University of Liège - Faculty of engineering

Master thesis

Simulation of complex actuators

Author : Hubert Woszczyk
Promotor : Prof. Bernard Boigelot

Master thesis conducted for obtaining the Master's degree in
Electrical Engineering by Hubert Woszczyk

Academic year 2015-2016

Simulation of complex actuators

Hubert Woszczyk, under the supervision of Prof. Bernard Boigelot

Academic year 2015-2016
Faculty of Applied Sciences
Electrical Engineering

Abstract

The word *robot* has been crafted by Czech writer Karel Čapek in the beginning of the 20th century and is derived from the slavic world *robota* which means *work*, as in *there is work to be done*. Much has changed since those days, and this master's thesis is about robots who prefer to play football rather than work in factories. This manuscript is the result of the planned participation of a team of students to the *RoboCup* contest. A team of humanoid robots needs to be built and with several design choices still open we cannot begin building a prototype. To avoid time consuming real-life experiments, a simulation tool able of modelling the interaction of a robot with a physical world is needed. We will take a survey of the existing simulators and choose the one that fits our needs best before using it to perform some basic simulations on the model of our robot. These simulations are used to detect design flaws that made the robot unable to stand up from a prone position or unable to walk. The end result of this work is the design of a robot that is able to stand, walk and stand up when toppled over. Furthermore, this report serves as confirmation of various design decisions that were made beforehand such as the choice of the MX-28R as the servomotor to be used in the robot's joints.
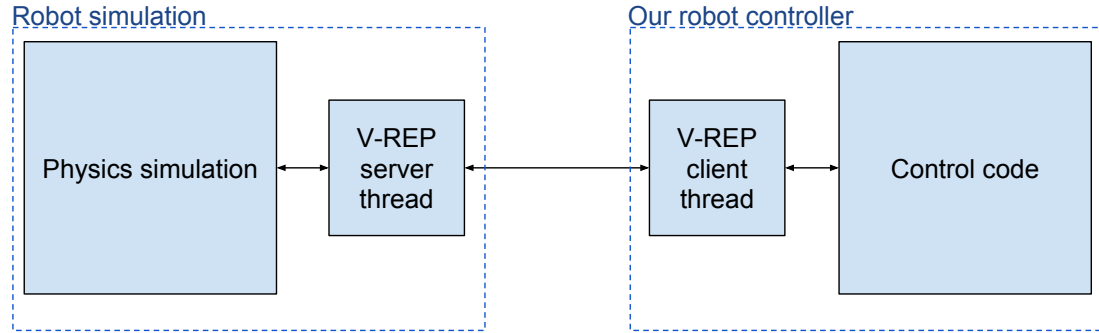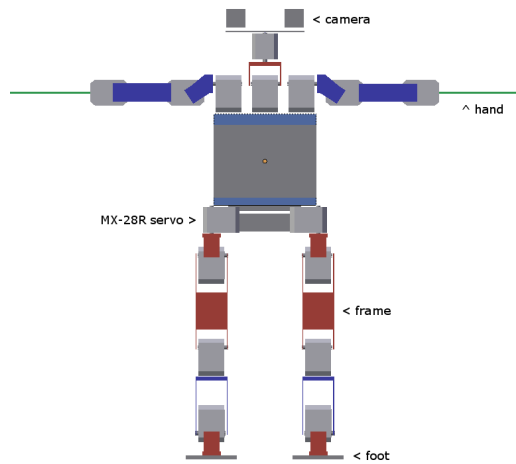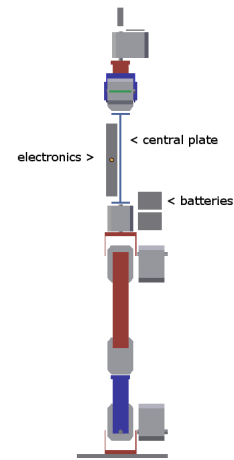
Figure 1: Representation of the architecture of the simulation setup. While the physics simulation is done in a simulator (V-Rep) along with the local control of the servos, the higher level control algorithms are executed outside. The communication between the simulator and high level control code is based on a TCP socket.

(a) Front view of the robot

(b) Side view of the robot

Figure 2: Front and side views of the final robot design this master thesis produced. The servos (in grey) are connected together through different types of frames (in red and blue). At the top, two cameras are discernible and at the centre, the electronics and batteries can be seen.

Acknowledgements

My first thanks go to Prof. Bernard Boigelot who made it possible for numerous students, including me, to work in the passionate field of robotics. I also wish to thank him for his guidance, help and accessibility.

I am deeply grateful to Delphine and Laurine for reading and correcting this manuscript. I also want to thank Grégory and Guillaume with whom I had the pleasure of working together one last time.

Finally, I would like to express my sincere thanks to all those who helped me complete this master thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

For the last ten years, students from the Montefiore institute have been participating in a robotic contest named *Eurobot*, a competition in which wheeled robots battle each other for points in various play environments. After some success and following a thirst for new challenges, it was decided to move on to another contest, *RoboCup*.
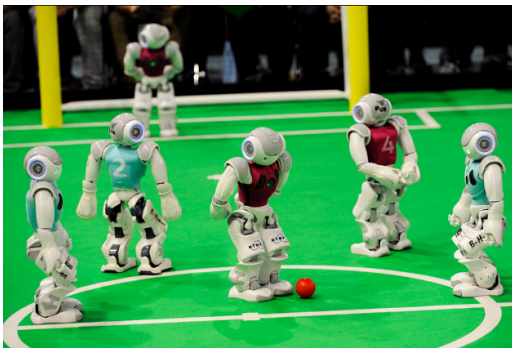
(a) Two teams of Nao robots playing against each other in the 2014 edition of RoboCup Soccer standard platform league. *[Photo courtesy of RoboCup]*
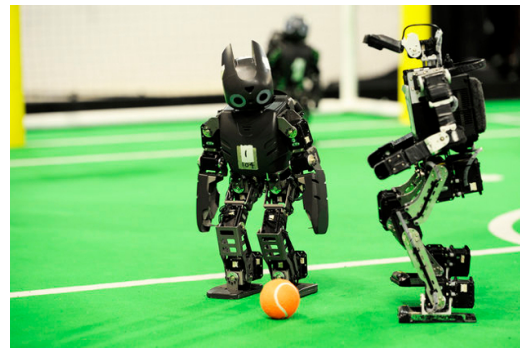
(b) Two robots of opposing teams looking at the ball, in the 2013 edition of RoboCup Soccer kidsize league. *[Photo courtesy of RoboCup]*

Figure 1.1: Robocup standard and kidsize leagues

This contest is quite vast and, as of 2016, is divided into several categories (called domains in RoboCup jargon):

– RoboCup Rescue: as the name suggests, a domain where robots must perform various rescue operations in diverse scenarios.

– RoboCup Industrial: a category with industrially oriented competitions.

– RoboCup@Home: centred around domestic robots, such as robotics helpers for the elderly or robotic butlers.

– RoboCupJunior: more of an initiative that aims to foster robotics interest in children rather than a contest, it helps organize various robotics events for younger minds.

– RoboCup Soccer: historically the first category, centred about humanoid robots playing football. The objective of this category is to have a team of robots beat the world champions by 2050. This is the category we will compete in.

RoboCup Soccer is further subdivided into 4 sub-categories called leagues:

– Standard platform, where the teams all use the same robot, *Nao*, as illustrated in Figure 1.1a.

– Simulation, a league that does not feature physical robots but focuses on team strategies and artificial intelligence. The matches take place in 2D or 3D simulators.

– Adultsize, for the taller robots.

– Teensize, for middle sized robots.

– Kidsize, for the smaller robots. Figure 1.1b shows a match in progress from that league.

This year's team is preparing to participate to the Kidsize league and this master's thesis, along with two others, is the by-product of that team's activity. Since this is our first time participating we have no experience regarding humanoids robots. Therefore, to avoid spending countless hours building and testing different designs we need a tool able of simulating a robot model and its interactions with a physical world.

1.2 Goals of the project

The goal of this thesis is to provide the team with a physics simulating tool with the following features:

– realistic simulation of the physics of rigid bodies. This means that the tool should handle inertia, collisions, friction and constraints between objects. Simulation of springs and dampers is an interesting bonus.

– receive and process orders incoming at a relatively high frequency. The processing need not be in real-time.

– the model of our robot should receive the same orders as the real robot would. That is, the simulator should provide the same interface to the control code as the real robot would.

– 3D visualization of the simulation.

That simulator will be used to:

1. Test different robot designs and choose the best one, in a more efficient way than it could be achieved by physically building the designs.

2. Speed up development and testing of the control code because multiple teams will be able to work in parallel.

1.3   Structure of the report

This report begins with an overview of the basic concepts behind physics simulation on computers in chapter 2. We then move on to chapter 3 that explains the problem we want to solve and motivates the choice of V-Rep as the main simulation tool for this project.

Chapter 4 is about the modelling of our humanoid robot in preparation for chapter 5 to go into the core of the subject with some simulations. Chapter 5 also explains how our work influenced the design of the robot.

Chapter 6 concludes this work by summing up what is achieved and laying out future prospects.

Chapter 2

Principles of interactive rigid body simulation

This chapter briefly introduces the basic concepts relative to physics engines, such as collision detection and constraints modelling. The purpose is to have the necessary background information to make the right choices later on.

2.1 Problem statement

Physics engines are used to simulate classical Newtonian mechanics in a computer. They model how objects accelerate, move and react to collisions with other objects. They also model how objects can be constrained to each other, for example with a hinge, and how it affects their movements. We restrict ourselves to the simulation of rigid bodies, which is a significant simplification of the problem since rigid bodies are idealized solid objects which never change shape.

2.2 Major components of a physics engine

The section is a summary of Jan Bender, Kenny Erleben, Jeff Trinkle and Erwin Coumans' work in [BETC12]. We will explain the key elements of rigid body physics simulation that we need to know in order to orient our work in the later chapters: collisions, the physics model and how it is integrated in time to produce a simulation.

2.2.1 Collision detection

Collision detect is broken into three phases called the *broad phase*, the *mid phase* and the *narrow phase*, as represented in Figure 2.1.

During the broad phase, objects are approximated by simple geometric primitives as distances between such shapes are easy to compute. It is common to use spheres: if the distance between them is greater than the sum of their radii then the objects most certainly do not collide.
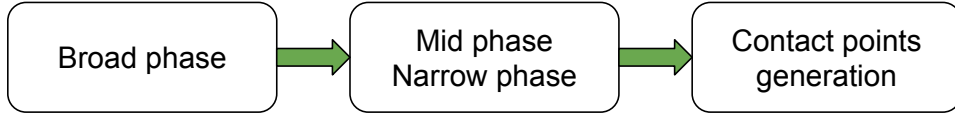
Figure 2.1: Modular description of the collision detection in a physics engine. The mid and narrow phases are grouped together because they are often combined for performance reasons.

When an object has a complex shape, an additional phase called the mid phase separates the object into several simpler shapes to detect collisions. Finally the narrow phase uses the exact geometries of the object to find the contact points. These are then used in the collision resolution part of the simulation loop.

We do not need to dwell on collision detection longer than we already have. For our foreseen application it is sufficient to know that simpler shapes are easier to handle. For further discussion, different possible implementations are detailed in [JTT01].

2.2.2 Collision resolution

When bodies collide, high forces of very short duration are exerted. In the case of rigid bodies the duration is infinitesimal and as a consequence the forces become infinite. This is problematic because the simulator integrates forces to obtain velocities and positions: an infinite force would break the simulation loop.

A solution to this was proposed by Brian Mirtich in [Mir96]. The idea is to use the standard integration rule of the simulator up to the collision, use an impulse-momentum based collision rule to determine the velocities after impact and then resume the integration.

Several collision rules exist. One of them is called Newton's Hypothesis and it states that

$$\mathbf{v}_n^+ = e\mathbf{v}_n^-$$

where $\mathbf{v}_n^+$ is the relative normal velocity after impact, $\mathbf{v}_n^-$ is the relative normal velocity before impact and $e \in [0, 1]$ is called the coefficient of restitution. When $e = 1$ the collision is perfectly elastic and when $e = 0$ all the energy is lost.

2.2.3 Physics model

The model is based on the three laws of Newton:

– The velocity of an object remains unchanged if no force act upon it.

– The rate of change of the momentum of an object is equal to the force acting on it.

– For every force there is an equal and opposite force.

Physics engines usually represent positions, orientations and velocities in an inertial frame, which is a requirement for the laws of Newton to be true:

- Position is given by a vector $\mathbf{p} \in \mathcal{R}^3$, to represent the three translational degrees of freedom (DOF).

- Orientation can be represented either by rotation matrices or unit quaternions, the latter being usually preferred. A unit quaternion is four numbers $[Q_s, Q_x, Q_y, Q_z]$ constrained so that the number of their squares is one. Q_s is computed in terms of the other numbers, so we correctly represent the three rotational DOF.

- Translational velocity is given by a vector $\mathbf{v} \in \mathcal{R}^3$. It should be noted that when the body rotates this vector only describes the translational velocity of the reference point of the body, which is usually chosen to be the centre of mass.

- Rotational velocity is given by a vector $\boldsymbol{\omega} \in \mathcal{R}^3$.

We define $\mathbf{q} = [\mathbf{p}^T, \mathbf{Q}^T]$ as the tuple containing the position of the COM and the orientation of a rigid body and $\mathbf{u} = [\mathbf{v}^T, \mathbf{w}^T]$ as its generalized velocity. When using quaternions, the velocity kinematic equations of a rigid body, i.e. the equations that relate $\mathbf{q}$ to $\mathbf{u}$ are:

$$\dot{\mathbf{q}} = H\mathbf{u} \tag{2.1}$$

$$H = \begin{pmatrix} \mathbf{I}_{(3\times3)} & 0 \\ 0 & \mathbf{G} \end{pmatrix} \tag{2.2}$$

$$\mathbf{G} = \frac{1}{2} \begin{pmatrix} -Q_x & -Q_y & -Q_z \\ Q_s & Q_z & Q_y \\ -Q_z & Q_s & Q_x \\ Q_y & -Q_x & Q_s \end{pmatrix} \tag{2.3}$$

$$\tag{2.4}$$

where $\mathbf{I}_{(3\times3)}$ is the 3-by-3 identity matrix.

Applying the second law of Newton to both translational and rotational forces produces what are called the Newton-Euler equations:

$$m\dot{\mathbf{v}} = \mathbf{f} \tag{2.5}$$

$$\mathbf{I}\dot{\boldsymbol{\omega}} + \boldsymbol{\omega} \times \mathbf{I}\boldsymbol{\omega} = \boldsymbol{\tau} \tag{2.6}$$

where $\mathbf{I}$ is the *inertia* of the object. It represents the tendency of an object to rotate around an axis. $\boldsymbol{\tau}$ is the *torque* and represents the tendency of a force to rotate an object around an axis. It is defined as the cross product ($\times$) of a force vector and the vector of the distance between that force and the point of fixation of the rigid body. These equations express the relationship between forces acting on a rigid body and its velocity.

Equation (2.1), Equation (2.5) and Equation (2.6) are the basis of the physics model. Alone, they are sufficient to model the behaviour of a free falling rigid body. To handle collisions and constraints we must add equations that express different types of constraints : non-penetration of bodies, friction forces act in the direction that will most quickly stop the sliding and many others.

All these equations form a differential non-linear complementarity problem (dNCP) that cannot be solved in closed form. It is thus discretized in time producing a series of non-linear complementarity problems (NCPs) whose solutions are

an approximation of the state of the system. These solutions are usually found by linearising the NCP into a linear complementarity problem (LCP) to take advantage of the rich background for that type of problems.

2.3   Simulation loop

Figure 2.2 describes the main loop of a physics engine. A simulation step begins with the detection of collision points (Collision detection) which are then used to correct the positions of bodies to prevent interpenetration (Collision resolution) before being used to derive constraints to be incorporated in the complementarity problem (Contact handling), along with other active forces (gravity, or a motor generating torque).

Finally, the CP is solved (Time integration) to determine the positions and velocities of the rigid bodies at the next time step and the loop begins anew.
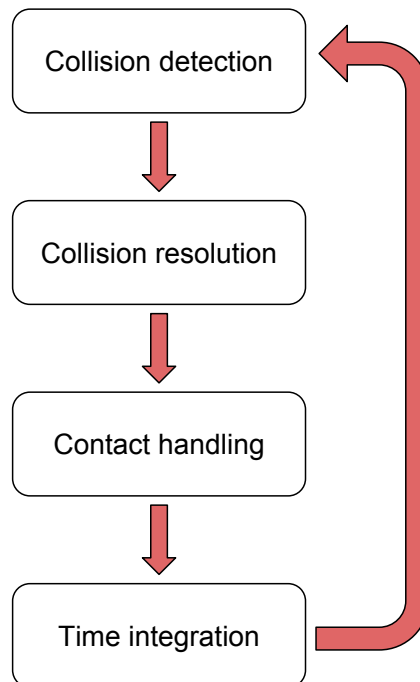
Figure 2.2: Modular phase description of the sub tasks of a rigid body simulator. Source: [BETC12]

Chapter 3

Choice of the simulation platform

In this chapter we define exactly the kind of concepts and objects we want to represent in our simulation tool before making a survey of the existing simulation software. We then motivate our choice.

3.1 Problem statement

Our robot will consist in a a number of servos connected together through frames in addition to a central plate that will contain the electronics. We also plan to use spring-dampers in the legs to better mimic the human movement an, what is more interesting to us, walk in a more energy efficient way.

If we want to simulate it we thus need a tool that supports:

– inertia, to have physically accurate dynamics.

– joints, to have constraints between objects.

– collision and friction, to accurately model the interaction of the feet with the ground.

– spring-dampers.

– remote control of the simulation. This is a critical requirement as the goal is to use the same code to control the robot in real-life and the model in the simulation. By control code we mean the high-level code that decides which angles the servos should target, amongst other tasks.

The servo used in the robot will have several sensors such as accelerometers, rotary position encoders. We want to model them in the simulator since they will be used by the high-level control code and one of our objectives is the ability to test the latter on the simulator as if it was the real robot.

We will also have cameras on the robot, so the ability to simulate them would be a plus, still in the mindset that the simulator should be able to substitute itself to the real robot.

3.2 Barebone physics engines

The physics engine is the cornerstone of a physics simulation tool. There exist a quantity of them, we present here the most popular ones that have the following features' set: multi rigid body dynamics, collision detection, joints (hinge, spring-dampers, generic 6DOF).

1. **Bullet**[1]**:** As of now, the most popular open source physics engine. Although primarily used in video games it is also used in applications such as Blender, V-Rep or NASA's tensegrity robotics toolkit.

2. **Newton**[2]**:** Another open source engine, not quite as popular as Bullet and ODE but nevertheless used for commercial games and simulation.

3. **ODE**[3]**:** Open source, a little older that Bullet. As it was already mature when simulators started to be developed, it is present in a lot of robotics simulation tools (V-Rep, Webots, Gazebo...). It was also designed with games in mind but was influenced by its success in more serious applications.

4. **PhysX & Havok:** Both are proprietary engines used primarily for games. They won't be further discussed because their focus is on speed rather than accurateness and as such they cannot be used in a simulation that aims to be realistic, as stated by Erez *et al* in [ETT15].

3.3 Simulators

In this section we will discuss software that provide a higher level interface to the physics engines we presented earlier. Simulators integrate physics engines and add several higher level functionalities on top of them: visualization, modelling, scripting, etc.

1. **Blender**[4] is a 3D modelling software suite and as such has integrated the Bullet engine to help make more realistic animations. It features the ability to make Python scripts that use that engine to make games or physics simulations.

   It is open source and cross platform.

2. **Gazebo**[5] was the official simulator of DARPA's Robotics challenge. It features multiple physics engines (Bullet, Simbody, Dart and ODE), allows custom plugins and uses the SDF format for its models.

   It is open source but binaries are not provided for Windows and OSX.

3. **V-Rep**[6] is another simulator that lets you choose the physics engine(Bullet, ODE, Newton, Vortex) and it also allows custom plugins in the form of LUA

---

[1]http://bulletphysics.org/wordpress/
[2]http://newtondynamics.com/forum/newton.php
[3]https://bitbucket.org/odedevs/ode/
[4]https://www.blender.org/ [Accessed 21/05/2016]
[5]http://gazebosim.org/ [Accessed 21/05/2016]
[6]http://www.coppeliarobotics.com/ [Accessed 21/05/2016]

scripts. It uses its own format for storing models but can import standard formats(COLLADA, 3ds, etc...).

It is cross-platform and free to use for educational purposes.

4. **Webots**[7] has virtually the same features as V-Rep.

It is cross platform but not free.

5. **Matlab** is not a dedicated robotics simulator *per se* but can be used to model the robot analytically and to write simulation code for it.

3.4 Tested software

In this section we try some of the proposals presented previously and give our thoughts on them. We will mainly look at the modelling facilities and the access to the underlying simulation options as all the proposed tools possess the required simulation capabilities and are able to deliver similar looking results.

3.4.1 Blender

Blender is convenient to use because the robot's model can be easily changed inside. Furthermore, the fact that the scripting language is Python make code development faster and the latter's support of TCP sockets allows an external program to control the simulation and the robot inside it. The internals of the physics are obscured and some interesting object properties, such as inertias, are hard to reach. It is also hard to change the simulation parameters making it difficult to obtain stable results when using a higher number of objects and constraints. Furthermore, support for the game engine, the basis of a simulation project, is uncertain, as stated in the development roadmap [Ble15].

Pros:

– Easy modelling of the elements.

– The Python API is well documented.

Cons:

– Bullet's simulation parameters are hidden behind an incomplete interface, making it impossible to modify the timestep or the number of iterations of the LCP solver.

– Furthermore, the version Blender uses is an old one and lacks many improvements in the handling of constraints.

– Inertias are approximated by the principal values Ixx, Iyy and Izz.

---

[7] https://www.cyberbotics.com/ [Accessed 21/05/2016]

3.4.2   Gazebo

Gazebo is attractive because it has the support of DARPA and handles multiple physics engines. The main drawback lies in the modelling abilities. It does feature an internal modelling tool but it is too limited to be usable. That would not be a problem if it could import models easily but that is not the case: it uses a format called Unified Robot Description format(URDF) which is a xml based storage format. The problem is that the only tool that can export models in that format is Solidworks[8], a commercial product. The team behind Gazebo seems to be well-aware that this is an issue since as of may 2016 it is focusing on developing an internal model editor.

Pros:

– Choice of the physics engine.

– Inertias definable as a matrix.

– Friction represented in physical values.

Cons:

– Robot model must be in URDF, making it hard to iterate over robot designs as each model would need to be created by hand.

– Internal model editor prohibitively limited, cannot be used to create a complicated model.

3.4.3   V-Rep

V-Rep also has multiple physics engines available and has a user-friendly interface. It also has an internal modelling tool but there is not much use for it since it allows the import of models in the COLLADA format. Moreover, it supports TCP sockets and even provides code for a client thread in the custom application. The options of the physics engines are also pretty accessible and lots of sensor types are natively supported by the simulator.

Pros:

– Choice of the physics engine.

– Inertias definable as a matrix.

– Friction represented in physical values.

Cons:

– Limited internal editor, can hardly be used for modelling.

---

[8]<http://www.solidworks.com/>

3.5 Choice

3.5.1 Simulator

We make the choice of using a simulator rather than just a physics engine for several reasons:
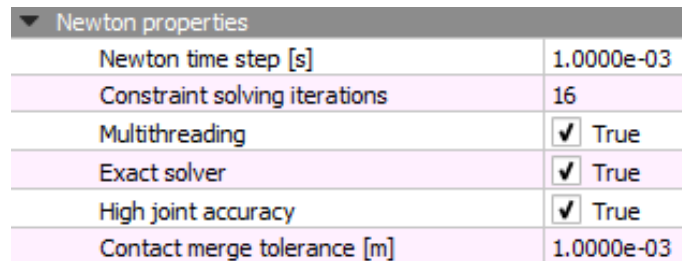
– The product of this master's thesis is expected to be used by other students next year and they would rather spend as little time as possible learning how to use it. Hence an established simulator rather than an in-house one is preferred.

– A simulator eliminates the need of developing 3D visualization by ourselves.

– A simulator already provides much needed features as the import of 3D models or an interface to the settings of the simulations.

– Most simulators have an already defined API for remote control.

– By using an existing simulator we can request help from other users.

From the simulators we surveyed earlier in this chapter the best overall seems to be *V-Rep*. This choice is further confirmed by Ivaldi *et al.* in [IPPN14] who shows that V-Rep is one of the highest rated tools amongst roboticists.

3.5.2 Physics engine

Inside V-Rep, we chose *Newton Dynamics* as the physics engine because simple tests showed it to be the most stable with a high number of joints. That choice is further confirmed by Hummel *et al.* in [HWS+12] where Newton Dynamics is stated to be the best engine when it comes to handling a high number of constraints.

We will be using the options that are shown in Figure 3.1

Newton properties	
Newton time step [s]	1.0000e-03
Constraint solving iterations	16
Multithreading	✔ True
Exact solver	✔ True
High joint accuracy	✔ True
Contact merge tolerance [m]	1.0000e-03

Figure 3.1: Simulation properties of Newton Dynamics. What is not shown in this figure is the at which the simulator renders the simulation, which is 100Hz.

3.5.3 Modelling software

Although *Blender* was not chosen as the primary simulation tool for the project, it shall be used as a modelling tool for the robot. The primary reason is that we are already familiar with and we do not need to create elaborate meshes.

Chapter 4

Modelling a robot

This chapter focuses on the modelling of the atomic elements of the robot. We begin by explaining how to create the model of an object inside V-Rep. We then show how to use to model simple elements such as frames or electronics who are not active during the simulation i.e. they do not perform any function. Those can be represented strictly mechanically. We then show how to model active pieces such as the cameras of the servos before finally presenting a complete model of a robot.

4.1   Problem statement

Our robot will be made from a number of elements that all need to be represented in the simulation:

  – **Miscellaneous mechanics and electronics:** A type of element that must be included in the model are the frames, Figure 4.1b shows the FR07-H101 frame. On the actual robot, they will link the servos together. Other mechanical elements are the hands, the feet and the plate that will act as the trunk of the robot.

    In addition to all the aforementioned elements we plan to equip the legs of the robot with springs. Their purpose will be to make the robot walk more efficiently by storing energy in these springs when the foot hits the ground and releasing it when the leg takes another swing.

    The robot will also have an array of electronic elements. An Intel Atom processor (Figure 4.1d) to perform the high-level control (giving target angles to the servomotors, computing trajectories, processing the inputs from the cameras, etc), batteries to stock the energy necessary to power them and the electronics to convey that energy through the robot.

  – **Cameras:** The robot will be equipped with two cameras of the type shown in Figure 4.1c. Situated at the top of the robot, it will use them to locate itself and points of interest in the play arena[1]. As we may test some machine vision algorithms during our (future) simulations we need to have the ability of retrieving the image or a video stream a camera captures.

---

[1]This is the subject of an on-going master thesis at Montefiore.

– **Servomotors:** We will be using the MX-28R servo, shown in Figure 4.1a, as the driving element of the joints of the robot. We need to represent it as a rigid body as well as its function as a mechanical device that can generate torque on its axle to keep a targeted angle (hence the name *servo*, from the Latin word for *slave*).

(a) MX-28R servo.

(b) FR07-H101 hinge type frame.

(c) LI-USB30-M021C camera.
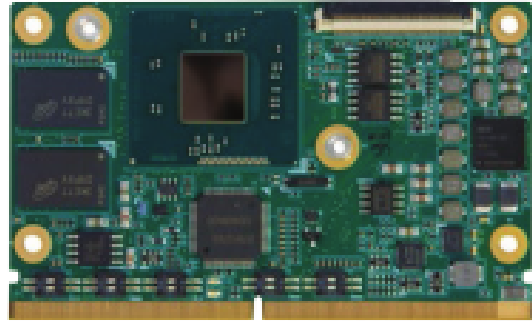
(d) SMARC Short Size Module.

Figure 4.1: Atomic elements of our robot. The MX-28R and the camera are more complex to model as they each perform a function in the simulation. On the other hand, the frame and the motherboard are only dead weight.

4.2 Modelling in V-Rep

This section is a short introduction to the creation of a dynamic model of an object in V-Rep. We present how to set its mass, inertia and friction and how to make it collide with other models and respond to forces. Later, we present how to create constraints between models and explain why it is useful.

4.2.1 Creating the model of an object

The creation of a rigid body inside V-Rep begins with the creation of a mesh[2] in a modelling tool which in our case is Blender. It is preferred to have a convex mesh, i.e. one that whose all interior meshes are less or equal to 180°, as they are easier for the simulator to handle. When it is absolutely necessary to keep the mesh concave, a solution is to separate it into several convex meshes that will be grouped together in V-Rep. Once saved in a file format recognized by V-Rep the mesh can be imported into the latter.
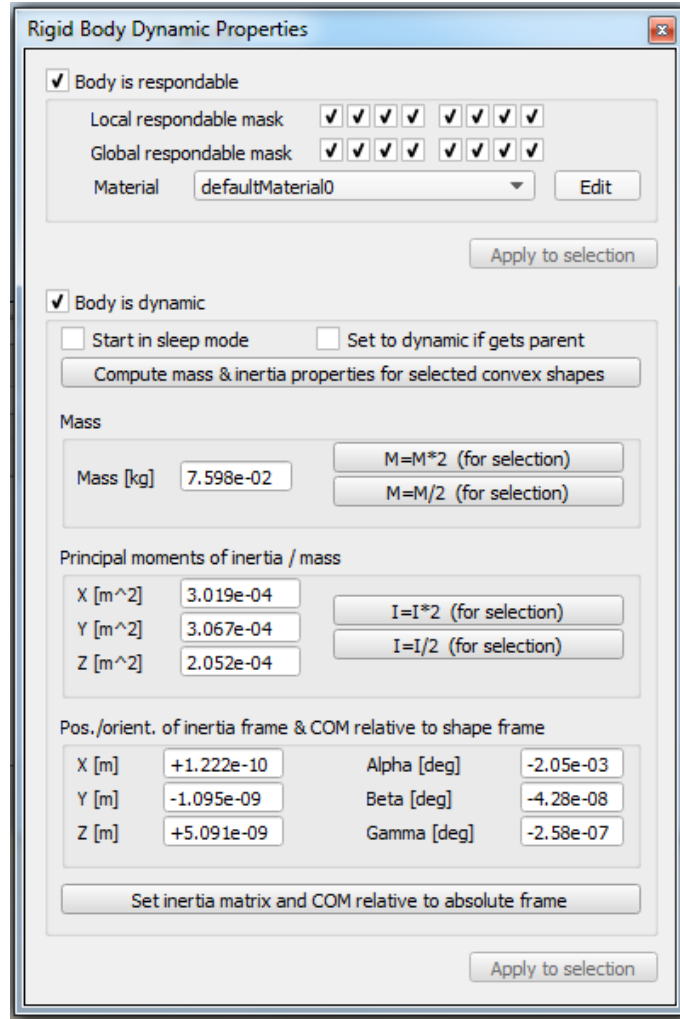
Figure 4.2: Configuration window of the dynamic properties of the selected rigid body. We can make it *respondable*, i.e. it should collide with other objects. We can also make it *dynamic* in which case it will be dynamically active in the simulation and react to forces. When dynamically active, the movement of this body will be influenced by its *mass* and its *inertia*. It is also possible to set what is called *Material* by V-Rep which is a set of parameters such as the friction of the material or its restitution parameter.

In V-rep, we open the properties of the mesh (the window in Figure 4.2) and mark it as dynamic, respondable (if necessary) and set the mass and the inertia. It is possible to have V-Rep compute it automatically from the geometry and the density but it can also be set manually.

---

[2]an ensemble of vertices and faces that represent an object

4.2.2   Creating constraints between models

In V-Rep the constraints exist under the form of *joints*. There exist three types of joints:

1. **Revolute joints:** they have 1 degree of freedom (DOF) and enforce rotational movements between objects. The range of the rotation around the axis of the joint can be restricted. By default, it is passive but it can be set to *torque/force mode* where it will try to hold a target angle, very much like a servomotor would.

2. **Prismatic joints:** they have 1 DOF and enforce rotational translational movements between objects. As in the case of the revolute type of joint, the range can also be restricted. There also exist a torque/force mode setting but in the case of a prismatic joint its purpose is to represent a spring-damper. The stiffness and damping can be specified in the properties of the joint.

3. **Spherical joints:** they have 3 DOF and enforce rotational translational movements between objects. Spherical joints do not have a torque/force mode and can only be passive.

A very important characteristic of joints is that they must link two dynamic objects together in order to work properly during the simulation. Two joints cannot be directly connected one to another.

4.3   Modelling the miscellaneous mechanical elements and electronics

The modelling of most of the mechanical elements and electronics is rather straightforward but there are some where the process is more troublesome. This is the case of the feet, frames and springs, and those will be detailed hereunder after a short explanation of the modelling process of the simpler pieces.

4.3.1   Modelling the electronics, batteries, hands and the central plate

The modelling of electronics, batteries, hand and central plate consists in applying the procedure explained in Section 4.2. These elements are already pretty simple in reality as they all have cuboid shapes. All their dimensions and weights are in Table 4.1 along with a suggested material in the case of pieces that will have to be custom made.

4.3.2   Modelling the feet

In essence, the feet of our robot will be plates that will be fixed to the last servos of the legs. Contrary to other elements, feet *must* be respondable as they come into contact with the floor. It follows that the friction parameter will be important too

Module	Weight $[g]$	Material	Density $[kg/m^3]$	Dimensions $x[mm] \cdot y[mm] \cdot z[mm]$
Odroid C-2	40	[-]	840	$85.0 \cdot 56.0 \cdot 10.0$
Li-Po battery	188	Li-Po	2304	$103.0 \cdot 33.0 \cdot 24.0$
Hand	30	Aluminium	3000	$70.0 \cdot 31.0 \cdot 31.0$
Feet	52	Polymer	1000	$70.0 \cdot 125.0 \cdot 6.0$
Central plate	209	Aluminium	4000	$140.0 \cdot 30.0 \cdot 124.5$

Table 4.1: Weights and dimensions of the pieces of the robot. The density is useful for the automatic computation of the weight and inertia of the pieces in V-REP. The differences in density between different pieces made of the same material represent the differences in geometry, i.e. pieces with holes in them that are not modelled but taken into account by making the piece lighter.

and it will have to be chosen in accordance to the friction parameter of the material they will be made of.

It should be noted that for the needs of the simulation, their z dimension has been exaggerated up to $6mm$ because of collision problems (sometimes, the contact point was on the upper side of the feet while it should have been on the down side) that disturbed the course of the simulation. Therefore, the z dimension reported in Table 4.1 is the double of the actual one and the density has been halved to keep the same mass.

4.3.3 Frames

By *frame* we mean the mechanical element that is used to connect the axle of a servo to another object which may be another servomotor. The frame in Figure 4.1b is the standard one that will be part of the robot.

Frames are not directly present in the simulation. By that we mean that the link between the servomotors are made through joints without the intermediary of a frame. This reduces the accuracy of the model a bit but not by much. In exchange, we gain in speed and stability by reducing the number of constraints and not having a convex shape in the model of the robot.

4.3.4 Springs

As mentioned in Section 4.2.2 a spring can be represented by a prismatic joint. We link two objects together and set the joint to force/torque mode and we specify the stiffness and damping.

This is not enough if we want to model a spring that is between two elements that should be able to move not only in direction of the axis of the joint. For that purpose we add two spherical joints at the extremities of the prismatic joint. As it is impossible to link two joints together, we must add some intermediary objects between them. They are the objects called *spring_anchors* in Figure 4.3b.

Another problem we have to solve is that we have to create a loop inside the

hierarchy of the model in order to connect a spring between two elements in a leg. V-Rep provides the ability of closing loops like this through *dummy* objects (visible in Figure 4.3b). A pair of objects like these can be ordered to keep the same position.

Now, in the case of the spring in Figure 4.3, we can attach it to a leg by moving it to the desired position and by making the *Spherical_joint5* a child of the element that is higher in the leg and *Dummy4* a child of the element that is lower in the leg.

(a) Visual representation of a spring in V-Rep.
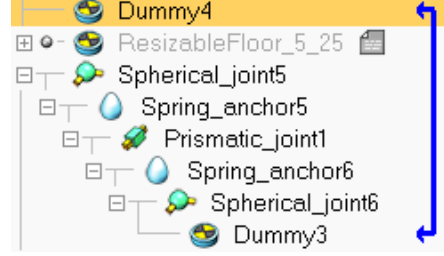
(b) Hierarchy of the elements constituting the spring on Figure 4.3a.

Figure 4.3: Representation of a spring that can connect two elements of the same leg.

4.4 Modelling the cameras

As mentioned earlier, our robot will have two cameras at the top of its body. We are currently experimenting with cameras of the type illustrated in Figure 4.1c. As far as the shape goes, the camera are modelled as cubes which dimensions and weights are in Table 4.2.

	Data	Unit
Weight	22	g
Dimensions	26.0 x 26.0 x 14.7	mm^3
Density	2213	kg/m^3

Table 4.2: Characteristics of the LI-USB30-M021C camera

The active part of the camera, capture of videos and images, is simulated by a *vision sensor*. As the name implies, this sensor captures what it sees. It has an array of properties, as shown in Figure 4.4, which influences what it can *see exactly*. Some of them are rather performance related but we have access to the most essential parameters a camera has: the focal length and its resolution.

The possibility of embedding image processing is very interesting as by default the vision sensor captures the exact image of the world it sees. Therefore, a script can modify that image to be closer to what the actual camera would capture. For example, if the camera captured blurry images, a blur filter could be embedded in the vision sensor.
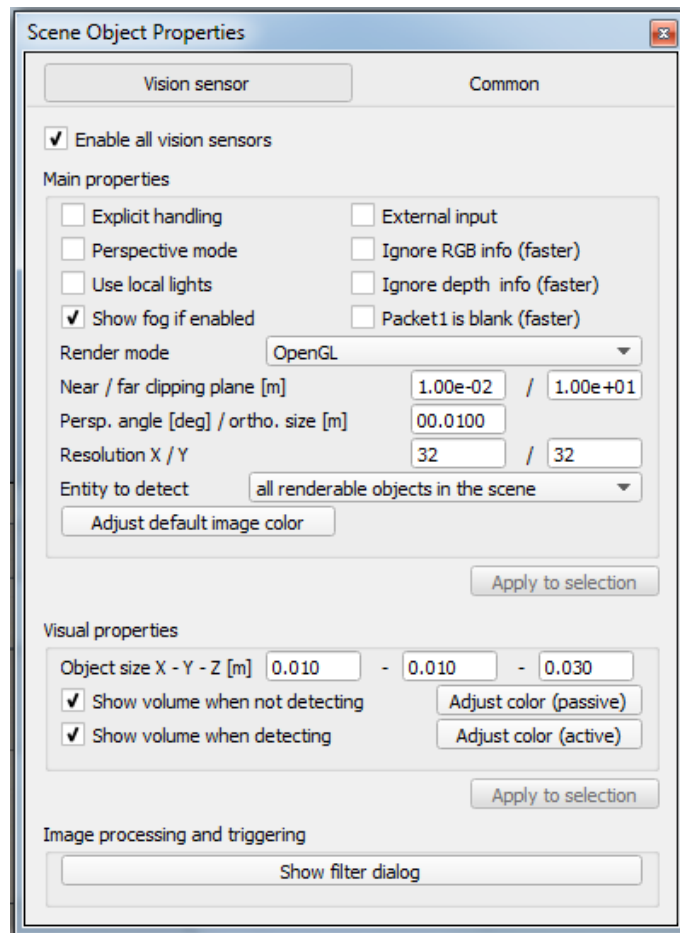
Figure 4.4: Properties of a vision sensor. We can set how far and how close it can see, which can be useful for performance. We can also set its resolution and the focal length (persp. angle [deg]/ortho. size[m] option). The *show filter dialog* at the bottom gives the possibility to embed image processing scripts which could be useful in tuning the output of the camera.

4.5 Characterizing and modelling the MX-28R servomotor

This section explains how the characteristics of the MX-28R servo are determined in order to make a physically accurate model of it. We will experimentally determine its torque and rotation speed.

4.5.1 Determining the continuous torque

The primary parameter we must know the value of to properly model the MX-28R is the torque it generates. The manual ([Dyn16a]) provides the value of the stall torque ($2.5Nm$ @$12V$) and a graph showing the efficiency of the servo at different torques and speeds but it does not provide a value of the actual continuous torque. We thus compute it from the maximal torque of the DC motor and the reduction ratio of the gears.

$$ContinuousTorque = TorqueMotor \times ReductionRatio$$
$$= 3.67e^{-3} \times 193$$
$$= 0.7083Nm$$

The continuous torque is thus equal to $0.7Nm$.

4.5.2 Determining the stall torque

The stall torque is determined through a small experiment with a real MX-28R. The experimental setup is explained in Figure 4.5. The experiment itself is presented in Figure 4.6.

(a) The MX-28 servos are powered by a DC generator and controlled by a laptop equipped with a USB2DYNAMIXEL(USB-2-DYN) device.

(b) A USB2DYNAMIXEL. It turns an USB port into a serial port (RS485, TTL or classic serial connector) that can be used to control Dynamixel manufactured servos. Source: [Dyn16b]
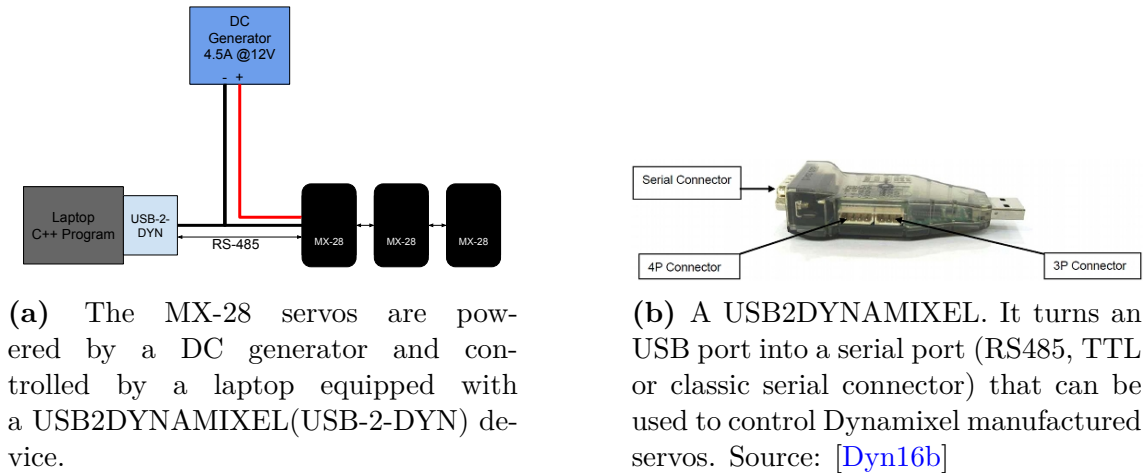
Figure 4.5: Experimental setup

In our case, d was equal to $22.5cm$ and we could reach a weight w of $740g$ at $14.8V$. This equals to a torque of $1.64Nm$. The complete results are in Table 4.3.
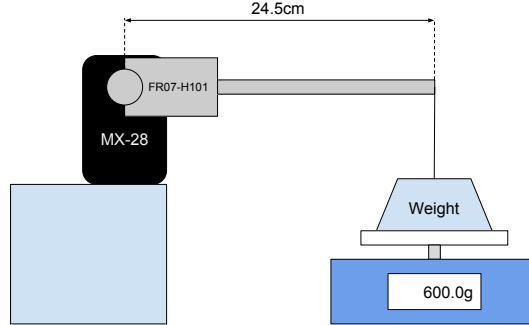
Figure 4.6: Experimental setup for torque testing. A weight w is suspended at distance d from the servo, resulting in an applied torque of $w \times g \times d$. w is augmented until the servomotor is unable to lift the arm at which point we have determined the maximal torque it can generate.
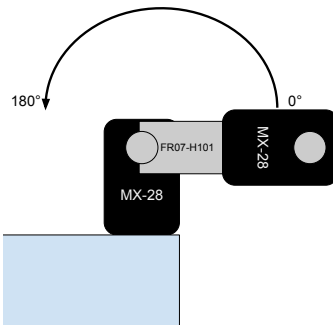
Figure 4.7: Experimental setup for dynamics testing. One servo lifts the other. The goal is the measure the time it takes to swing the arm from 0° to 180°.

4.5.3 Determining the rotation speed

The goal of this experiment is to determine the unloaded rotation speed of the MX-28R. The setup is shown in fig. 4.7. The measuring is done by taking advantage of the rotary encoder inside the servomotor: we can measure the time it takes for the servo to go from 0° to 180° through a program.

The result is that executing this manoeuvre at $12V$ with no imposed speed limit takes $620msec$. This corresponds to a rotation speed of 48.33 rotations per minute (Rpm).

4.5.4 Results

The results of all the previous experiments along with the characteristics of the MX-28R servomotor are in Table 4.3.

|	Data	Unit
Weight	77	g
Dimensions	$35.6 \times 50.6 \times 35.5$	mm^3
Ixx	$22,649$	$g \cdot mm^4$
Iyy	$12,868$	$g \cdot mm^4$
Izz	$17,733$	$g \cdot mm^4$
Announced stall torque @11.1V	2.1	Nm
Experimental stall torque @11.1V	1	Nm
Announced stall torque @12V	2.5	Nm
Experimental stall torque @14.8V	1.2	Nm
Announced stall torque @14.8V	3.1	Nm
Experimental stall torque @12V	1.6	Nm
Nominal continuous torque @12V	0.7	Nm
Announced unloaded rotation speed @12V	55.00	Rpm
Experimental unloaded rotation speed @12V	48.33	Rpm

Table 4.3: Characteristics of a MX-28R servomotor. Data taken from [Dyn16a] and from http://www.robotis.com/view/DXL-INERTIA/RX-28_INERTIA.pdf[Accessed 4/6/2016].

4.5.5 Modelling the MX-28R servomotor

As before, the model creation begins with the drawing of a mesh inside Blender but this time the mesh, in Figure 4.8b, is a bit more complex. It is actually made of two meshes : one for the hull and the other one for the axle. Its purpose is to act as a position marker in V-Rep, where a joint is going to placed in the axis of that axle.

(a) MX-28R servo.
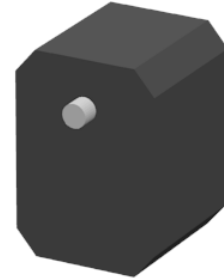
(b) Model of the MX-28R.

Figure 4.8: Side by side of a MX-28R servo and its 3D model. The shape has been simplified but retains outer appearance of the servo. The axis is used as a position marker and will be removed once the joint is in place.

When the joint is in place, we edit its properties to set in torque/force mode and tick both the motor and control loop boxes of the joint dynamic properties window of Figure 4.9. The maximum torque is set to $1.2 Nm$ and the maximum rotation speed to $(48.33 \cdot 360)/60 = 290°/s$. The MX-28R uses a PID controller to reach and hold the target positions so we do not need to write a custom control script and can content ourselves with modifying the proportional and integral parameters.

A PID controller is a feedback based control loop that is quite common in the engineering industry. An example of such a controller is the one that is implemented in the control code of the MX-28R that can be seen in Figure 4.10. The current position is measured and compared to the target position. The difference between those two is called the *error*. The error then follows three parallel paths. In the first one it is multiplied by a *proportional gain K_p*, which accounts for the present error. In the second one, it is integrated and multiplied by an *integral gain K_i* which accounts for the past errors. It is used to counter steady-state errors that are not countered by the proportional error. It will accumulate and the integral error will eventually be high enough to counter that offset. The last one is derived and multiplied by a *derivative gain K_d*. This gain accounts for the possible future values of the error and influences how fast the controller reacts. For more information on PID controllers and their use we please refer to [JM05].
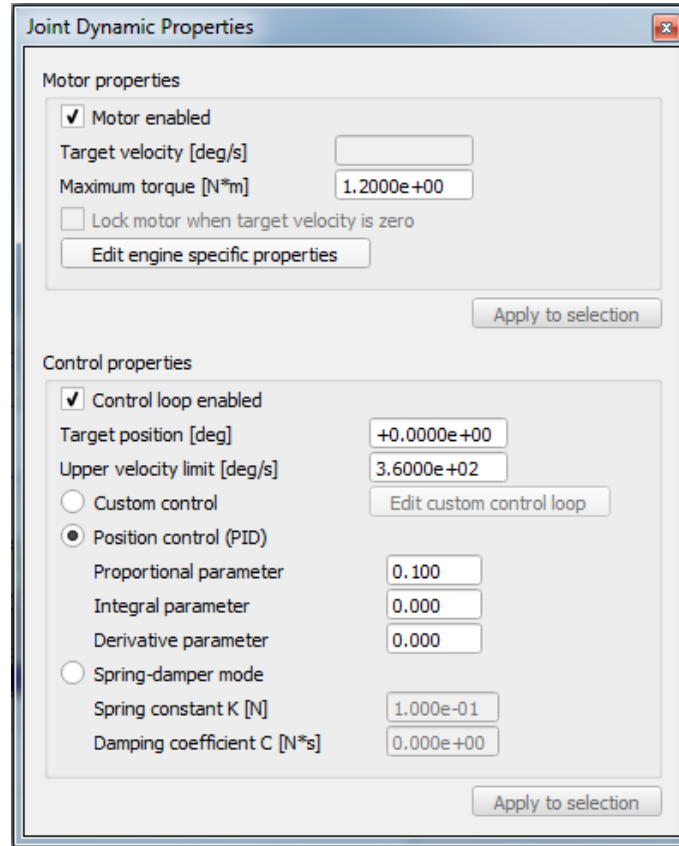
Figure 4.9: Dynamic properties of a revolute joint. When motor enabled we can set a target velocity along with a maximum torque and the joint acts as traditional motor. When the control loop is enabled the join acts as a servomotor and we can decide whether we use a PID controller or a custom control script.
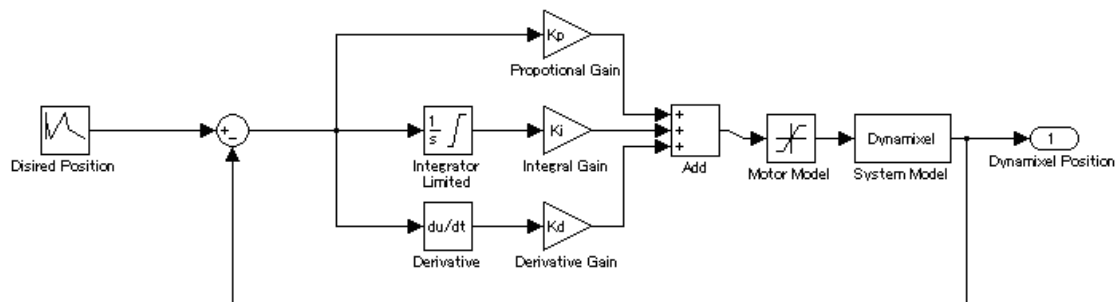
Figure 4.10: Block diagram of the PID controller implemented in the MX-28R. It is a standard PID controller although by default only the P parameter is non-zero. The function of the *motor model* block is to clamp the computed correction torque in the allowed interval whereas the *system model* block is a simulink representation of the MX-28R and does not exist in the actual implementation inside the latter. Source: [Dyn16a]

24

4.6 Robot model

The elements we just modelled can be assembled together to create the robot model in Figure 4.11. The next chapter will focus on simulations involving this model.
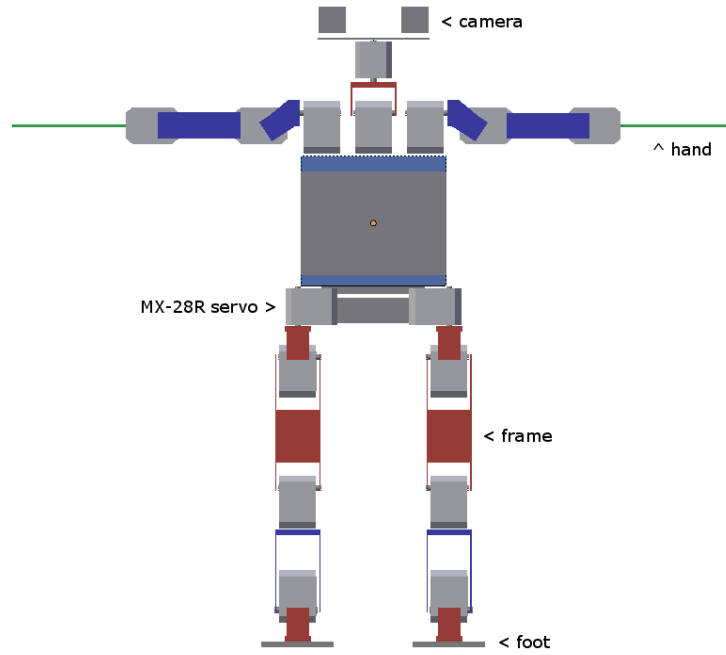
(a) Front view of the robot.
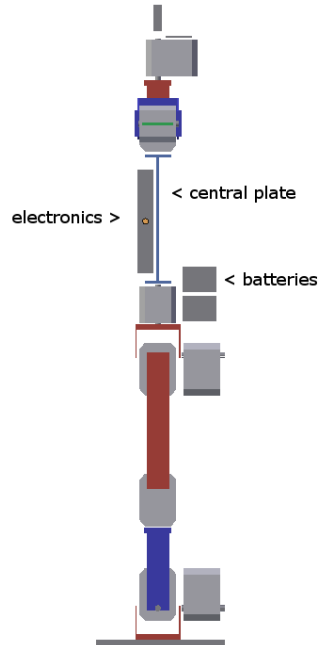
(b) Side view of the robot.

Figure 4.11: Front and side views of the final robot model to be used in the simulations in chapter 5.

Chapter 5

Applications

This chapter first explains how to combine the model we created in the preceding chapter and the remote control capabilities of V-Rep to finally perform some simulations. We then analyse some simulations we used to test robot designs. We finally summarize the influence of this master's thesis on the final design of the robot.

5.1 Problem statement

In V-Rep, we have the model of a robot and we want to test its ability to stand, stand up and walk. We need to somehow connect an external control program to the model inside V-Rep and control the servomotors. We must then use it to devise control sequences (routines) that will test the robot's abilities.

5.2 Overview of the simulation setup

The solution is quite simple because when V-Rep is running, it creates a TCP server socket and we are free to connect ourselves to it with another program (for the purpose of this master's thesis it is sufficient to know that a socket allows two programs to communicate). V-Rep even provides a library which implements a set of instructions[1].

Some of the most useful instructions are:

– **simxGetObjectHandle:** this function is used to retrieve a handle of an object by specifying its name. However, this function is the only function that can reach an object through its name. Therefore it is necessary to retrieve the handle of an object before using any other instruction on it.

– **simxSetJointTargetPosition:** this function sets a target position for a joint that is identified by the handle we give to the function as an argument.

– **simxGetJointPosition:** this function retrieves the current position of a joint, in radians.

---

[1]the whole list is available on http://www.coppeliarobotics.com/helpFiles/en/remoteApiFunctionListAlphabetical.htm

– **simxGetObjectVelocity:** this function retrieves the velocity of an object. It can be used to simulate an accelerometer, by differentiating two consecutive measures of the velocity of an object.

– **simxGetFloatSignal:** this function retrieves the value of a float signal. A signal is a type of variable that can be created during the simulation and that is accessible both to the simulator and an external program. This is useful if we want to extend the interface that V-Rep provides. For example, it could be used to retrieve the position of the COM of the robot.

The simulation thus has two main components : V-Rep (simulator) and a Matlab script (robot controller). The latter is written in Matlab mainly for convenience and personal preference, and could have been written in C++ or Java. This is a great architecture solution because we can present to the controller an interface that is identical to the one of the real robot. Figure 5.1 illustrates the simulation setup.
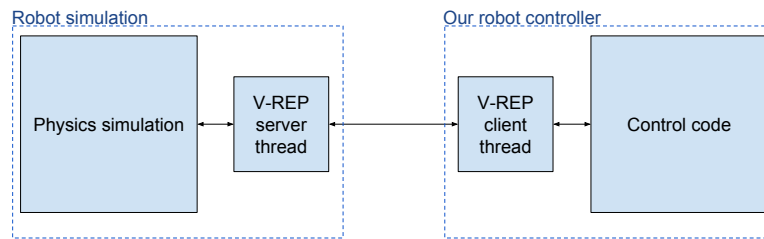
Figure 5.1: V-Rep simulates the robot while an external program sends order to the robot over TCP/IP thanks to the client/server thread provided by V-Rep.

These two components can either be executed synchronously or asynchronously. In synchronous operation mode, V-Rep will execute the simulation loop without interruption while handling the requests from the controller between two consecutive iterations. The controller cannot take too long to compute its orders or the simulation will have moved on and the robot will be in a different state than expected. In a way, this is also true for a real robot controller. We choose to work synchronously: each each simulation timestep must be triggered by the control code, as shown in Figure 5.2.

5.3 Applications

This section presents some simulations we made in order to prove the ability of a robot to stand, stand up or move from one lying position to another. The model we use is the one presented at the end of the previous chapter, shown in fig. 4.11.

5.3.1 Static stability

The first application is a simple test of the robot's ability to stand upright on its own, using the servomotors. The servomotors of the robot are simply ordered to hold their initial angle and the simulation determines that the robot can indeed stand upright without any elaborate control. This is a simple test that can rule out bad designs quite easily and more specifically designs in which the servomotors are not powerful enough.

Figure 5.2: Typical interaction between the simulator and the control code. The simulation runs on two threads: the simulation and the server thread. The latter can receive orders from a client thread which is controlled by a custom application of our own. The simulator waits for a trigger before simulating the next timestep. Source: [Rob16]

5.3.2 Going from a supine to a prone position

The main motivation for a routine that makes the robot move from a supine[2] position to a prone[3] one is that it allows us to only have one standing up routine.

The routine is defined as follows:

1. From 0 to 0.39*sec*: the robot brings his right arm above his head while preparing the left one to lift its body from the left side. Both legs are twisted towards the right side at the hips.

2. From 0.40 to 0.99*sec*: the left leg swings towards the right side while the right leg swings towards the left side. The left elbow prepares to push.

3. From 1.00 to 1.59*sec*: the left elbow pushes the body up and the hips untwist. The left feet touches the ground on the right side.

4. At 1.6*sec*: The robot relaxes all its limbs and is now prone.

The evolution of the angles held by the joints during the routine is presented in fig. 5.3 and fig. 5.4. The control code is in Listing D.1 of the appendices.

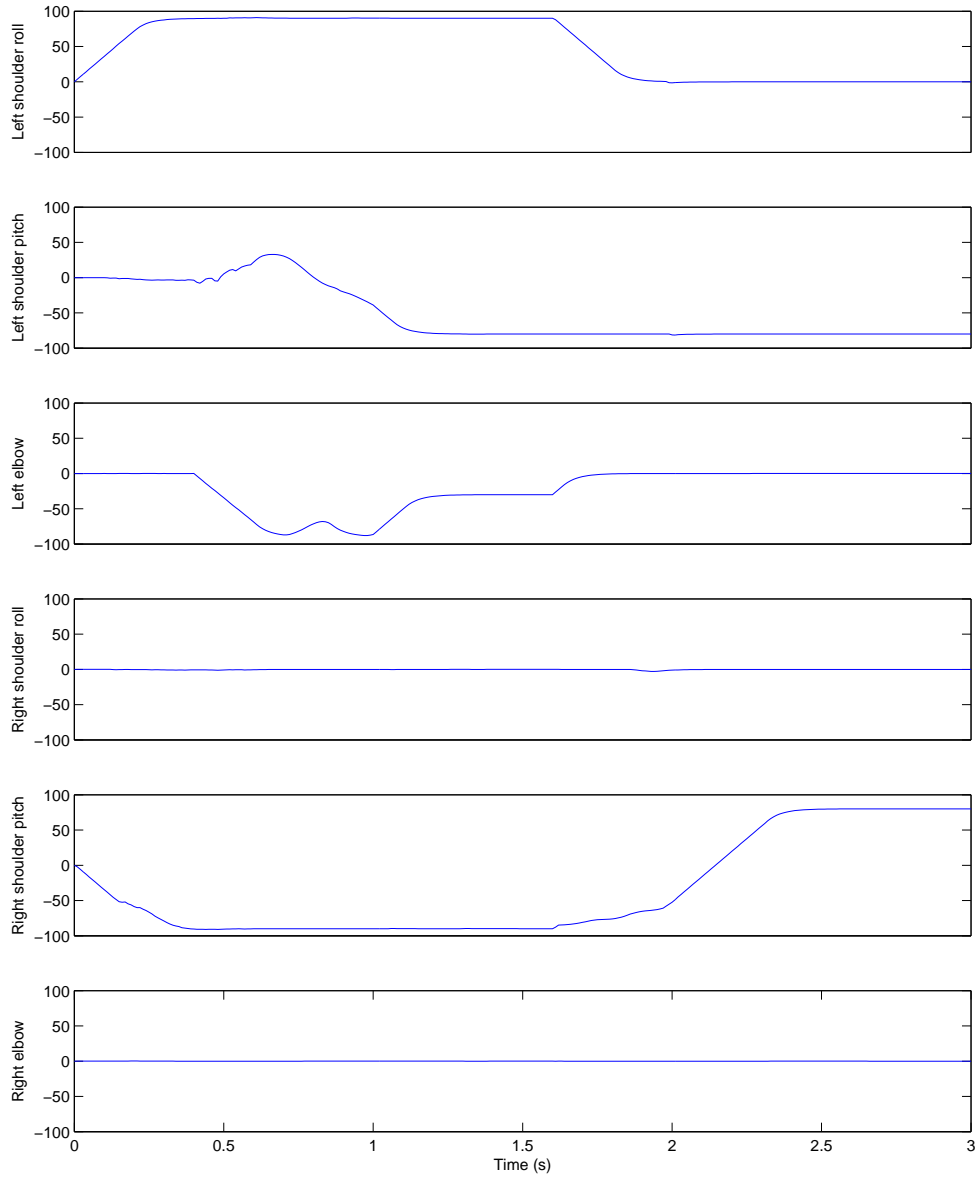[2]lying on the back
[3]lying on the belly

Figure 5.3: Angles of the arms during *supine to prone* manoeuvre. Predictably it is the left arm that is the most active as it is the one that is used to make the robot roll over. We notice that at times (around 0.7*sec*) the left elbow does not generate enough torque to hold the desired angle.
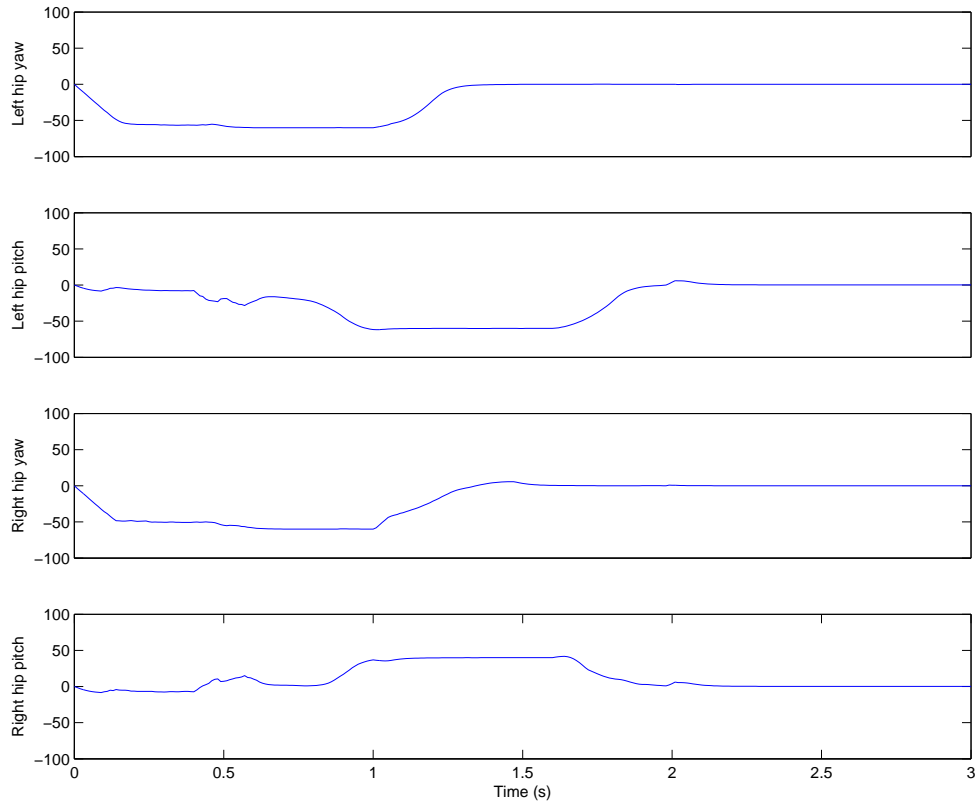
Figure 5.4: Angles of the legs during *supine to prone* manoeuvre. Both hips are twistes towards the right side but the legs go in opposite directions, the left one towards the right side and the right one towards the left side. Similarly to the previous figure, we notice that sometimes the servomotors cannot hold the desired angle.

5.3.3 Standing up from a prone position

This routine was inspired by Jörg Stückler, Johannes Schwenk, and Sven Behnke in [SSB06]. It is defined as follows:

1. From 0.00 to 0.19*sec*: in preparation for the arms to lift the body in the next step we adjust the roll of the shoulders.

2. From 0.20 to 0.49*sec*: in order to reduce the stress on the servos of the arm, we bend the arms a little before the next step.

3. From 0.50 to 0.89*sec*: lift the body up through the hips with the help of the arms. This step and the next are preparation for the next step where we will move the feet underneath the body.

4. From 0.90 to 1.39*sec*: now that the body is held up by the arms we reverse the holding angle for the hips, to move the hips up.

5. From 1.40 to 1.99*sec*: we move the feet underneath the body by bending the knees and bringing the legs closer to the body. The objective is to bring the COM inside the support area of the feet.

6. From 2.80 to 4.00*sec*: the robot slowly gets up by unbending the knees and the hips.

The evolution of the angles held by the joints during the routine is presented in fig. 5.5 and fig. 5.6. The control code is in Listing D.2 of the appendices.
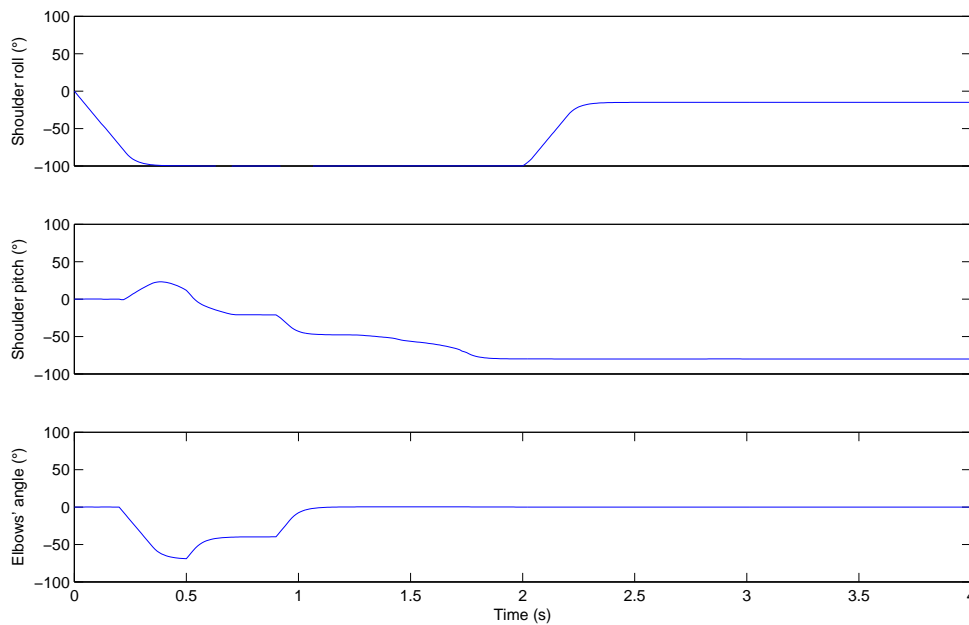

Figure 5.5: Angles of the arms during *supine to prone* manoeuvre. Contrary to last time both arms receive the same orders. At first, the servomotors in the shoulders rotate the arms to put them in the right position for pushing. The elbows are bent to help the servos in the shoulders lift the trunk.
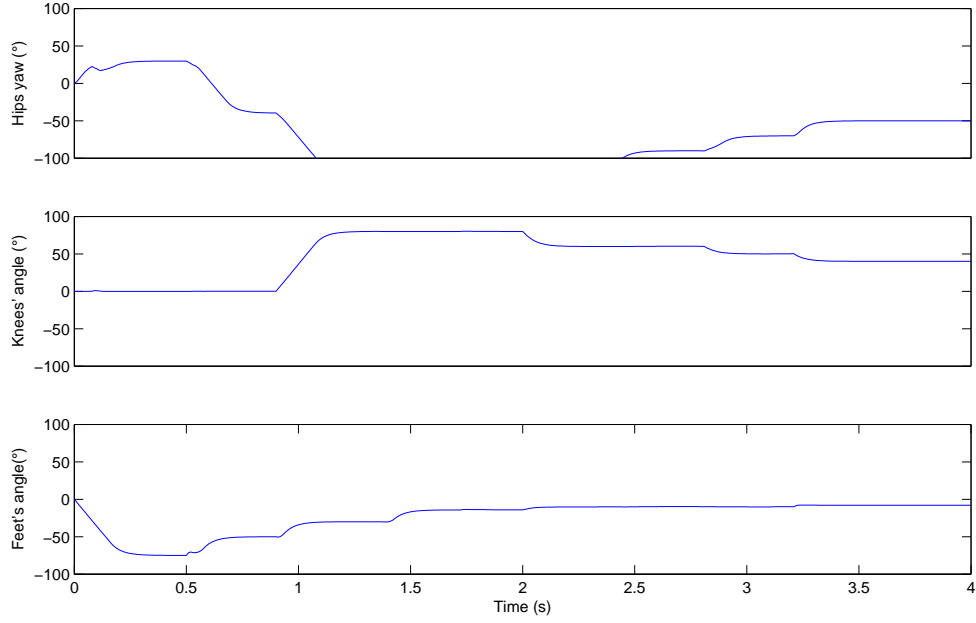
Figure 5.6: Angles of the legs during *supine to prone* manoeuvre. Again, both legs receive the same orders. Notice the importance of footwork in this routine.

5.4 Influence of the simulations on the design of the robot

The ultimate purpose of this work was to help the team of students design a humanoid robot able of participating in RoboCup contest. This section highlights the main influences our work had on the design of the robot.

The first robot we tried to simulate was the one in Figure 5.7a. At that time, it was not clear if the cameras at the top were to be moved by MX-28R servomotors or some smaller and cheaper servos hence the absence of servos under them. The hands (the balls) were still in their early phase and were approximated by balls, the main idea being that some sort of element is needed between the last servo of the arm and ground. The legs were very long as the servos were all the same plane. This design has never been thoroughly tested, it was more of an exercise. It was too inaccurate for the simulations to be of any use. We show it here for the purpose of presenting the design ideas that existed at the beginning of the project.

The first robot we really tested is the one in Figure 5.7b. A third servo has appeared at the top of the robot and the hands became cylinders. The legs are shorter, because the servos that connect the legs to the trunk are now hidden behind the trunk. Which got longer and larger. This robot did not perform very well. Its arms were too short for it to be able to stand up. The range of movements of its joints was also too limited, especially at the feet, knees and hips.

The third iteration (Figure 5.7c) tried to overcome these limitations. The arms are longer and the legs were modified to have a wider movement range (some servos were connected together, in the fashion that is visible in Figure 4.11b). After some adjustments of the length of the hinges in the legs it was able to stand up but by making the servos in the arms stronger than the MX-28R really is. It was this model
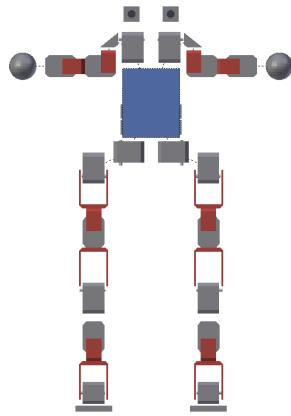
(a) 1st iteration.
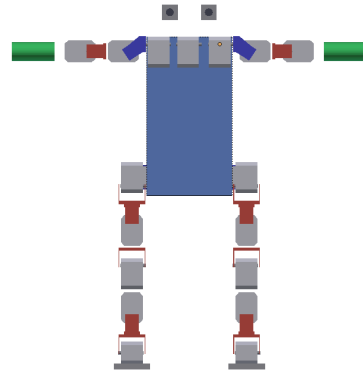
(b) 2nd iteration.
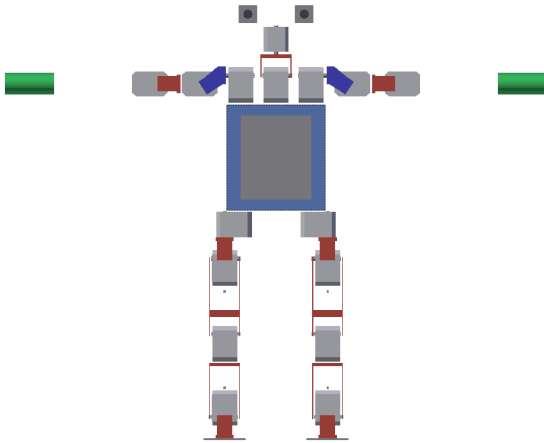
(c) 3rd iteration.
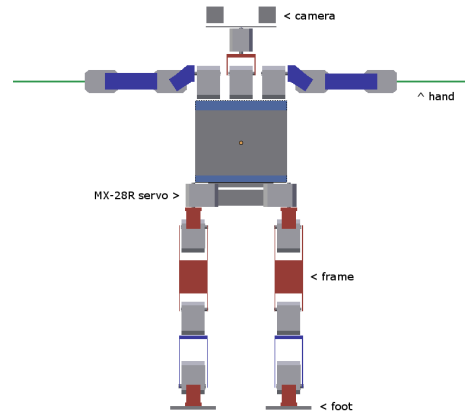
(d) Final and last iteration.

Figure 5.7: Evolution of the robot design.

that had collision problems we mentioned in Section 4.2 because its feet were too thin.

The last design (Figure 5.7d) thus has different arms from its predecessor. The servos at their extremities are placed in the middle. The hands became rectangular plates. The batteries are also placed lower in an effort to lower the centre of mass. This design, as you already know, was able to stand up and perform other routines. It also respect the rules of the contest (see appendix A):

1. Height (H_{top}): $40 \leq 61.75 \leq 90cm$.

2. Weight: $2.726kg$.

3. Height of centre of mass (H_{COM}): $34cm$. Foot area: $175 < cm^2$.

4. Foot aspect ratio: $1.79 \leq 2.5$.

5. Minimal width of the robot: $20 \leq 33.96cm$.

6. $69.72 \leq 74.10cm$.

7. Maximal height: $75.52 < 92.63cm$.

8. Leg length (H_{leg}): $21.61 \leq 27.5 \leq 43.23cm$.

9. Height of the head (H_{Head}): $3.09 \leq 10.75 \leq 15.44cm$.

Chapter 6

Conclusions

6.1 Contributions

The objective of this master's thesis was twofold, first to find or create a simulation tool able accurately simulating the behaviour of a robot in a physical world, second, to use that tool as a prototyping platform for the purpose of designing a humanoid robot able of performing actions such as standing up and walking.

6.1.1 Simulation tool

We solved the first problem by surveying the available physics simulation software and choosing the ones we were going to use. We decided to work with a simulator instead of just a physics engines in order to begin working with simulations sooner, without spending time on re-developing features that are already present in simulators. Out of the surveyed simulators, V-Rep was chosen because it has all the desired features (joints, access to the parameters of the physics engine, remote control of the simulation) and has good reviews, as shown in a survey done by Hummel *et al.* in [HWS+12]. However, V-Rep is not without flaws, the biggest one being the lack of a proper mesh creation capabilities. For that reason, we chose to use Blender to create the meshes of objects we want to model.

6.1.2 Simulations

We solved the second problem using both Blender an V-Rep to create models of the elements we deemed necessary for a complete model of a robot. Some objects were easier to model, as they do not perform any active role other than being physically present. Other necessitated the use of scripts to emulate their real-life behaviour. Such was the case of the cameras and servomotors. Fortunately V-Rep provides some basic infrastructure for the simulation of vision sensors and servomotors. Most of the work thus went into the determination of the correct parameters to use in order to produce a physically accurate simulation.

These objects were assembled together to create a complete model of the robot. Different designs were tested and each iteration brought us closer to a robot able of standing and walking which we finally created.

6.2 Problems encountered

A master's thesis is a major endeavour and these are rarely devoid of obstacles. During this year several elements obstructed the completion of this work:

– V-Rep is a fine tool but the lack of a proper internal modelling tool was a major thorn in the side as every major modification meant that the whole model had to be modified in Blender and re-imported into V-Rep. This created a lot of overhead work which contributed nothing of interest to this work.

This drawback is generalized amongst all the simulators that we surveyed at the beginning of this report and we feel it should be addressed quickly by their creators. Nevertheless, we understand that a modelling tool such as Blender took years to create so we would not expect simulators to catch up any time soon.

– We also learned of the importance of studying mechanisms carefully before using them. Though things might appear simple at first, subtle implementation details might change everything. A fine example of that is us melting the core of the motor inside a MX-28R servomotor during our tests because of our trust in the announced safety mechanisms. Needless to say, they proved insufficient and we should have examined the documentation more closely.

– Choosing a physics engine was difficult because the field is quite fragmented. On one hand there exist well established commercial solutions, but they are focused on games and make some significant shortcuts whenever possible in order to be as fast as possible. On the other hand there exist a quantity of open-source physics engines but they are usually the work of one man and are poorly documented. It was hard to motivate the choice of Newton Dynamics on any other basis than 'it worked best'.

6.3 Future work

6.3.1 Modelling

While the model is in a usable state it could still be bettered and we suggest to begin with the items listed hereafter:

– **Springs.** As of now our work with springs is just a proof of concept.

– **Inertia.** The model uses a simplified representation of inertia, in the belief that a controller should be able to correct minor differences in behaviour between the model and the actual robot. If inertia needs to be made more accurate, we suggest to use Meshlab[1] to compute the inertia of objects.

– **Model format.** It is still uncertain if Blender shall continue to support the Collaborative Design Activity (COLLADA) format, as mentioned in the development roadmap ([Ble15]). This format has been very useful throughout this master's thesis as it made possible to import a 3D model of the robot while keeping the distinct objects of the model separate. This is not the case

---

[1]http://meshlab.sourceforge.net/

with other mesh formats where the imported model is a single entity that we must separate into atomic elements.

Should the support for this format be dropped, a choice should be made whether to continue using the COLLADA format and find another modelling software that supports it, or to move on to another format. In the case the latter is chosen we suggest to use the Unified Robot Description format (URDF), basically a xml file which is supported by most of the robotics simulators we surveyed.

6.3.2 Routines

Now that we have a simulator and a complete model of the robot, more routines can be created.

– **Standing up from a supine position.** Even though the robot can roll from a supine to a prone position and use the standing from prone routine, it would be faster to be able to stand from a supine position directly.

– **Walking.** Being able to walk is the basic requirement for a robot to compete in RoboCup. A walking sequence is the last proof needed to be able to tell that the robot we designed is able to compete.

– **Shooting a ball.** As soon as the robot is able to walk, the next step should be testing if it can shoot a soccer ball.

6.3.3 Online simulation

In parallel or after creating the aforementioned routines, the simulator should be used to test the high level control code of the robot. The interaction between the simulator and the control code would be the same as in this master's but the control code would be much more complex than the routines we created.

Bibliography

[BETC12]   Jan Bender, Kenny Erleben, Jeff Trinkle, and Erwin Coumans. Interactive simulation of rigid body dynamics in computer graphics. In *EUROGRAPHICS 2012 State of the Art Reports*, pages 95–134. Eurographics Association, 2012.

[Ble15]   Blender. 2.8 project developer kickoff meeting notes, 2015. https://code.blender.org/2015/11/the-2-8-project-for-developers/ [Accessed 01-May-2016].

[Dyn16a]   Dynamixel. Robotis e-manual v1.27.00, 2016. http://support.robotis.com/en/product/dynamixel/mx_series/mx-28.htm [Accessed 18-April-2016].

[Dyn16b]   Dynamixel. Robotis e-manual v1.27.00, 2016. http://support.robotis.com/en/product/auxdevice/interface/usb2dxl_manual.htm [Accessed 18-April-2016].

[ETT15]   Tom Erez, Yuval Tassa, and Emanuel Todorov. Simulation tools for model-based robotics: Comparison of bullet, havok, mujoco, ode and physx. *International Conference on Robotics and Automation*, 2015.

[HWS+12]   Johannes Hummel, Robin Wolff, Tobias Stein, Andreas Gerndt, and Torsten Kuhlen. An evaluation of open source physics engines for use in virtual reality assembly simulations. In *Advances in visual computing*, pages 346–357. Springer, 2012.

[IPPN14]   Serena Ivaldi, Jan Peters, Vincent Padois, and Francesco Nori. Tools for simulating humanoid robot dynamics: a survey based on user feedback. In *Humanoid Robots (Humanoids), 2014 14th IEEE-RAS International Conference on*, pages 842–849. IEEE, 2014.

[JM05]   Michael A Johnson and Mohammad H Moradi. *PID control*. Springer, 2005.

[JTT01]   Pablo Jiménez, Federico Thomas, and Carme Torras. 3d collision detection: a survey. *Computers & Graphics*, 25(2):269–285, 2001.

[Mir96]   Brian Vincent Mirtich. *Impulse-based dynamic simulation of rigid body systems*. PhD thesis, University of California at Berkeley, 1996.

[Rob15]   Robocup. Robocup soccer humanoid league rules and setup, 2015.

[Rob16]   Coppelia Robotics. V-rep user manual 3.3.0, 2016. http://www.coppeliarobotics.com/helpFiles/index.html [Accessed 18-April-2016].

[SSB06]   Jörg Stückler, Johannes Schwenk, and Sven Behnke. Getting back on two feet: Reliable standing-up routines for a humanoid robot. In *IAS*, pages 676–685, 2006.

Appendix A

Rules

The robots that participate in the kidsize competition must respect the following characteristics:

1. $40cm \leq H_{top} \leq 90cm$.

2. Maximum allowed weight is $20kg$.

3. Each foot must fit into a rectangle of area $(2.2 \cdot H_{com})^2/32$.

4. Considering the rectangle enclosing the convex hull of the foot, the ratio between the longest side of the rectangle and the shortest one, shall not exceed 2.5.

5. The robot must fit into a cylinder of diameter $0.55 \cdot H_{top}$.

6. The sum of the lengths of the two arms and the width of the torso at the shoulder must be less than $1.2 \cdot H_{top}$. The length of an arm is defined as the sum of the maximum length of any link that forms part of the arm. Both arms must be the same length.

7. The robot does not possess a configuration where it is extended longer than $1.5 \cdot H_{top}$.

8. The length of the legs H_{leg}, including the feet, satisfies $0.35 \cdot H_{top} \leq H_{leg} \leq 0.7 \cdot H_{top}$.

9. The height of the head H_{head}, including the neck, satisfies $0.05 \cdot H_{top} \leq H_{head} \leq 0.25 \cdot H_{top}$. H_{head} is defined as the vertical distance from the axis of the first arm joint at the shoulder to the top of the head.

10. The leg length is measured while the robot is standing up straight. The length is measured from the first rotating joint where its axis lies in the plane parallel to the standing ground to the tip of the foot.

Source: [Rob15].

Appendix B

Design guidelines

For a dynamic simulation several design restrictions must be considered:

– use pure convex as much as possible, they are much more stable and faster to simulate. When a more complex shape is used, approximate it with several convex shapes.

– use reasonable sizes, neither not too small nor too big. Thin shapes may behave strangely.

– when using joints, keep the ratio of the masses below 10. Otherwise, the joint may have large orientation/position errors.

Source : [Rob16].

Appendix C

Minimal simulation control code

```matlab
function simulation_client_vrep()

disp('Program started');
vrep = remApi('remoteApi'); % use the prototype file
vrep.simxFinish(-1); % close all opened connections
clientID = vrep.simxStart('127.0.0.1', 19997, true, true,
    5000, 5);

if clientID < 0
    disp('Failed connecting. Exiting.');
    vrep.delete();
    return;
end
disp('Connected to remote API server');

% Close the connexion whenever the script is interrupted.
cleanupObj = onCleanup(@() cleanup_vrep(vrep, clientID));

% Set the remote mode to 'synchronous'
vrep.simxSynchronous(clientID, true);

% retrieve handles to servos, joints
h = robot_init(vrep, clientID);

% start the simulation
vrep.simxStartSimulation(clientID, vrep.
    simx_opmode_oneshot_wait);

t = 0;
dt = 0.1; %timestep of the simulation
while true && t < 3
    instructions = standup_prone(h, t);
    send_instructions(vrep, clientID, instructions);
    t = t + dt;
end
```

```
35  % Before closing the connection to V-REP, make sure that
        the last command sent out had time to arrive.
36  vrep.simxGetPingTime(clientID);
37
38  % Close the connection to V-REP:
39  vrep.simxStopSimulation(clientID, vrep.
        simx_opmode_oneshot_wait);
40  vrep.simxFinish(clientID);
41  vrep.delete(); % call the destructor!
42  disp('Program ended');
43  end
```

Listing C.1: Minimal example code that connects to the server, gets the handles and implements a basic simulation control loop.

```
1  function res = send_instructions( vrep, clientID,
      angle_instructions )
2      vrep.simxPauseCommunication(clientID, 1);
3      for i=1:size(angle_instructions, 1)
4          res = vrep.simxSetJointTargetPosition(clientID, ...
5              angle_instructions(i, 1), angle_instructions(i
                  , 2), ...
6              vrep.simx_opmode_oneshot);
7      end
8      vrep.simxPauseCommunication(clientID, 0);
9
10     vrep.simxGetPingTime(clientID);
11     vrep.simxSynchronousTrigger(clientID);
12 end
```

Listing C.2: Function that handles the sending of target angles to joints.

```matlab
1  function handles = robot_init( vrep, clientID )
2
3  handles = struct('clientID', clientID);
4
5  %% Retrieve center
6  [~, center] = vrep.simxGetObjectHandle(clientID, 'Central
       ',...
7      vrep.simx_opmode_oneshot_wait);
8  handles.center = center;
9
10 %% Joints
11 j = 1;
12 left_arm = [-1, -1, -1];
13 right_arm = [-1, -1, -1];
14 for i = 1:3
15     [~, left_arm(i)] = vrep.simxGetObjectHandle(clientID,
           ...
16          sprintf('left_arm%d', i), vrep.
              simx_opmode_oneshot_wait);
17     [~, right_arm(i)] = vrep.simxGetObjectHandle(clientID
           ,...
18          sprintf('right_arm%d', i), vrep.
              simx_opmode_oneshot_wait);
19
20     % Reset instructions
21     instructions(j, :) = [double(left_arm(i)), 0]; j = j
           + 1;
22     instructions(j, :) = [double(right_arm(i)), 0]; j = j
           + 1;
23 end
24
25 left_leg = [-1, -1, -1, -1, -1, -1];
26 right_leg = [-1, -1, -1, -1, -1, -1];
27 for i = 1:6
28     [~, left_leg(i)] = vrep.simxGetObjectHandle(clientID,
           ...
29          sprintf('left_leg%d', i), vrep.
              simx_opmode_oneshot_wait);
30     [~, right_leg(i)] = vrep.simxGetObjectHandle(clientID
           ,...
31          sprintf('right_leg%d', i), vrep.
              simx_opmode_oneshot_wait);
32
33     instructions(j, :) = [double(left_leg(i)), 0]; j = j
           + 1;
34     instructions(j, :) = [double(right_leg(i)), 0]; j = j
           + 1;
35 end
36
```

```
37  handles.left_leg = left_leg;
38  handles.right_leg = right_leg;
39  handles.left_arm = left_arm;
40  handles.right_arm = right_arm;
41
42  %% Retrieve cameras' handles
43  head = [-1, -1];
44  for i = 1:2
45      [~, head(i)] = vrep.simxGetObjectHandle(clientID,...
46          sprintf('head%d', (i)), vrep.
                  simx_opmode_oneshot_wait);
47  end
48
49  handles.head = head;
50
51  %% Reset positions
52  send_instructions(vrep, clientID, instructions);
53
54  end
```

Listing C.3: Function that retrieves the handles to the joints and the cameras.

Appendix D

Routines

```matlab
function inst = go_prone(h, t)

if t < 0.4
    % Left arm pushes ground
    inst(1, :) = [double(h.left_arm(1)), degtorad(90)];

    % Right arm close to body
    inst(2, :) = [double(h.right_arm(1)), degtorad(0)];
    inst(3, :) = [double(h.right_arm(2)), degtorad(-90)];

    %Position right leg
    inst(4, :) = [double(h.right_leg(1)), degtorad(-60)];
    inst(5, :) = [double(h.right_leg(3)), degtorad(-10)];

    %Position left leg
    inst(6, :) = [double(h.left_leg(1)), degtorad(-60)];
    inst(7, :) = [double(h.left_leg(3)), degtorad(-10)];
elseif t < 1
    % left elbow prepares to push
    inst(1, :) = [double(h.left_arm(3)), degtorad(-90)];
    inst(2, :) = [double(h.left_arm(2)), degtorad(-80)];

    %Position right leg
    inst(3, :) = [double(h.right_leg(3)), degtorad(40)];

    %Position left leg
    inst(4, :) = [double(h.left_leg(3)), degtorad(-60)];

elseif t < 1.6
    % Push with left elbow
    inst(1, :) = [double(h.left_arm(3)), degtorad(-30)];

    % Move right arm underneath body
    inst(2, :) = [double(h.right_arm(2)), degtorad(-90)];

    % Position right leg
```

```matlab
        inst(3, :) = [double(h.right_leg(1)), degtorad(0)];
        inst(4, :) = [double(h.right_leg(4)), degtorad(30)];

        % Position left leg
        inst(5, :) = [double(h.left_leg(1)), degtorad(0)];
    else
        % Relax right leg
        inst(1, :) = [double(h.right_leg(1)), degtorad(0)];
        inst(2, :) = [double(h.right_leg(3)), degtorad(0)];
        inst(3, :) = [double(h.right_leg(4)), degtorad(0)];

        % Relax left leg
        inst(4, :) = [double(h.left_leg(1)), degtorad(0)];
        inst(5, :) = [double(h.left_leg(3)), degtorad(0)];
        inst(6, :) = [double(h.left_leg(4)), degtorad(0)];

        % Relax right arm
        inst(7, :) = [double(h.right_arm(1)), degtorad(0)];
        inst(8, :) = [double(h.right_arm(2)), degtorad(80)];
        inst(9, :) = [double(h.right_arm(3)), degtorad(0)];

        % Relax left arm
        inst(10, :) = [double(h.left_arm(1)), degtorad(0)];
        inst(11, :) = [double(h.left_arm(2)), degtorad(-80)];
        inst(12, :) = [double(h.left_arm(3)), degtorad(0)];
    end

end
```

Listing D.1: Routine that flips the robot from a supine to a prone position.

```matlab
function instructions = standup_prone(h, t)

if t < 0.2
    % hips
    instructions(1,:) = [double(h.right_leg(3)), degtorad
        (30)];
    instructions(2,:) = [double(h.left_leg(3)), degtorad
        (30)];

    % feet
    instructions(3,:) = [double(h.right_leg(5)), degtorad
        (-75)];
    instructions(4,:) = [double(h.left_leg(5)), degtorad
        (-75)];

    % right shoulder
    instructions(5,:) = [double(h.right_arm(1)), degtorad
        (-100)];

    % left shoulder
    instructions(6,:) = [double(h.left_arm(1)), degtorad
        (-100)];
elseif t < 0.5
    % right shoulder
    instructions(1,:) = [double(h.right_arm(2)), degtorad
        (70)];
    instructions(2,:) = [double(h.right_arm(3)), degtorad
        (70)];
    % left shoulder
    instructions(3,:) = [double(h.left_arm(2)), degtorad
        (-70)];
    instructions(4,:) = [double(h.left_arm(3)), degtorad
        (-70)];

elseif t < 0.9
    % right arm
    instructions(1,:) = [double(h.right_arm(2)), degtorad
        (80)];
    instructions(2,:) = [double(h.right_arm(3)), degtorad
        (40)];

    % left arm
    instructions(3,:) = [double(h.left_arm(2)), degtorad
        (-80)];
    instructions(4,:) = [double(h.left_arm(3)), degtorad
        (-40)];

    % Right leg
    instructions(5,:) = [double(h.right_leg(3)), degtorad
```

```matlab
        (-40)];
    instructions(6,:) = [double(h.right_leg(5)), degtorad
        (-50)];

    % Left leg
    instructions(7,:) = [double(h.left_leg(3)), degtorad
        (-40)];
    instructions(8,:) = [double(h.left_leg(5)), degtorad
        (-50)];
elseif t < 1.4
    % Bring the knees in
    instructions(1,:) = [double(h.right_leg(4)), degtorad
        (80)];
    instructions(2,:) = [double(h.left_leg(4)), degtorad
        (80)];

    instructions(3,:) = [double(h.right_leg(3)), degtorad
        (-140)];
    instructions(4,:) = [double(h.left_leg(3)), degtorad
        (-140)];

    % Use arms to lift trunk up
    % right arm
    instructions(5,:) = [double(h.right_arm(3)), degtorad
        (0)];

    % left arm
    instructions(6,:) = [double(h.left_arm(3)), degtorad
        (0)];

    % Feet
    instructions(7,:) = [double(h.right_leg(5)), degtorad
        (-30)];
    instructions(8,:) = [double(h.left_leg(5)), degtorad
        (-30)];
elseif t < 2
    % Feet
    instructions(1,:) = [double(h.right_leg(5)), degtorad
        (-14)];
    instructions(2,:) = [double(h.left_leg(5)), degtorad
        (-14)];

    % right arm
    instructions(3,:) = [double(h.right_arm(2)), degtorad
        (80)];

    % left arm
    instructions(4,:) = [double(h.left_arm(2)), degtorad
        (-80)];
```

```matlab
    % head
    instructions(5,:) = [double(h.head(1)), degtorad(-70)
        ];
elseif t < 2.8
    % Time to stand
    % right leg
    instructions(1,:) = [double(h.right_leg(3)), degtorad
        (-90)];
    instructions(2,:) = [double(h.right_leg(4)), degtorad
        (60)];
    instructions(3,:) = [double(h.right_leg(5)), degtorad
        (-10)];

    % left leg
    instructions(4,:) = [double(h.left_leg(3)), degtorad
        (-90)];
    instructions(5,:) = [double(h.left_leg(4)), degtorad
        (60)];
    instructions(6,:) = [double(h.left_leg(5)), degtorad
        (-10)];

    % right arm
    instructions(7,:) = [double(h.right_arm(1)), degtorad
        (-15)];

    % left arm
    instructions(8,:) = [double(h.left_arm(1)), degtorad
        (-15)];

    % head
    instructions(9,:) = [double(h.head(1)), degtorad(0)];
elseif t < 3.2
    % right leg
    instructions(1,:) = [double(h.right_leg(3)), degtorad
        (-70)];
    instructions(2,:) = [double(h.right_leg(4)), degtorad
        (50)];
    instructions(3,:) = [double(h.right_leg(5)), degtorad
        (-10)];

    % left leg
    instructions(4,:) = [double(h.left_leg(3)), degtorad
        (-70)];
    instructions(5,:) = [double(h.left_leg(4)), degtorad
        (50)];
    instructions(6,:) = [double(h.left_leg(5)), degtorad
        (-10)];
else
    % right leg
    instructions(1,:) = [double(h.right_leg(3)), degtorad
```

```matlab
                (-50)];
        instructions(2,:) = [double(h.right_leg(4)), degtorad
            (40)];
        instructions(3,:) = [double(h.right_leg(5)), degtorad
            (-8)];

        % left leg
        instructions(4,:) = [double(h.left_leg(3)), degtorad
            (-50)];
        instructions(5,:) = [double(h.left_leg(4)), degtorad
            (40)];
        instructions(6,:) = [double(h.left_leg(5)), degtorad
            (-8)];
    end

end
```

Listing D.2: Routine that makes the robot stand up from a prone position.