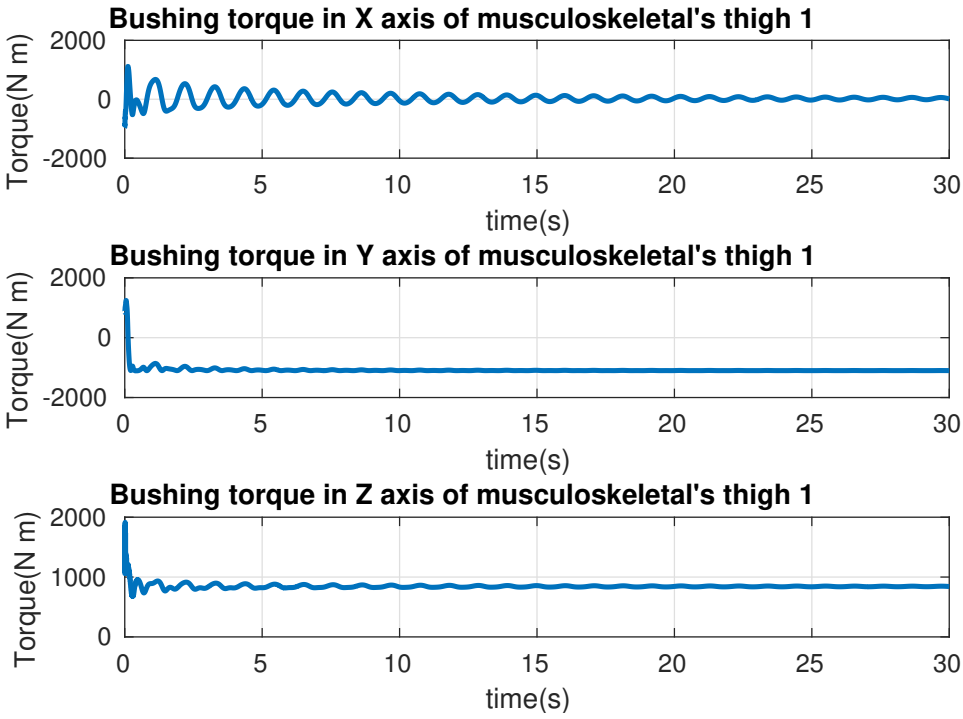


(b) Translation bushing forces of musculoskeletal's thigh 1

Figure 8.11: Translation bushing forces of exo's and musculoskeletal's thigh 1

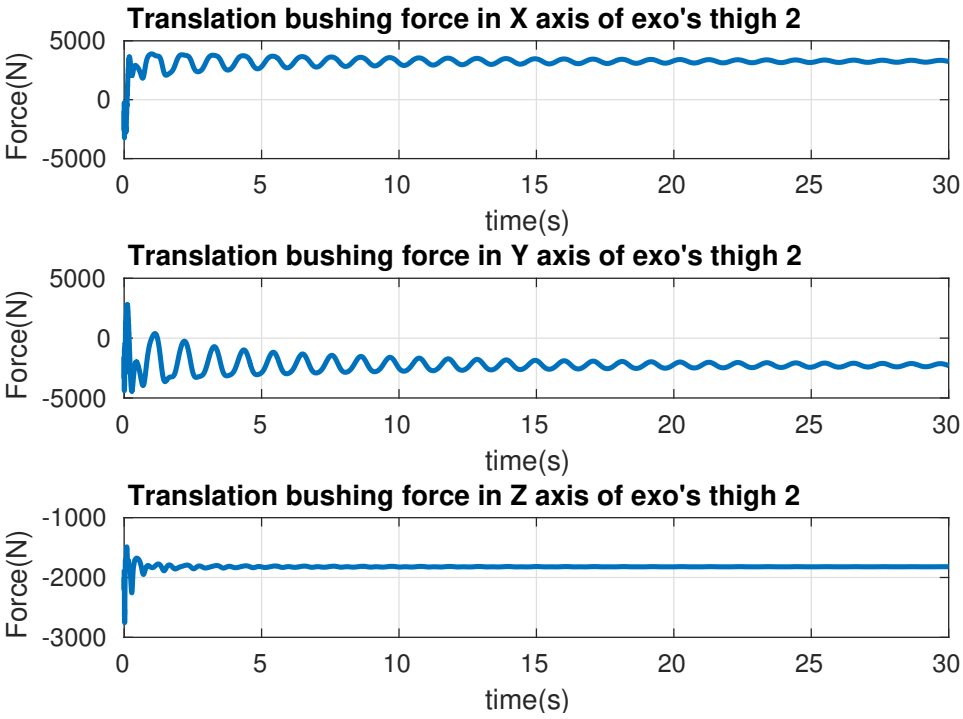


(a) Bushing torque of exo's thigh 1

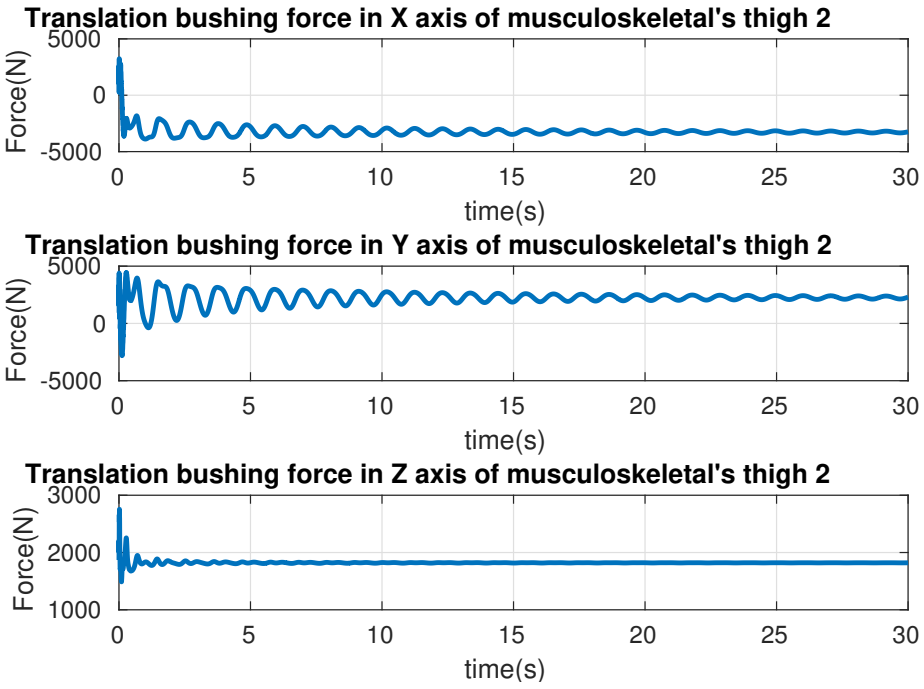


(b) Bushing torque of musculoskeletal's thigh 1

Figure 8.12: Bushing torque of exo's and musculoskeletal's thigh 1

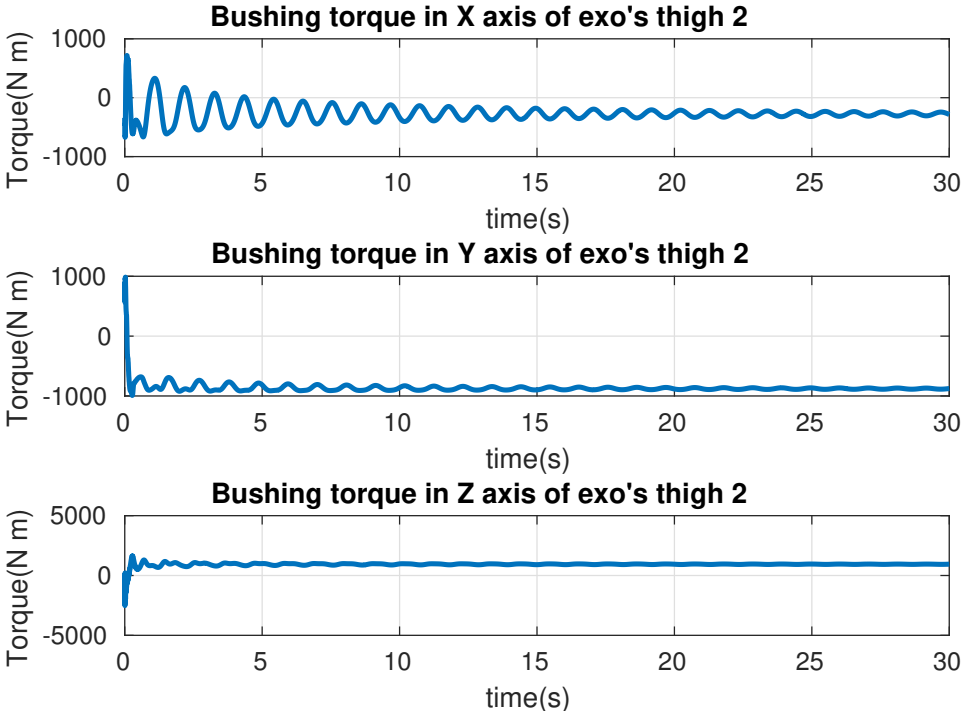


(a) Translation bushing forces of exo's thigh 2

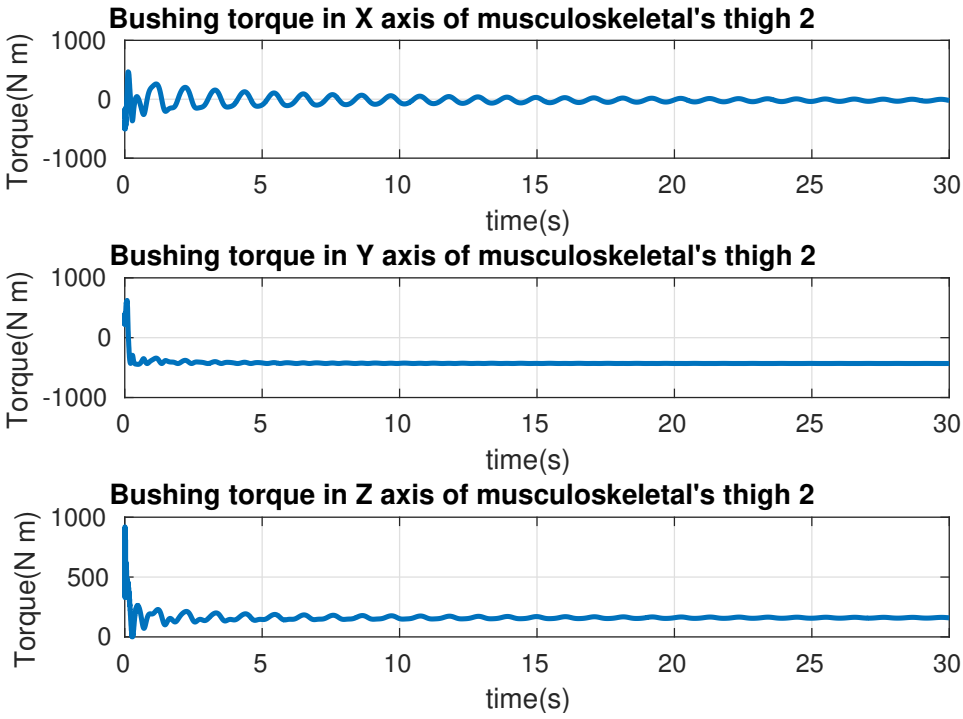


(b) Translation bushing forces of musculoskeletal's thigh 2

Figure 8.13: Translation bushing forces of exo's and musculoskeletal's thigh 2

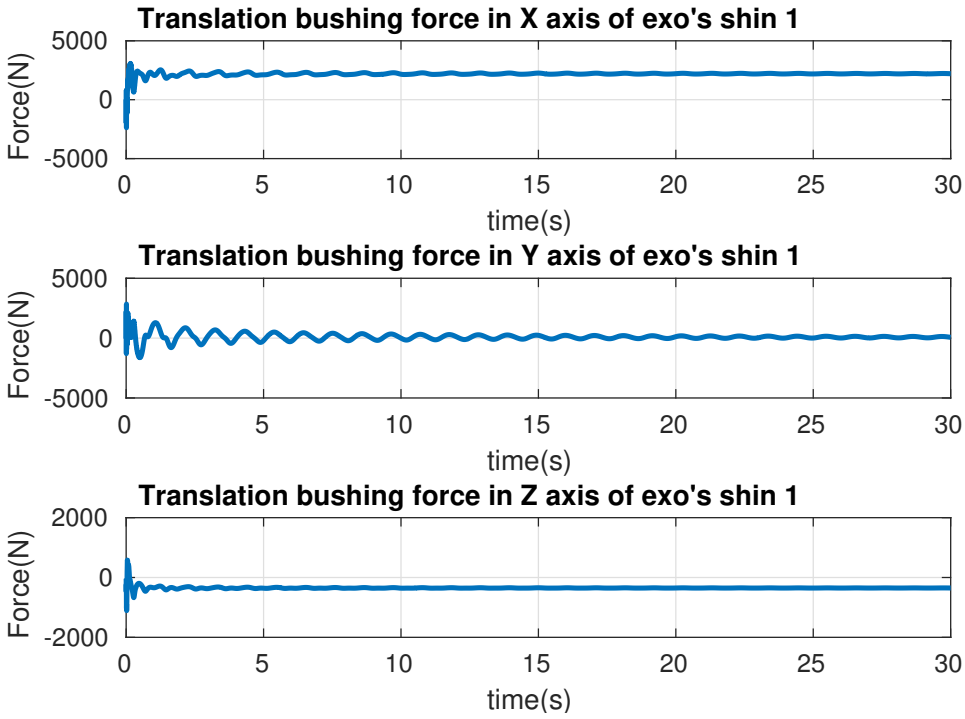


(a) Bushing torque of exo's thigh 2

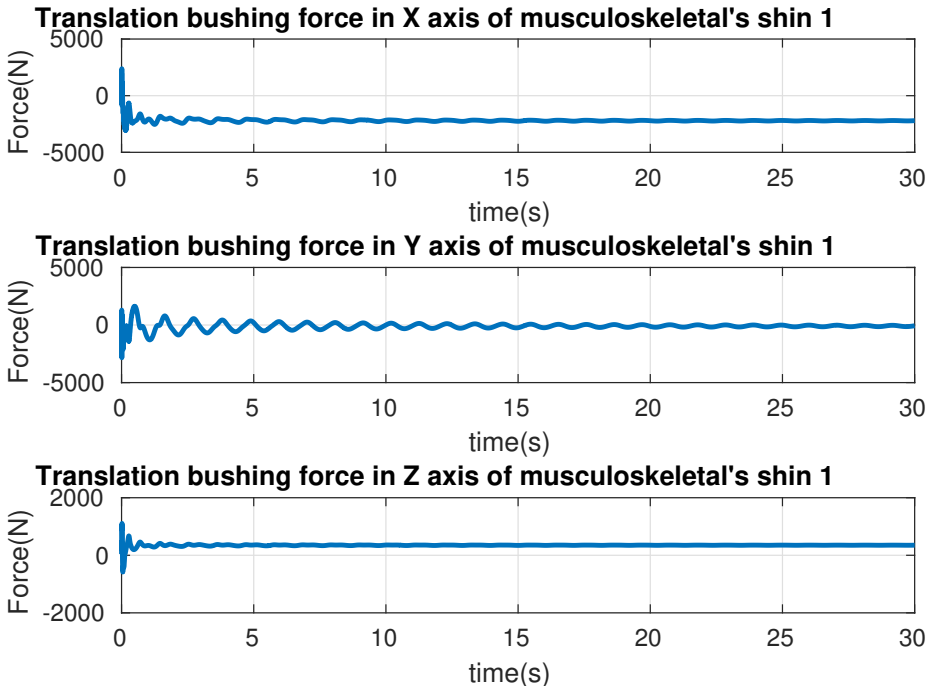


(b) Bushing torque of musculoskeletal's thigh 2

Figure 8.14: Bushing torque of exo's and musculoskeletal's thigh 2

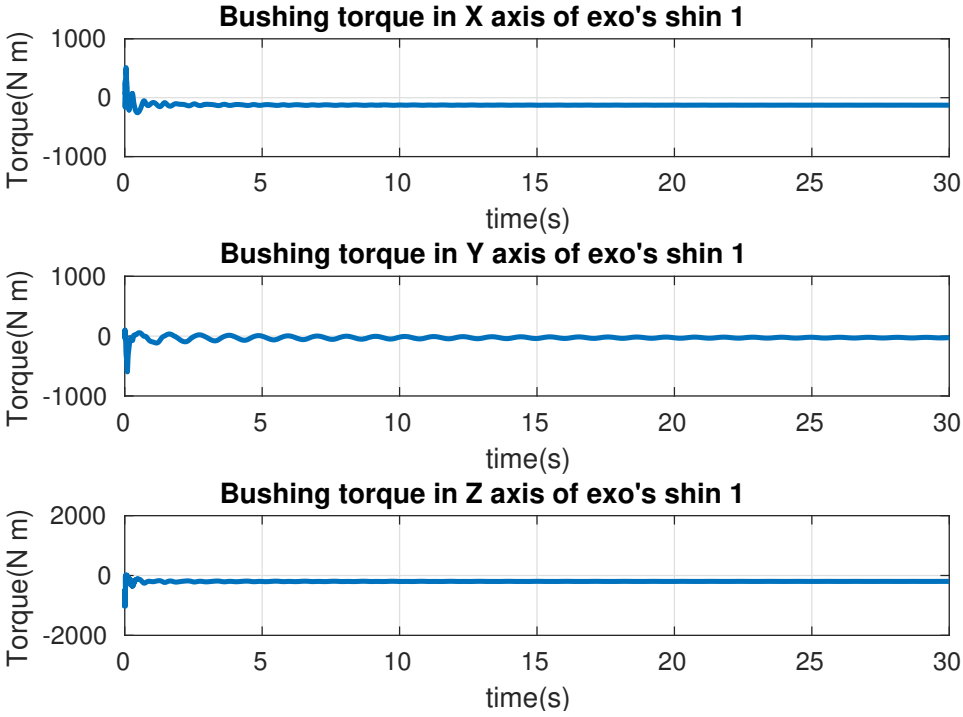


(a) Translation bushing forces of exo's shin 1

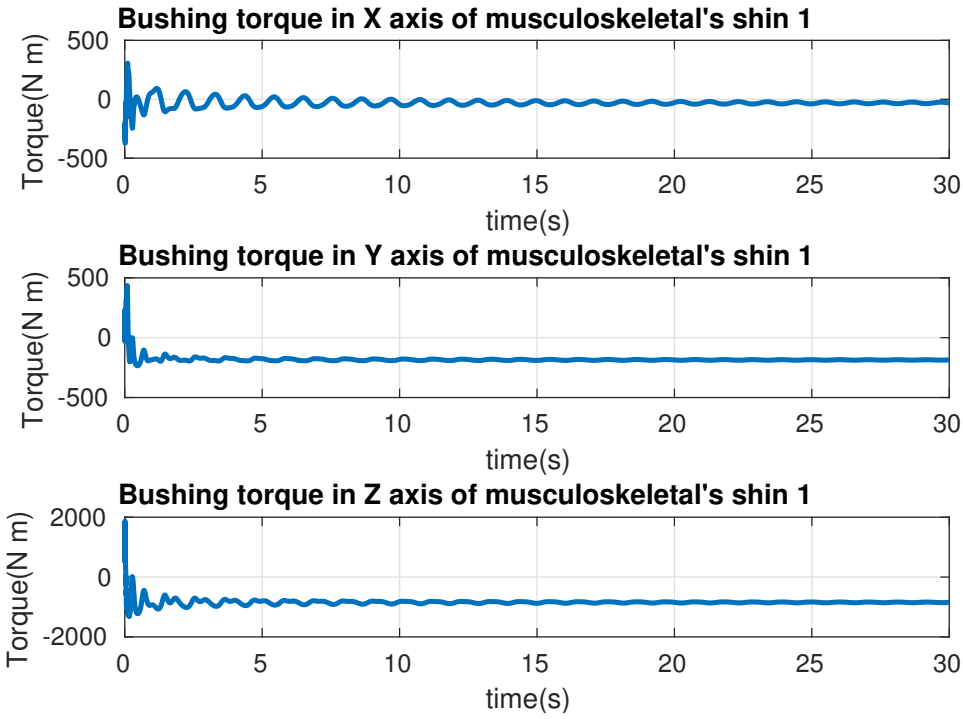


(b) Translation bushing forces of musculoskeletal's shin 1

Figure 8.15: Translation bushing forces of exo's and musculoskeletal's shin 1

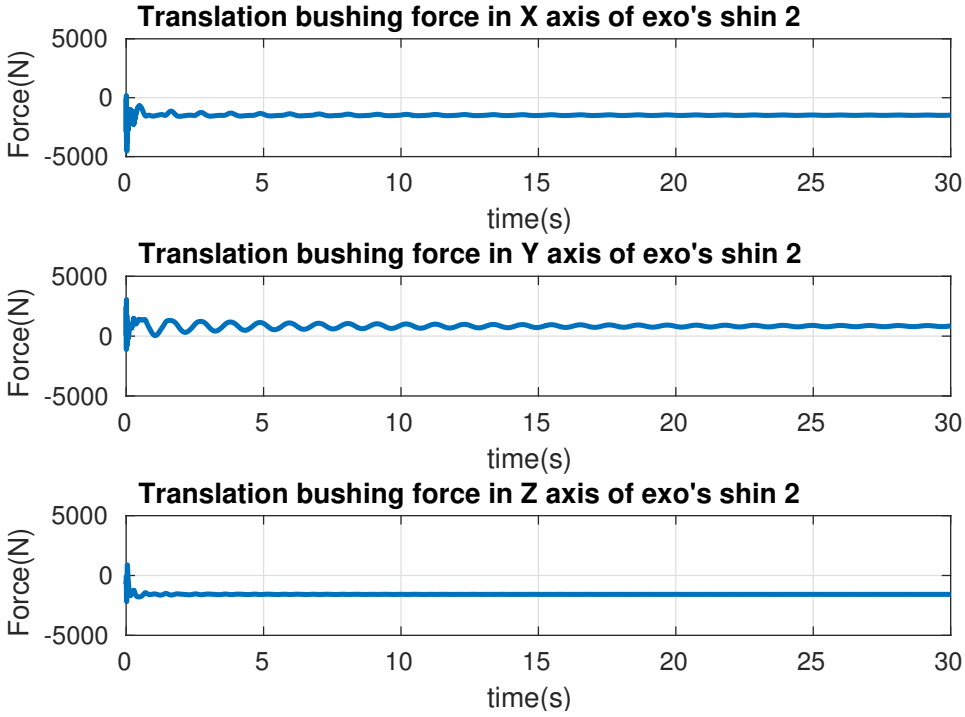


(a) Bushing torque of exo's shin 1

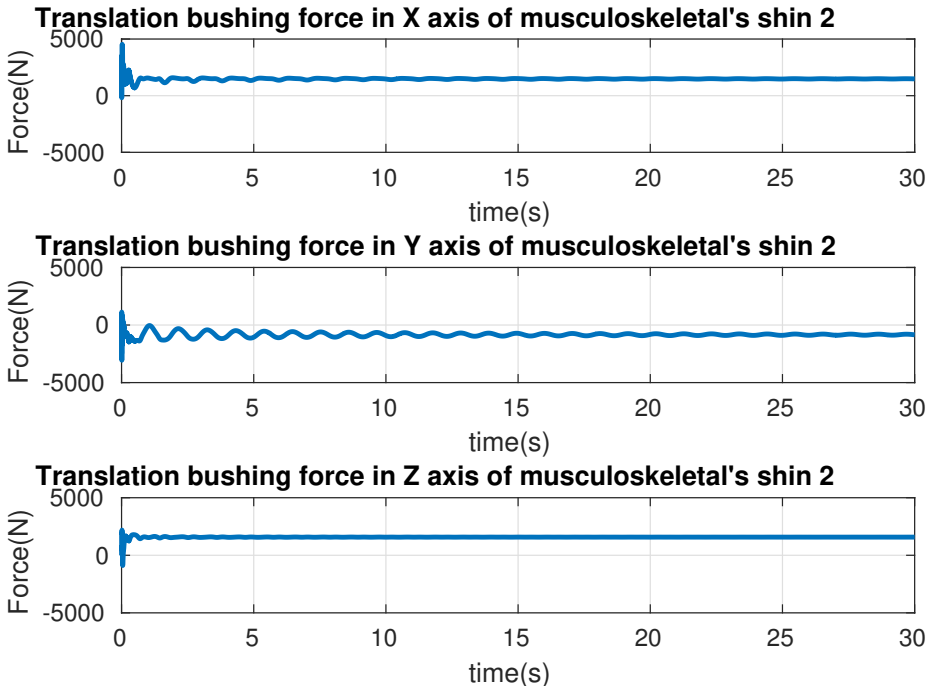


(b) Bushing torque of musculoskeletal's shin 1

Figure 8.16: Bushing torque of exo's and musculoskeletal's shin 1

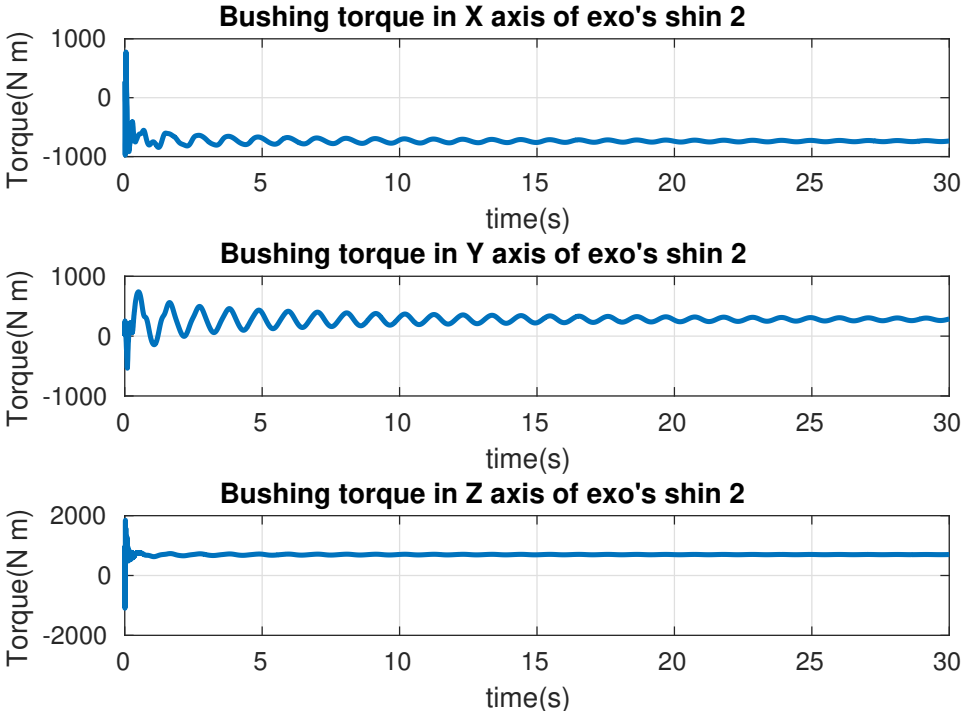


(a) Translation bushing forces of exo's shin 2

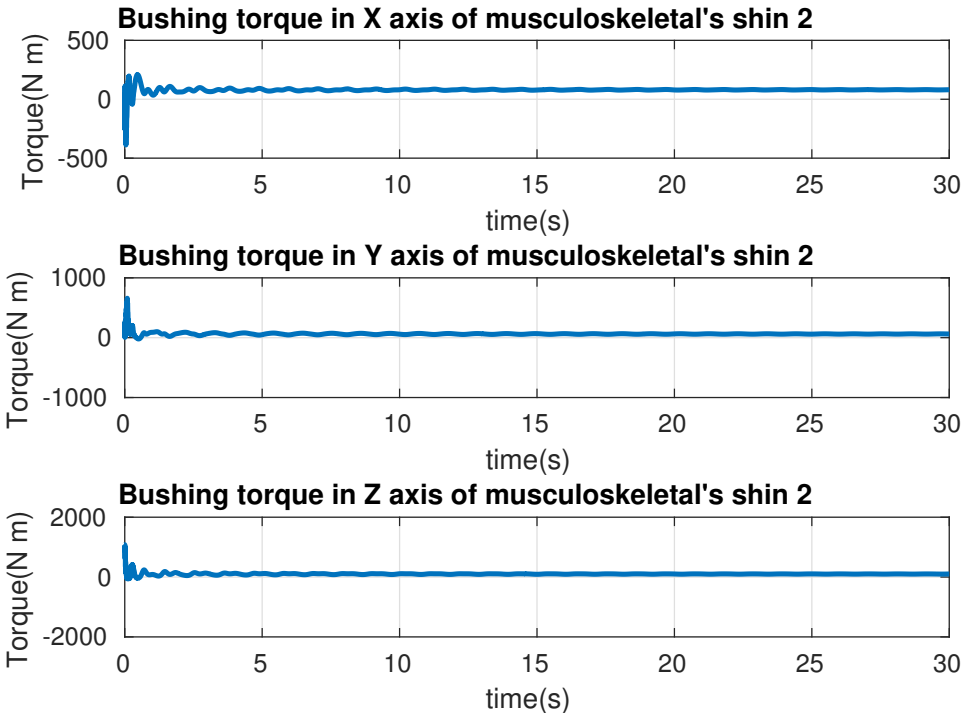


(b) Translation bushing forces of musculoskeletal's shin 2

Figure 8.17: Translation bushing forces of exo's and musculoskeletal's shin 2

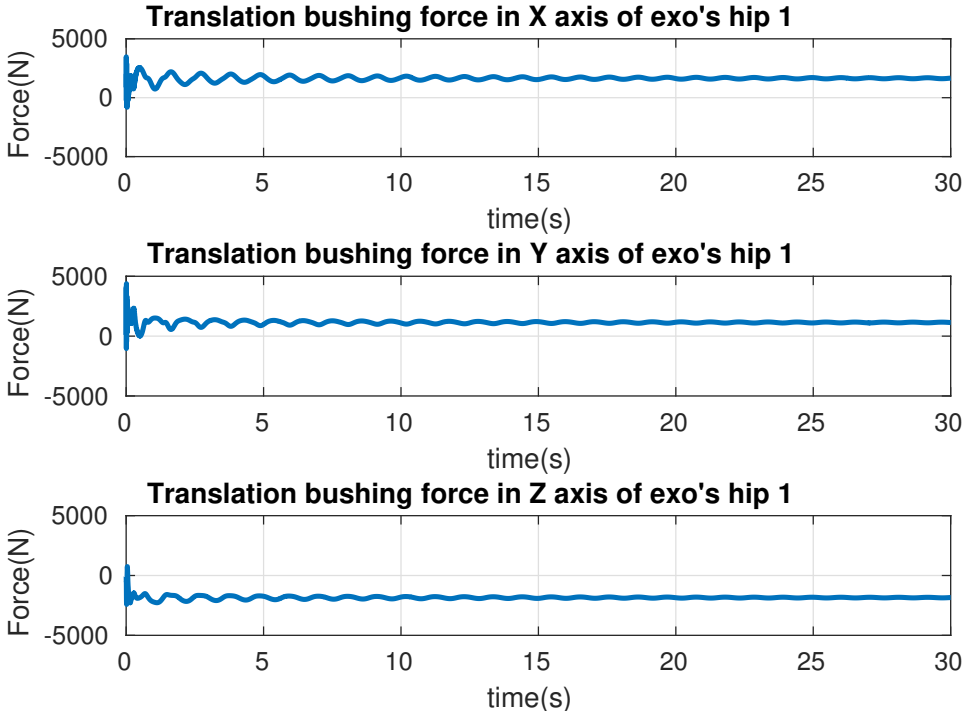


(a) Bushing torque of exo's shin 2

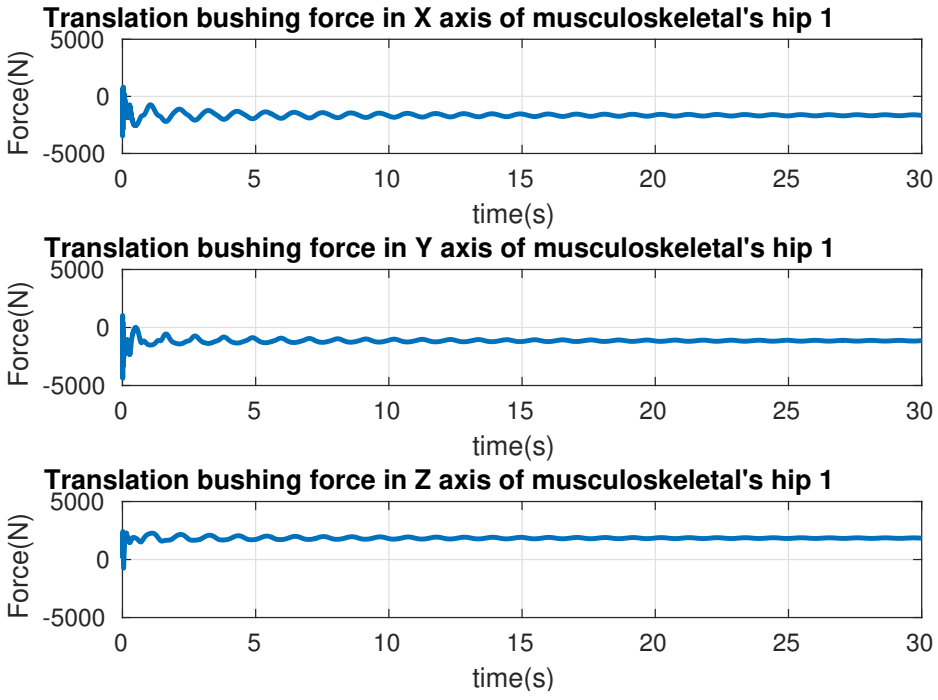


(b) Bushing torque of musculoskeletal's shin 2

Figure 8.18: Bushing torque of exo's and musculoskeletal's shin 2

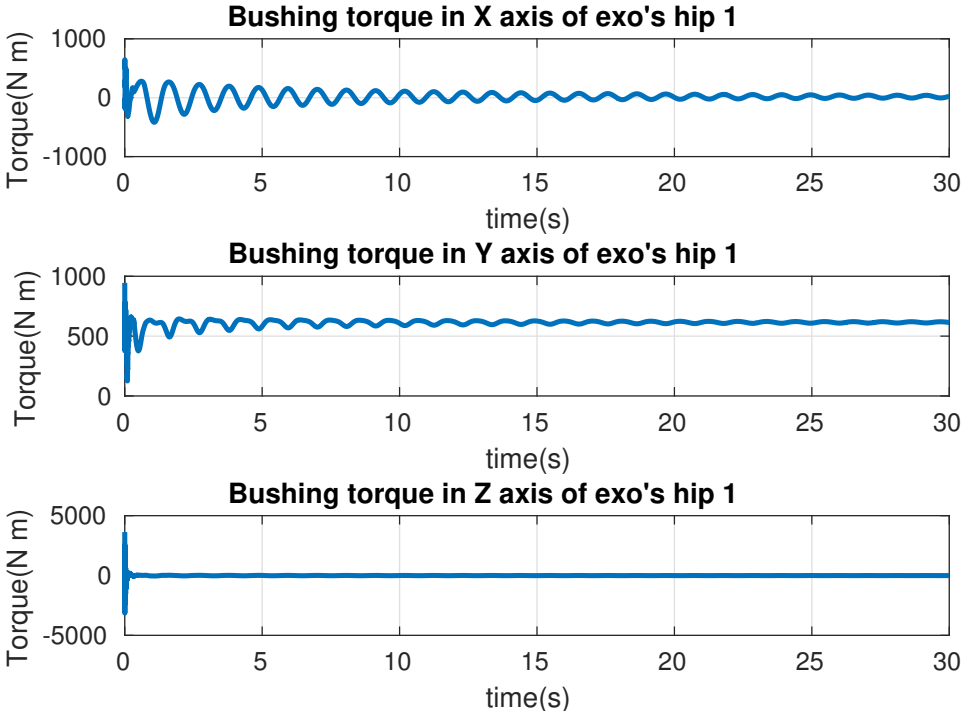


(a) Translation bushing forces of exo's hip 1

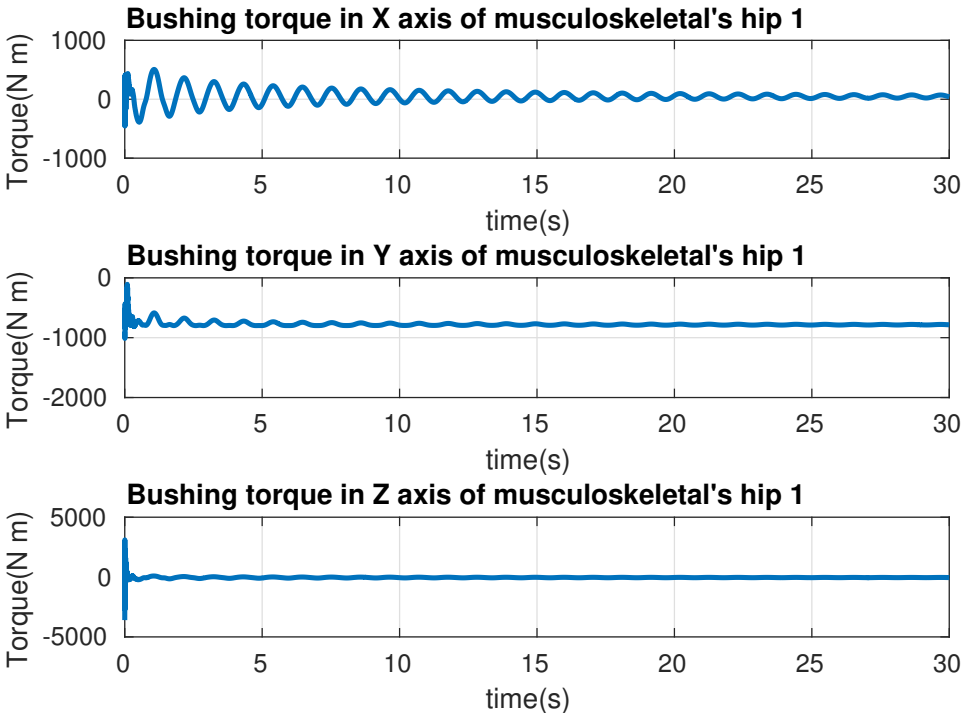


(b) Translation bushing forces of musculoskeletal's hip 1

Figure 8.19: Translation bushing forces of exo's and musculoskeletal's hip 1



(a) Bushing torque of exo's hip 1



(b) Bushing torque of musculoskeletal's hip 1

Figure 8.20: Bushing torque of exo's and musculoskeletal's hip 1

References

[1] World Health Organization. *Disability and Health*. [ONLINE]. Available at: http://www.who.int/mediacentre/factsheets/fs352/en/

[2] World Health Organization, *World Report On Disability*. 2011.

[3] Magdo Bortole. *Design and Control of a Robotic Exoskeleton for Gait Rehabilitation*

[4] P.M. Lukehart, "Algorithm 218. Kutta Merson" *Comm. Assoc. Comput. Mach. ,* 6 : 12 (1963) pp. 737–738

[5] OpenSim Standford. *OpenSim Community*. [ONLINE]. Available at: http://opensim.stanford.edu/about/people.html

[6] *Simbody and Molmodel User's Guide*. Release 2.2, June, 2011.

[7] *Simbody Advanced Programming Guide*. Release 2.2, June, 2011.

[8] SimTK Confluence. *Simulation-based soft exosuit design*. [ONLINE]. Available at: https://simtk-confluence.stanford.edu/display/OpenSim/Simulation-based+soft+exosuit+design

[9] John J. Craig, *Introduction To Robotics Mechanics and Control, Third Edition*

[10] Richard M. Murray, Zexiang Li, S. Shankar Sastry, *A Mathematical Introduction to Robotic Manipulation*

[11] B. Noble,*Applied Linear Algebra*, Prentice-Hall, Englewood Cliffs, NJ, 1969.

[12] Domingo Biel, *Fundamental Lyapunov Theory*, Non Linear Control Systems.

[13] Freeman, Randt A.; Petar V. Kokotović (2008). *Robust Nonlinear Control Design* (illustrated, reprint ed.). Birkhäuser. p. 257. ISBN 0-8176-4758-9. Retrieved 2009-03-04.

[14] SimTK-Confluence Standford. 2012. *Model leg6dof9musc description*. [ONLINE]. Available at: https://simtk-confluence.stanford.edu:8443/display/OpenSim/The+Strength+of+Simulation%3A+Estimating+Leg+Muscle+Forces+in+Stance+and+Swing

[15] Grabcad. 2015. *3D Exoskeleton Model*. [ONLINE]. Available at: https://grabcad.com/library/exoskeleton-5

[16] Second order system response, Michigan State University. http://www.egr.msu.edu/classes/me451/jchoi/2007/handouts/ME451_S07_lecture17.pdf

[17] Haag, Michael (June 22, 2005). *Understanding Pole/Zero Plots on the Z-Plane*. Connexions. Retrieved January 24, 2010.

[18] SimTK-Confluence Standford.*Simbody™ API Reference Documentation*. [ONLINE]. Available at: https://simtk.org/api_docs/simbody/api_docs33/Simbody/html/index.html

[19] SimTK-Confluence Standford. 2012.*OpenSim 3.3 API Documentation*. [ONLINE] Available at: https://simtk.org/api_docs/opensim/api_docs/index.html

[20] Sharcnet.AS IP, Inc. *Body Joint*. [ONLINE]. Available at: https://www.sharcnet.ca/Software/Ansys/17.0/en-us/help/wb_sim/ds_Joints_types_bushing.html

[21] Recurdyn. FunctionBay, Inc. . [ONLINE]. Available at: http://support.recurdyn.com/types-recurdyn-forces/?ckattempt=1

[22] SolidWorks Corporation. 2017. *SolidWorks Main Page*. [ONLINE]. Available at: http://www.solidworks.es/

[23] FreeCad. *FreeCad Main Page*. [ONLINE]. Available at: https://www.freecadweb.org/

[24] didix21, Dídac Coll, *Exoskeleton in OpenSim*, 2017, GitHub repository, https://github.com/didix21/opensim_exoskeleton.git