Universidad Carlos III de Madrid

Departamento de Sistemas y Automática

Bachelor's Thesis:

Learning locomotion gait through hormone-based controller in modular robots

Author:
David Estévez Fernández

Advisor:
Avinash Ranganath

June 22, 2014

Acknowlegments

First of all I would like to thank my family, my brother and specially my parents for all their efford by which I am here today, and for supporting all my decisions during all of these years.

I would also like to thank my old teachers Juan and Alberto, for introducing me in the world of robotics and 3D printing, and turning me a self-taught person, no longer waiting for someone to teach me something I want to learn.

To the people in Samsamia and the Robotics Society, for teaching me a lot of things along this year that have help me a lot with the development of this work.

And last but no least, I would like to thank my classmates David, Irene and Elena not only for all the labs, reports and works we have suffered together, but also for all the good times we have spent during these four long years that now have come to an end.

Learning locomotion gait through hormone-based controller in modular robots

David Estévez Fernández

Abstract

Modular robots are robots composed of multiple units, called 'modules'. Each module is an independent robot, with its own control electronics, actuators, sensors, communications and power. These modules can change their position and configuration in order to adapt to the requirements of the situation, making modular robot suitable for tasks that involve unknown or unstructured terrains, in which a robot cannot be designed specifically for them. Some examples of those applications are space exploration, battlefield reconnaissance, finding victims among the debris in natural catastrophes and other similar tasks involving complicated terrains, which require a high versability.

But this versability comes with several drawbacks. As modular robots are composed of several independent robots, the nature of their controller is distributed, which difficults their design and programming, requiring additionally a robust communication protocol to share information among modules. The high number of modules also results in a robot with a with number of degrees of freedom, for which achieving the coordination required for locomotion becomes increasingly difficult. Finally, as the modules are fully independent robots, the cost of researching modular robotics is usually very high, since the price of building a single robot has to be multiplied by the high number of modules.

This thesis addresses those three mentioned problems: obtaining optimal locomotion gaits from a biologically inspired approach, using sinusoidal oscillators whose parameters are found through evolutionary optimization algorithms; developing a homogenous, distributed controller based on digital hormones that can recognize the current robot configuration and select the proper gait; and the development of a low-cost modular robotic platform to reseach locomotion gaits for different configurations.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Modular Robots

Modular robots are robots composed by several autonomous units, called "modules", that work together in order to increase the overall capabilities of a single unit. Each module is a complete robot itself, having its own control electronics, actuators, sensors, power supply and some way of connecting to other modules to form a modular robot.

Modular robots have several advantages over traditional robots, the first being adaptability. Their modular design allows some modular robots to reconfigure themselves, changing the position of some of the modules within the robot body. This self-reconfiguration is very useful in unknown or changing environments, as the robot can adapt its body depending on the terrain, for example, becoming a snake-like robot to pass through pipes or holes, and later on reconfiguring to a legged robot to stand on top of obstacles. In some cases they can even detach themselves from the robot in order to explore the surroundings and act like a swarm, and then return back and reconstruct the robot again. Figure 1.1 shows the reconfiguration of a M-TRAN III modular robot from a quadruped configuration to a snake robot.

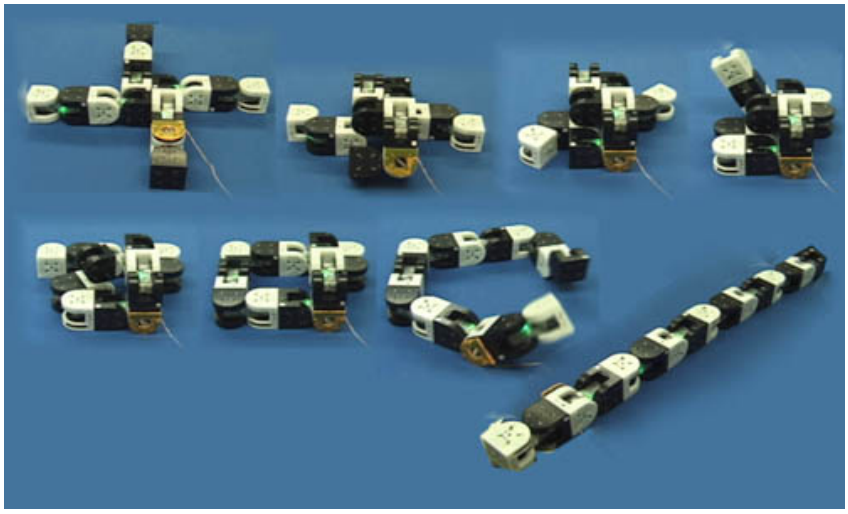


Figure 1.1: M-TRAN III modular robot reconfiguring from quadruped to snake robot.

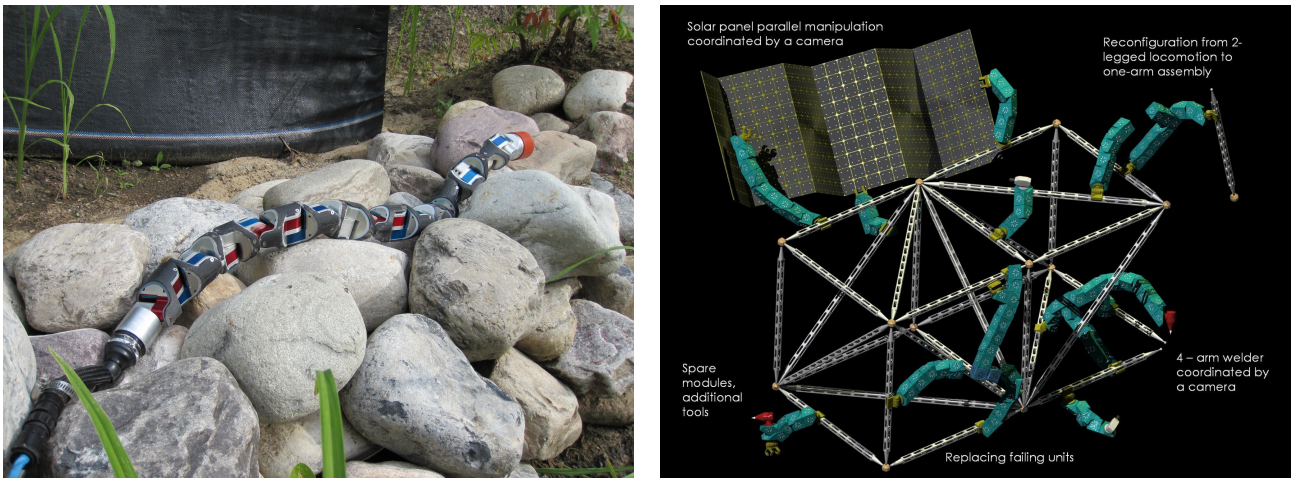
In general, modular robots are used in applications in which the operating conditions of the robot are not known when the robot is to be designed, such as space exploration, battlefield reconnaissance, finding victims among the debris in natural catastrophes and other similar tasks involving complicated terrains.

As the robot does not rely on a single module, modular robots are fault-tolerant, and most of the robot functionality will remain even when some modules fail. Those bad-functioning modules can be substituted by other new modules,

giving self-reconfigurable modular robots the property of self-repairing.

Modular robots can generally be classified by the arrangement of their basic unit in lattice type, chain type or hybrid type. Lattice modular robots have modules arranged in some regular pattern along 3D space, resembling atoms in crystals and they can move by changing the position of the individual modules in that lattice. In chain (or tree) modular robots the modules are connected forming strings or trees, allowing this kind of robots to reach any point of the space. Hybrid modular robots can behave as lattice type or chain type, combining the fast reconfiguration of the lattice modular robots with the ability of reaching any point of the chain type robots.

Modular robots can be also classified by their shape and functionality in homogeneous modular robots, in which all the unit modules follow the same design and heterogeneous modular robots, whose modules are different and each one is specialized in certain functions.



(a) Locomotion through unstructured terrains.

(b) Space applications.

Figure 1.2: Typical applications of modular robots.

In homogeneous modular robots, mass production can lower the cost of the robots, as they are all identical. But as currently these robots are used mainly for research, they are not mass produced, and usually very small batches of prototypes are manufactured. As each module is an autonomous robot, and therefore they need to have their own controller hardware and software, actuators (like motors), sensors, communication hardware and batteries, modular robots are usually very expensive, the cost of one single module has to be multiplied several times, increasing rapidly the cost of the robot.

Despite their versatility in uncertain situations, due to the high number of modules that compose a modular robot, they are usually hyper-redundant robots, robots with a large or infinite number of degrees of freedom. This complicates the search for adecuate locomotion gaits due to the increase in complexity to find the inverse kinematics of the robots, as well as the increase in difficulty of coordinating the movement of all the joints.

Other problem that arises in modular robotics is the distributed nature of the system, that requires each module to have its own controller, which has to interact with the other modules controllers in order to achieve a correct and optimal behavior of the whole modular robot. In this aspect, having a homogeneous controller (i.e. all the modules share exactly the same controller) eases the development and mainteinance of the controller, but a proper, scalable controller and communication protocol is still required for collaboratively control the entire robot.

In this thesis we address those three problems described: locomotion gait generation on a modular robot, designing a homogeneous distributed controller that can control the whole modular robot selecting the most appropiate gait for each configuration, and the development of a cheap and simple modular robot platform to test and validate locomotion gaits and controllers.

1.2 Objectives

The main objective of this thesis is to solve some of the problems in modular robotics mentioned in the introductory section. More precisely, to develop a homogeous distributed controller and optimal locomotion gaits that allow a modular robot to move as fast as possible adapting its gaits to its current configuration, and test this controller on both a simulated and a real modular robotic platform.

In order to achieve that objective successfully, we have divided it into four main objectives:

1. **To find optimal locomotion gaits** for at least three different modular robot configurations by means of stochastic optimization algorithms.

2. **To develop a homogeneous, distributed controller and communication algorithm** that allows a modular robot to discover its current configuration and select the most suitable gait for that configuration.

3. **To develop a software framework** that allows to simulate the modular robot and test the obtained optimal gaits, as well as the homogenous controller for their validation. This framework should be flexible enough to serve as a base for the development and testing of other controllers and configurations in the future.

4. **To develop a cheap hardware platform** for testing the obtained optimal gaits and the distributed controller on the real world. This includes the design of both the mechanical part of the module as well as the control electronics, and the later assembly of the different configurations of the modular robot. This platform should also be upgradeable and reusable in future research related to modular robots.

1.3 Phases of the project

A brief description of the different phases of this project is presented here chronologically ordered:

1. **Study of the existing work on the topic.**
   (a) Study of the state of the art on modular robotics.
   (b) Test existing open source modular robotics platforms.

2. **Development of basic software framework for simulation.**
   (a) Development of basic digital model of the module to be used.
   (b) Select and setup simulator.
   (c) Development of the basic software for the control of the modular robot on the simulation.

3. **Optimization of modular robot gaits.**
   (a) Study and selection of stochastic optimization algorithm to be used.
   (b) Optimization of gaits for the main configurations to be studied.

4. **Development of the distributed control algorithm.**
   (a) Develop the theoretical distributed control algorithm for configuration discovery and gait selection.

5. **Development of the remaining software framework for testing the gaits and distributed controller.**
   (a) Development of the software related to the communication between modules.
   (b) Development of the distributed controller for the module.
   (c) Testing of the controller and gaits on the simulated modular robot.

6. **Development of the hardware platform for testing the gaits and distributed controller.**
   (a) Design, manufacturing and assembly of the control board.
   (b) Design, manufacturing and assembly of the mechanical module.
   (c) Assembly of the different modular robot configurations.
   (d) Test of the locomotion gaits and distributed controller on the physical modular robot.

7. **Results documentation**
   (a) Comment and document software.
   (b) Upload software and hardware designs to online repositories under a open source license.
   (c) Write and defend thesis.

Chapter 2

State of the Art

In this chapter we will present the state of the art on the field of modular robotics. More precisely, we will discuss the state of the art of the main modules used in modular robotics research, from which we took inspiration to design our platform; the state of the art of the controllers used for achieving locomotion on chain-type modular robots and the state of the art of modular robotics communications.

2.1 Modular Robotic Platforms

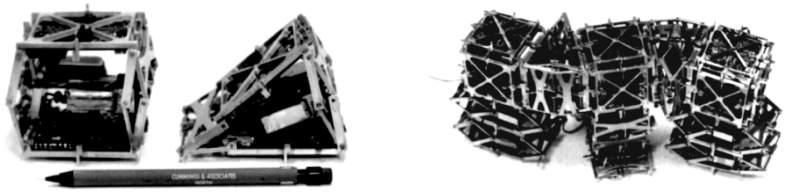
This section summarizes the current development in modular robotics, showing the most significant modules in the field, as well as the most related to the design developed by the author, and presented in chapter 4.

2.1.1 PolyPod

PolyPod [55, 54] is a modular robot created by Mark Yim in 1994 for his PhD thesis, and can be considered the first robot created with the modular robotic paradigm in mind.

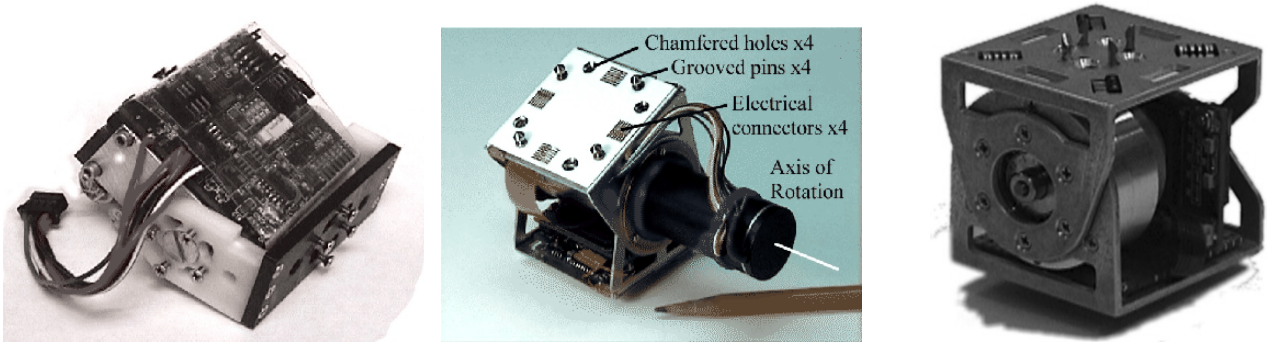
PolyPod is a heterogeneous system made of two different types of modules, called "segments" and "nodes". "Segments" are two degree of freedom parallel mechanisms composed of 10 links, resulting in a mechanism similar to two prismatic joints joined together by a revolute joint where the prismatic joints are constrained to have the same length. One of the degrees of freedom is linear, contracting and expanding the module from 1 inch up to 2.5 inches, and the other a revolute joint with a range of $[-45, 45]$ degrees. They also have 2 connectors that allow linear configurations and, in order to enable non-serial configurations, "Nodes" are used. "Nodes" are squared modules of aproximately 5cm x 5cm x 5cm with 6 connection ports that contain the gel-cell batteries for powering the robot.

PolyPod has dynamic reconfigurability, it can change its own shape by itself, adopting the configuration most suitable for each task.



(a) PolyPod segments (b) Robot made of PolyPod modules

Figure 2.1: PolyPod



(a) G1v4 (b) G2 (c) G3

Figure 2.2: Three generations of PolyBot modules

2.1.2 PolyBot

PolyBot[57] is a modular robot developed by Mark Yim at the Palo Alto Research Center (PARC), as the evolution of the PolyPod robot. It has been designed with space missions in mind, resulting in a small and light module. It is currently formed by G3 modules, which are the third generation of modules developed for PolyBot.

The first generation modules were simple modules with one degree of freedom made from laser cut plastic with genderless symmetric passive connectors joined by screws, and a hobby servo for joint motion. They were not able of self-reconfiguration and the power and computations were given externally.

The second generation modules were made from laser cut stainless steel, and the joint was actuated through a brushless motor that laid partially outside the module body due to the size of the gearbox. The connector was also upgraded with IR sensors and shape-memory alloy actuators for self-reconfiguration and active attachment/detachment. Communication among modules was carried through two CAN buses.

For the third generation of modules, a smaller custom made gearbox was added to the brushless motor so that it would be contained inside the module. Power consumption was reduced and several improvements were introduced to the connectors, allowing them to make passive connections and increasing the IR accuracy.

2.1.3 CONRO

CONRO[11] is a self-reconfigurable robot developed at the University of Southern California with search, rescue and surveillance operations in mind. The modules for the CONRO robot are fully autonomous, and are divided into three main sections: a passive connector, an active connector and the main body.

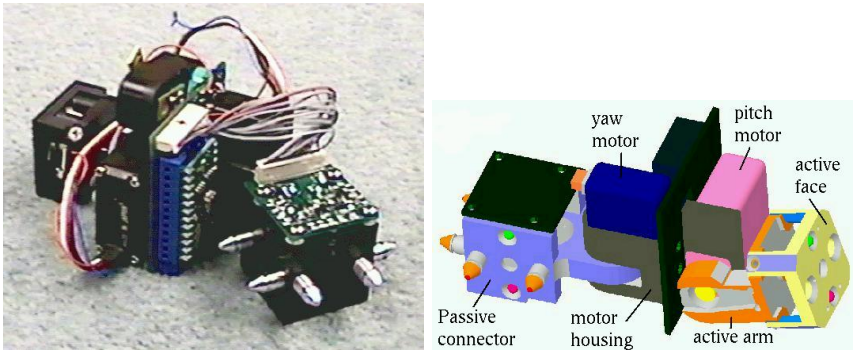


Figure 2.3: CONRO module

The passive connector is a plastic cube with a pair of pins in three of its faces. These pins are made from aluminium and have a cylindrical shape, with a groove to allow the active connector to lock them. The passive connector is hollow, and holds two 160mAh lithium batteries of 3V and 6V and the IR serial communication trasmitters, receivers and control circuitry, which allows the module to communicate with its neighbours when connected and also in the docking process. The IR system also works as a position feedback information to position the modules correctly while docking.

The main body holds two hobby servo which are connected with the passive and active connectors respectively and provide two degrees of freedom for the module. The body also contains the control board, which as a zero insertion force socket that allows the module to use three different microprocessors depending on the task requirements: a Stamp II based on a PIC16C57 or a Stamp IIe or II-SX based on a SCENIX SX28AC/SS processor.

Finally, the active connector is equipped with a pair of holes for the passive connector pins and a latch for holding the modules together. Connection of the two modules is passive, whereas a shape-memory allow wire allows the disconnection of the module.

2.1.4 Superbot

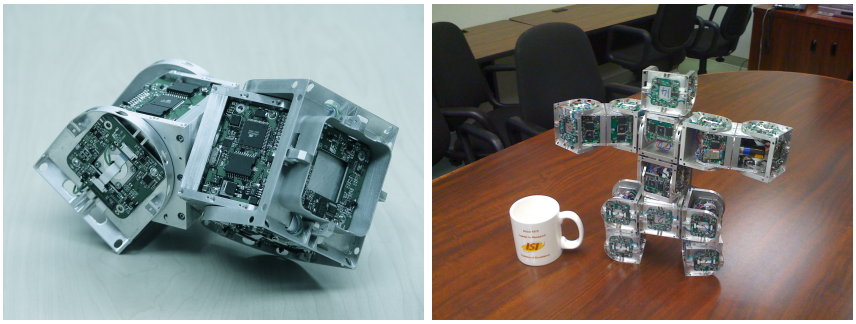


Figure 2.4: Superbot module

The Superbot[43] modular robot is the descendant of the CONRO module, and has been developed by the Polymorphic Robotics Laboratory at the University of Southern California. Funded by the NASA to be used for space applications, it is a hybrid module, as it can perform as a chain-type module or a lattice-type module.

Superbot modules are made from two cube-like bodies of 84x84x84mm that have each one a degree of freedom. The joint between both cubes can rotate about 270º so it has a total of three degrees of freedom, which allows the module to move freely on a plane.

Each degree of freedom is actuated by a DC motor equipped with a planetary gearbox and an external gearbox, and controlled by a software PID which recieves the feedback information from a potentiometer coupled to the motor shaft.

The electronics design is modular and the main circuitry is divided into two boards, a master board on one half of the module, and a slave board on the other half. Each board has an ATmega128 microcontroller and both are connected through a I2C bus. Each one of the six connectors of the module has an IR communication system that allows communication with it neighbors and provide position and distance feedback for docking with other modules.

Power is supplied to the module by a 1600mAh, 7.4V lithium-polymer battery, and can be shared to the neighboring modules when needed through the connectors. This sharing process is controlled as a high-level routine by the microcontroller that manages this functionality.

2.1.5 M-TRAN

M-TRAN (Modular TRANsformer) [40, 35, 33] is a self-reconfigurable robot being developed by AIST and Tokyo-Tech since 1998. Their third and latest iteration of the robot is called M-TRAN III. M-TRAN is a hybrid module able to

Figure 2.5: M-TRAN module

perform as a lattice-type modular robot for reconfiguration and as a chain/tree type for displacement.

The module is made from two semi-cylindrical boxes and a link joining them together. One of these boxes is active, and has three connectors with hooks that are able to connect with the passive box of the other modulues. This design improves the previous one (used in M-TRAN and M-TRAN II) that used permanent magnets for connection and a shape-memory alloy coil for detachment, as it is several times faster (around 5 seconds for M-TRAN III system versus nearly 1 minute for the previous systems). Each box is able to rotate 180º around its joint with the link, giving M-TRAN a total of 2 parallel degrees of freedom. While in lattice mode, this joints are actuated only in multiples of 90º, allowing a checkerboard pattern in which active connectors coincide with passive connectors for reconfiguration.

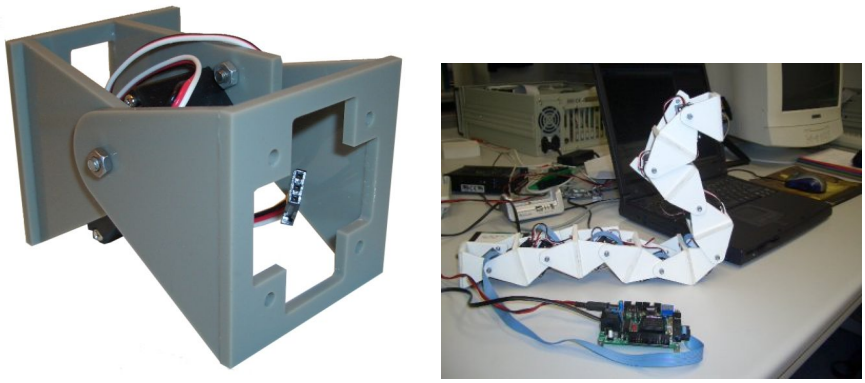
For the robot control, M-TRAN has four microcontrollers in total: one as master, that carries the main high-level behaviour and three slaves, that are in charge of several subsystems as the motor control, the communication system or sensors like the 3-axis accelerometer. It has several communication methods, such as bluetooth, IR and even a physical CAN bus through some pins on the connectors, which allows the modules to communicate no matter if they are physically in contact or in different assemblies.

A battery and a power supply circuit are placed in the passive box of the M-TRAN module, supporting autonomous operation.

2.1.6 Y1

Y1[18] modules were developed by Juan Gonzalez-Gomez at the Autonomous University of Madrid based on the first generation of PolyBot modules. The main objective of the Y1 modules was to create a cheap and open platform for researching modular robotics.

The Y1 modules are composed of a cheap hobby servo and a two-part housing made from laset cut PVC. This modules



(a) Y1 (b) Cube Revolutions (Y1 snake)

Figure 2.6: Y1

can be connected in a linear configuration, either pitch-pitch or pitch-yaw, allowing the resulting robots to move in a plane. Reconfiguration has to be done by hand, as the modules are joined by means of screws. The control electronics and the power source are supplied off-board.

The Y1 module is open source hardware, which means that the plans, the files for manufacturing and the assembly instructions are available online for anyone to access to them.

The REPY-2.1 module developed by us and used for testing the gaits of this thesis is based on the Y1 module. More details about the Y1 and REPY-2.1 modules can be found in chapter 4.

2.2 Gait Generation on Modular Robots

The lower level control for locomotion in mobile robots with wheels or tracks is usually not complicated, since it only involves turning the motors in order to produce movement. Difficulties appear at the higher level control, such as motion planning or navigation.

But, when the robot is articulated, either with legs or apodal, the lower level control becomes more complicated, as the problem of coordination appears, even if the robot is to travel through a flat surface without any kind of obstacles. In this kind of robots the movement of each of the joints must be coordinated with the movement of the other joints so that the robot can move. In more formal terms, the coordination problem can be stated as follows: *For a robot with N articulations, find the value of each joint as a function of time $\varphi_i(t)$ so that the robot can achieve locomotion.* The solution of this problem is not unique, and depends on the type of gait that one desires to obtain (e.g. walking on a straight line, turning, trotting, galloping, etc).

In order to solve this problem, several approaches can be found in the literature, and they will be explained in this section.

2.2.1 Gait tables

Gait tables are tables that include the joint position values for each module at different steps of a gait. For each moment of the gait, each of the modules look up in the table what is the required joint value for the current step, and move their joints to that value. When the robot arrives at the end of the table, it starts again with the first position of the table, achieving a repeating pattern.

This is an easy and simple way to implement locomotion gaits for modular robots, and was first used by Mark Yim in his PolyPod [54]. When the robots are composed of few modules and the desired movements are simple, these gait tables can be formed by hand, allowing a fast exploration of possible gaits for locomotion, as well as detecting mechanical defects on robot prototypes. Gait tables are not only useful for designing gaits for modular robots, but they can also be applied to other types of robots, such as quadruped, hexapods or even humanoid robots.

The main disadvantage of gait tables is that they lack flexibility, as for generating new gaits, or variations of the existing gaits, a new control table has to be created.

2.2.2 Central Pattern Generators (CPGs)

Central pattern generators (CPGs) are biological neural networks capable of producing coordinated patterns of rhythmic activity without any rhythmic inputs from sensory feedback or from higher control centers [27]. They are in charge of many rhythmic behaviours in both vertebrate and invertebrate animals, such as breathing, locomotion, bowel movements, etc.

CPGs are distributed networks composed by multiple coupled oscillatory centers, as observed in experiments with lampreys and salamanders, in which small sections of their spinal cords were capable of producing rhythmics activity. The lamprey is one the vertebrates most used to study CPGs, because its spinal column is transparent, contains few cells,

and lasts at least a week outside of the animal (in a saline solution) without deterioration [41], easing the work of the biologists.

Sensory feedback is not needed for the generation of the rhythms, but plays a key role in shaping the rhythmic patterns and keeping the body movements and the CPG coordinated. This coupling is so tight that is posible to induce CPG activity mecanically moving the tail of a lamprey, and to induce a normally looking walking gait in a decerebrated [1] cat by placing it on a treadmill. If the treadmill is accelerated, the gait can even change from trot to gallop.

The complex locomotion behaviour generated by the CPG circuits is controlled by simple signals, that in many vertebrates are generated in a specific region of the brain stem known as *Mesencephalic Locomotion Region* (MLR). Some experiments with electrical stimulation of this region have shown that the level of stimulation can modulate the speed of locomotions, and induce an automatic gait transition (from walk to trot to gallop on derebrated cats, and from walk to swimming on decerebrated salamanders). Therefore, basic rhythmic patters are generated at the spinal CPGs, but the modulation of those patterns according to environmental factors is controlled by the higher-level centers, such as the motor cortex, cerebellum and basal ganglia.

These biological CPGs have been matematically modelled as differential equations, and successfully applied to different robots, such as the *Salamandra Robotica*, a salamander-like robot developed at the EPFL [28]. Since CPGs are a distributed approach, it has been also applied to modular robots, such as YaMoR [38] or the Roombots [48]. One advantage of using CPGs is that the transition between two steady state oscillations is bounded, continuous and relatively smooth, so if we are implementing an optimization algorithm for the gaits, and we randomly change the control parameters, the joint values will not change too abruptly, which helps preventing the motors from breaking [27].

2.2.3 Sinusoidal Oscillators

CPGs are very powerful, but they are complex and require lots of computing power. For that reason some researchers tried to substitute on their controllers the CPG model obtained by neurocomputing scientists by a simpler one that performs in a similar way, but using less resources. CPGs behave as fixed frequency oscillators in steady state, making sinusoidal oscillator a suitable canditate for being used as gait generators, as they are much simpler to model and require less resources for their implementation than CPGs. Implementations of sinusoidal oscillators have been tested successfully on snake robots by Lipkin et al., who also defined piecewise functions to perform specialized tasks, such as stair climbing [37], and Gonzalez-Gomez [19], who also studied the minimal configurations required for locomotion with sinusoidal oscillators [20]. Sinusoidal oscillators also been applied to legged robots such as the hexapod robot Melanie-III with success [7].

Other approaches simplify even more the coupled oscillators of the CPG approach, substituting them by a single sinusoidal waveform as a function of time and current module, that is used to control the all the joint positions [50].

Due to its simplicity, sinusoidal oscillators are the approach selected in this thesis to solve the coordination problem, and are described in detail in chapter 5.

2.3 Coordination and Communications on Modular Robots

Communication between robots is essential for achieving coordination. This becomes of special importance when working with modular robots, as because of their nature, they require their modules to colaborate with each others to complete a task. In order to achive reconfiguration or locomotion the modules need to know what their positions inside the modular robot are, what tasks or steps have been completed and which remain still pending for completion.

On modular robots, these communications are implemented on hardware in very different ways. Some of them, such as PolyPod, M-TRAN or SYMBRION / REPLICATOR have physical connectors that are used for communication when

[1] Decerebration is the elimination of cerebral brain function in an animal by removing the cerebrum, cutting across the brain stem, or severing certain arteries in the brain stem [3].

two modules are attached together [56, 34, 36], whereas most of the existing modules, such as CONRO, SUPERBOT or ATRON, communicate using IR or IrDA communications [12, 44, 9], which have the advantage of being wireless and that can be also used as distance sensors. A few modules, such as YaMoR or M-TRAN, can communicate using Bluetooth communications, allowing modules that are not physically attached to interact with each other[39, 34].

Communications between robots can be classified as global or local communications. In local communications, robots only talk to their nearest neighbors and information is shared locally. This approach is tipically used in modular robotics to find the topology of the robot and to coordinate local tasks. On the other hand, when using global communications all modules can communicate between each other and achieve coordination of tasks involving distant modules.

The type of communications a certain module can perform conditions its control strategy, modules with a global communication system usually apply centralized control methods, such as PolyPod's central gait tables[56] or M-TRAN's centralized central pattern generators[32] whereas modules with local communications tipically use distributed control methods, like CONRO's and SUPERBOT's distributed digital hormones [46].

Local communication methods are used mainly for coordination of reconfiguration in lattice-type modular robots and for coordination and synchronization of locomotion movements in chain-type modular robots. Butler et al. described a renconfiguration method for lattice modular robots using only local information based on cellular automata, a simple set of rules that control the reconfiguration steps depending on the neighbors attached to the module, and on the obstacles detected, allowing a flow-like locomotion[10]. Funiak et al. presented a distributed method for module location inside large modular robots, consisting in breaking the cluster of modules in smaller clusters using normalized cut to identify dense sub-regions with small mutual localization errors[16].

For locomotion synchronization and coordination the most notable distributed approach is digital hormones. Digital hormones are a nature-inspired communication method published by Shen et al., based on biological hormones, that consist on signals or messages that are propagated by the modular robot, triggering different actions depending on the function of the module that receives them. Those local actions are executed by the modules without the help of the hormone, and include joint movement and hormone manipulation and destruction, among others [45].

For this thesis digital hormones were selected as communication method due to their proved usefulness in distributed control of locomotion in chain-type robots [46, 26]. Digital hormones and our developed algorithm will be described in detail on chapter 6.

Chapter 3

Software

In this chapter we will describe and explain the software framework implemented to work with modular robots and test the locomotion gaits and digital hormone-based controller. This framework is named *Hormodular*, a combination of the terms "hormone" and "modular".

All the software developed for the *Hormodular* framework is open source, and can be found in the following github repository: `https://github.com/David-Estevez/hormodular`. Being open source, anyone can download, use and study the code freely. This is very important in research, as any researcher interested in modular robotics in any part of the world can use this code, learn from it, improve it or repeat the experiments described in this thesis to test the validity of the results presented here.

This chapter will start explaining what are the software dependencies used in this project, then we will describe the Test-Driven Development methodology followed to develop the project. Next, the software structure will be discussed, with a detailed description of all the different classes implemented. Finally, the compilation procedure and program usage are offered for anyone interested on downloading and using this software.

3.1 Software dependencies

Due to the complexity of the project, some specific tasks required the use of third-party libraries perform them. According to the open source nature of our project, these libraries were chose to be also open source, and compatible with GNU/Linux systems. These libraries are in charge of tasks such as simulation of the modular robot, optimization of the oscillator parameters in order to achieve locomotion or parsing XML configuration files.

3.1.1 Simulation software: OpenRAVE

The simulation software used by the student for this project is OpenRAVE. OpenRAVE [15] is an Open Source project developed by Rosen Diankov that provides simulation tools for working with robots and trajectory planners. It can be embedded on other controllers and larger frameworks, and its functionality can be extended by means of plugins, adding new trajectory planners, controllers, collision checkers, inverse kinematics solvers, robots or sensors as needed.

OpenRAVE was chosen over other simulators due to its open source nature and its extensive documentation [14], as well as its easy integration with larger projects, such as this one.

3.1.2 Modular Robotics plugin for OpenRAVE: OpenMR

OpenMR [25] is a plugin for the OpenRAVE simulator developed by Juan Gonzalez-Gomez that allows the user to simulate servo motors on the joint of OpenRAVE robots.

It adjusts the angular velocity of the robot joint by means of a PD controller feedbacked with the joint angular position, so that the user can control the joint position value with a simple interface.

3.1.3 Evolutionary Computation Framework: ECF

The Evolutionary Computation Framework (ECF) [30] is a C++ library that allows the user to apply several evolutionary optimization algorithms in a very customizable way. It offers several common evolutionary optimization algorithms such as particle swarm optimization (PSO), differential evolution (DE), genetic annealing, artificial bee colony (ABC), and genetic algorithms with steady state tournament and generational roulette-wheel selection, among others.

By means of a XML file, the user selects the optimization algorithms to be used, and its parameters. That XML file also contains information about the genotype that will encode the different parameters to be evolved. The user can specify his own fields that can be later read by the software containing the ECF to set user-defined paramters.

The interface of the ECF is very simple, and to optimize a given function the user just has to inherit from the base class "EvaluateOp", registering the custom parameters needed by that class to be extracted from the configuration XML file, and specifying the actions to me made for the initialization and evaluation of the function with a given genotype.

In this case the function to be evaluated is the distance travelled by the modular robot running for a certain period of time using the oscillator parameters specified by the genotype.

3.1.4 XML parsing: TinyXML2

TinyXML2 [51] is an Open Source lightweight C++ library that supports Document Object Model (DOM) parsing of XML with a very small memory footprint. DOM parsing is a cross-platform and language-independent convention for representing tagged documents such as HTML, XHTML or XML files, and means that the data inside the xml file is represented as a tree in which each node is an object that can be addressed and manipulated.

TinyXML2 can be easily integrated in a project with almost any configuration required. It is also has a very simple API, is fast and requires a very small amount of memory, being those the main reasons for using it in this project, instead of coding a XML parser from scratch or using a bigger XML parser.

XML files are used in this project for storing the robot configuration in a way that is easy for a human to setup manually, and also simple for a machine to extract that information later.

3.1.5 Linear algebra library: Eigen

Eigen [29] is an Open Source C++ library for performing linear algebra operations with matrices, vectors and algorithms related to them. It is implemented as a template library that only includes header files, and supports matrices of all sizes and numeric types, including integers, floating-point numbers and complex numbers.

Eigen also support homogeneous transformation matrices, vector-axis pairs and quaternions, that are used in this project to obtain the relative orientation between two modules from the data obtained by the simulated IMU.

3.1.6 Software build tool: CMake

CMake [2] is a cross-platform, open source build system designed to build, test and package software. Using simple and compiler-independent configuration files, CMake allows to control the build and linking process, generating all the files and environments required by the compiler chosen by the end user to build the code.

CMake supports building programs that require several libraries, cross-compilation, as well as complex directory hierarchies and it is used in many large open source software projects, such as Blender, KiCAD, OpenCV, Point Cloud Library or Qt5.

3.2 Development Methodology: Test-driven Development (TDD)

For the development of the software of this project, the author has used a software development methodology called "Test-driven Development (TDD)", part of a larger set of development methods called "Agile software development". "Agile software development" methods allow a fast development of the project based on an iterative and incremental process, in which the code evolves from a simple version to the final one by adding functionality little by little as required by the project.

"Test-driven Development" methodology [8] is based on a short development cycle that is repeated iteratively, in which a new desired feature or improvement is used in a test case that initially fails (since there is no code for performing that feature yet). Then, the programmer implents the minimum code required to pass that test, and finally refactors the code into a clearer and more mantainable form.

Since every functionality in the code has a test associated, the code has a greater maintainability. Extending or improving the code becomes easier, faster and safer, since the tests can be run before commiting changes to the repository, ensuring that the new code does not break any old functionality. For example, when a team works on a software project, TDD allows them to check that the new code each member adds does not interfere with the old tested code, reducing the number of bugs and reducing the code interdependency.

Tests also help programmers understand the code written by other people. Since the test uses the code (functions, classes, etc) in order to test them, they can be used by programmers as a reference of how that code is supposed to be used and behave.

3.2.1 Test-Driven Development main cycle

The main cycle of developing a new feature or improvement with a Test-Driven Development methodolody is the following:

1. **Add a test for the new feature or improvement.**
   When a new feature or improvement is required, a new test is written. Since there is no code yet that implement that feature, this test must fail. The test must check all the requirements of that particular feature to be added. That way, the developer can focus on the requirements and not add any unnecessary code.

2. **Run all tests. Check that the new one fails.**
   This steps checks that all previous tests pass without any issue, and that the new added test fails. The new test must fail, because there is no code yet for that feature, and that allows the developer to check whether the test was correct or not. A new test that passes without the addition of any code is useless for the programmer, since it does not check the new code to be implemented.

3. **Write the code for the new feature.**
   In this step the developer implements the minimum code required to make the test pass. This code does not need to be as eficient or elegant as the original, it just has to pass the test. This code is not definitive, and it will be improved and refined in later steps.

4. **Run tests.**
   At this point the developer has to run all the tests, including the new one. This will ensure that the code developed meets the tested requirements, and that the new code does not break any previous functionality tested in other tests.

5. **Refactor code.**
   The tested code has to be now cleaned up in order to be eficient, elegant and maintainable. In this step the code is moved to where it belongs logically within the project and encapsulated in a class if needed. If there is any duplicated piece of code, the developer should remove it, for example, by creating a function that encapsulates it.

He has also to check whether the names given to functions and variables make sense and are representative of their current use. All these actions are called "refactoring code".

By running againt the tests, the programmer can be confident that the refactored code still performs as required, and that the older code works as expected.

6. **Repeat.**
   Since this is an iterative process, it has to be repeated for each new feature to be added, incrementing the overall functionality of the project. These increments have to be small enough to allow a fast development and to minimize the amount of time spent in debugging the code.

3.2.2 Test-Driven Development example

In order to explain better the development cycle under a "Test-driven development" methodology, we will present an example. In this example, we will develop a simple calculator that performs sums and wrap it on a C++ class. For the test we will use Google's GTest C++ Framework, that will be described in detail in section 3.2.3.

The first step is to define what is the functionality that the new code has to perform, and write the first test. In this case the functionality is to perform a simple sum, so the test will be very simple. One will usually test the error-prone or critical cases, in this case, we will test the sums of some positive numbers, the sum of positive and negative numbers, and the addition of 0. If we were developing the division function, for example, we would test the behaviour when dividing by 0, checking for example that the code raises an exception when the user tries to divide by 0.

The test would look like this:

```
class TestCalculator : public testing::Test
{
       /* These class will contain the elements common to all the test,
       but for now it is empty */
};

/* TEST_F stands for test fixture, test fixtures use a test class inheriting from testing::Test,
like TestCalculator. The first parameter is the testing::Test class to be used, and the second
one is the name of the test to be implemented. */
TEST_F( TestCalculator, calculatorAddsPositiveNumbers)
{
       /* EXPECT works as an ASSERT, but if the condition is not accomplished, it continues the rest
       of the test, reporting the error at the end. The expected result is placed as first argument,
       and the expresion to be evaluated is placed as second argument.*/
       EXPECT(42, calculator.add(21, 21));
}

TEST_F( TestCalculator, calculatorAddsZero)
{
       EXPECT( 5, calculator.add(5, 0);
}

TEST_F( TestCalculator, calculatorAddsNegativeNumbers)
{
       EXPECT(-1, calculator.add(1, -2);
}
```

If we try to run this test, it will not even compile, as calculator, which is a instance of the class Calculator has not been created. In fact, the class Calculator has not been declared, so we will add it to the test, and we will instantiate it

to be able to run the test:

```
/* We add this class to the test file. Note that we did not implement the add method yet. */
class Calculator
{
  public:
    int add( int a, int b);
};

/* We modify the test class, adding the calculator instance to be tested. */
class TestCalculator : public testing::Test
{
  public:
    Calculator calculator;
};
```

When this test is run, it should not pass any of the tests, since the functionality has not been implemented yet. Once we have checked that the test fails, we will implement the requested functionality:

```
/* We implement the calculator class in the same file */
int Calculator::add( int a, int b)
{
    return a+b;
}
```

Running the tests now will result in all the tests passing. After this code we would refactor all the code to a more maintainable form. In case we wrote the code directly in the test, we would put it on a function or class. Since we did use a class from the beginning, refactoring will consist on moving the class to its own header file "Calculator.hpp". After we have refactored the code, the tests must be run again to check that the code still passes them.

If we want to add more features to the calculator, such as subtraction, multiplication or division, we would repeat the process again from the start, creating a new test for the new operations to be implemented.

3.2.3 Google Test (GTest)

Google Test (GTest) [4] is a framework for writing tests in C++ developed used by Google in their software projects, and released publicly under a Open Source license. By means of several macros, one can add tests that are automatically discovered by GTest, as well as assertions to be ensured by the code.

Different tests of the same class or library can be grouped in a test fixture. As seen in the previous example (section 3.2.2), GTest provides a class *testing::Test* that can be used to set up the data or prerequisites for the test. The programmer can define a class that inherits from *testing::Test* and, before each test of the test fixture is run, the *SetUp()* method implemented by the programmer will be called by GTest in order to setup the required elements used in the test. After the test is finished, the *TearDown()* function will be called by GTest in order to perform the cleanup of the used elements, or to free the allocated memory.

Each new test is added by using the *TEST()* macro, taking as argument the name of the test. If the tests are going to be grouped in a test fixture that uses the same data or instances of the class to be tested, the *TEST_F()* macro can be used instead. The *TEST_F()* macro takes two arguments: the first one is the name of the class inheriting from *testing::Test* that will prepare the data for each test, and the second one is the name of the test to be implemented. GTests are recognized at compilation time, and integrated in a GTest application, that runs the tests and shows a report like the shown in figure 3.1.

Inside the test, the conditions that have to be ensured are checked with the macros *ASSERT()* and *EXPECT()*. The

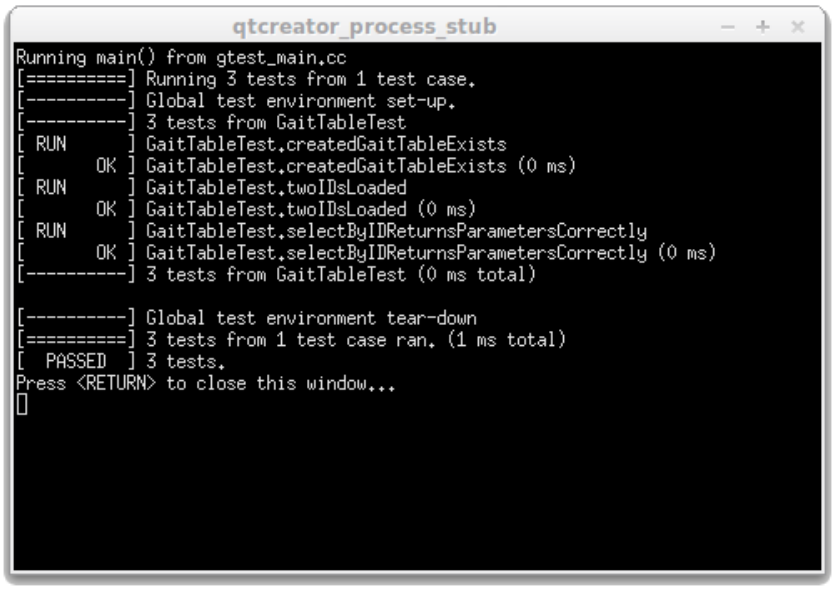


Figure 3.1: GTest report

main difference between them is that if *ASSERT()* is used, the execution of the test ends if the condition imposed is not met, whereas if *EXPECT()* is used instead, the execution of the test continues even thought that condition fails. One typically uses *ASSERT()* when the test cannot continue if the condition fails, or if it has no sense to continue with the test if the condition fails. On the other hand, using *EXPECT()* allows the program to continue testing the code, so that if more than one bugs are present they can be found on the same run of the test, and corrected at the same time, speeding up the debugging process.

3.2.4 Hormodular tests

To develop the software of this project a test-driven development was followed, and several tests were consecutively implemented, increasing the project functionality until the project was completed. Here were will describe what functionality is tested on each of the tests.

- **TestConfigParser**: Tests that the class *ConfigParser* is able to parse a test configuration file and extract from it the configuration parameters.
- **TestConnectionsFromConfigParser**: Tests the module interconnection. It creates a series of modules and attaches them according to the information stored in a *ConfigParser*, checking the connections. After that, runs the hormone communication protocol and tests that the IDs calculated by the hormones are the correct ones.
- **TestGaitTable**: Tests the main functionality of a gait table: loading the data from a text file and returning the parameters stored correctly.
- **TestModularRobot**: Creates a *ModularRobot* with a *SimulatedModularRobotInterface* and tests that the *ModularRobot* is able to move at least 10cm in 25ms.
- **TestMovement**: Creates a series of *SinusoidalOscillators* with the parameters required for a 2-module snake robot to move in straight line ($A = 60^o$, $O = 0^o$, $\Delta\phi = 120^o$, $T = 1s$) and sends the joint position to the simulated module using a *SimulatedModularRobotInterface*, testing that the snake robot moves more than 10cm.
- **TestMovementWithGaitTable**: Similar to the previous test, but in this case the parameters are loaded on a *GaitTable* from a file, and later retrieved from the *GaitTable* and set on the *SinusoidalOscillator*.
- **TesOrientation**: Tests the different mathematical operations that can be performed with the *Orientation* class, such as sums and substractions. It also tests that the calculation of the relative orientation between two connectors is performed correctly.

- **TestSerialCommSinusoidal**: Tests the connection with the robot by opening a serial port and sending to the robot joint values that follow a sinewave.
- **TestSerialModularRobotInterface**: Tests that the modular robot joints move when the joint values are sent with the *SerialModularRobotInterface*. It also tests toggling the LED on the controller board.
- **TestSimulatedModularRobotInterface**: Tests that the simulated robot joints move when the joint values are sent with the *SimulatedModularRobotInterface*.
- **TestSinusoidalOscillator**: Tests that the *SinusoidalOscillator* outputs values according to a sine function.

3.3 Software structure

The software was developed with modularity and code reusability in mind, defining several interfaces that help to add new features or implement the existing ones in a different way. The main class of the project is the *ModularRobot* class, that represents a modular robot made of a series of modules. With this class is possible to test different controllers for the modules, use different kinds of oscillators to generate the locomotion gaits or interface with different modular robots (both simulated and real). In this section the general structure of the code will be explained, including a detailed description of each of the different classes that compose the *ModularRobot* class, and their function in the project.

The class *ModularRobot* models the whole modular robot as a set of *Modules*. Even thought the controller is distributed in nature, the hardware used to test the gaits is centralized, having only a single controller board, so this class is needed to join the distributed controllers into a single robot encapsulating them in order to communicate with the hardware. The *ModularRobot* is configured using a xml file read by the *ConfigParser*, which uses the TinyXML2 library to load the configuration parameters in the xml file to a data structure that the *ModularRobot* and the *Module* can access for setting their parameters.

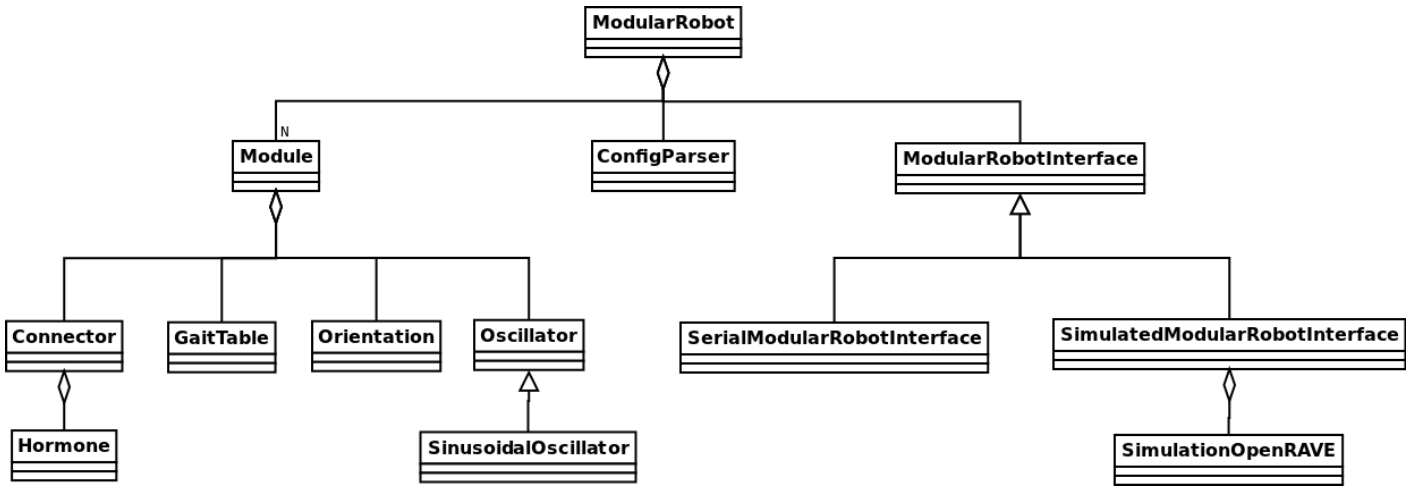


Figure 3.2: Main class diagram

Each *Module* has several *GaitTables* to store the parameters of the *SinusoidalOscillator* for the different configurations. At each step of the controller, the *SinusoidalOscillator* class calculates the joint position according to the oscillation. The *SinusoidalOscillator* can be changed by another kind of oscillator thanks to the *Oscillator* interface. The joint angle calculated the *Oscillator* in each *Module* is sent to the robot using a *ModularRobotInterface* interface. *ModularRobotInterface* offers a interface so that several types of robots can be used, either simulated (using the *SimulatedModularRobotInterface*) or real robots communicated through a serial connection (using the *SerialModularRobotInterface*).

A *Hormone* class was defined in order to implement the hormone-communication protocol. *Hormones* are sent and received by the any of the four *Connectors* present in each *Module* that model the interconnection of the different modules,

allowing the *Hormones* to flow through the modular robot.

Finally, the *Orientation* class is a data structure for storing the Tail-Bryan angles (Roll, pitch and yaw) that are obtained by the simulated Inertial Measurement Unit.

3.3.1 Class ModularRobot

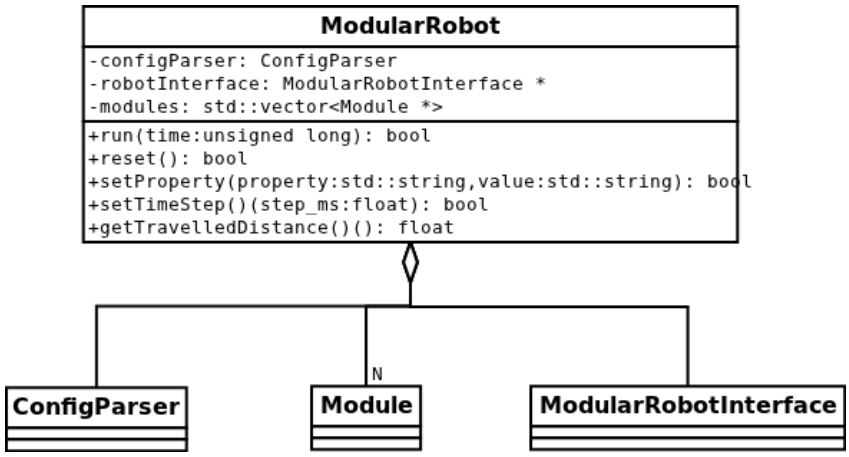


Figure 3.3: ModularRobot class diagram

ModularRobot is the class that encapsulates both the modules and interface with the actual robot (either simulated or real). It counts with several functions that control the robot to perform several operations such as starting or resetting the controllers of all the modules, etc.

A *ModularRobot* is constructed using a *ConfigParser* containing the configuration of the modular robot, read from a xml configuration file, which determines the number of modules to be created. A suitable *ModularRobotInterface* is also created, depending on if the robot to be controlled is a real one or a simulation of the modular robot.

A series of functions exist to configure the behavior of the *ModularRobot* after the object is created. The function *setTimeStep()* configures the resolution of the simulation for the simulated robot and the period between packet transmission for the robot controlled by serial port, and using *setProperty()* one can configure other aspects such as enabling/disabling the simulation viewer.

After the robot parameters are configured, the module interconnections are read from the *ConfigParser*, and the connectors of the different modules are connected together according to that configuration. This interconnection allows the communication of the different hormones between the modules.

Once the robot is configured, and its modules are connected, to start it, the function *run()* is called passing the amount of time, in ms, that the robot will be active. After that amount of time the robot will stop until it receives another call to the *run()* method. Before running again the robot controller, is recommended to make a call to the function *reset()*, to restore the initial configuration, position of the simulated robot, etc.

Even though the controller is distributed, and it is supposed to be run in each of the modules independently, the current implementation is simplified to a sequential execution in order to test the hormone-communication protocol in a quick way, so that it can be validated or discarded on an early development stage. Concurrent software is typically difficult to develop and debug, since resources are shared between processes/threads and bugs may appear depending on the order in which those resources where accessed, which causes the appearance of bugs that are difficult of reproduce and fix, since that order is not deterministic. Other common bugs in concurrent software are corruption of data due to simultaneous access to unprotected shared variables or deadlocks (a process p_1 has a lock l_1 and it is waiting for another lock, l_2, which is held by a process p_2 which happens to be waiting for the lock l_1).

Because of those reasons, the different tasks to be performed by the module controller are implemented as independent member functions, and they are called sequentially for every module before executing the next one. This way a performance similar to the concurrent approach is achieved, but without the increase in development difficulty and time due to concurrent programming. Communications tasks are executed each T_{comm} ms, whereas the joint position is updated each T_{step} ms, allowing to update the joint values more frequently, as the communication tasks can be performed with a larger period, since it is not a time-critical task.

Figure 3.4 shows the flowchart of the *run()* function of class *ModularRobot*.

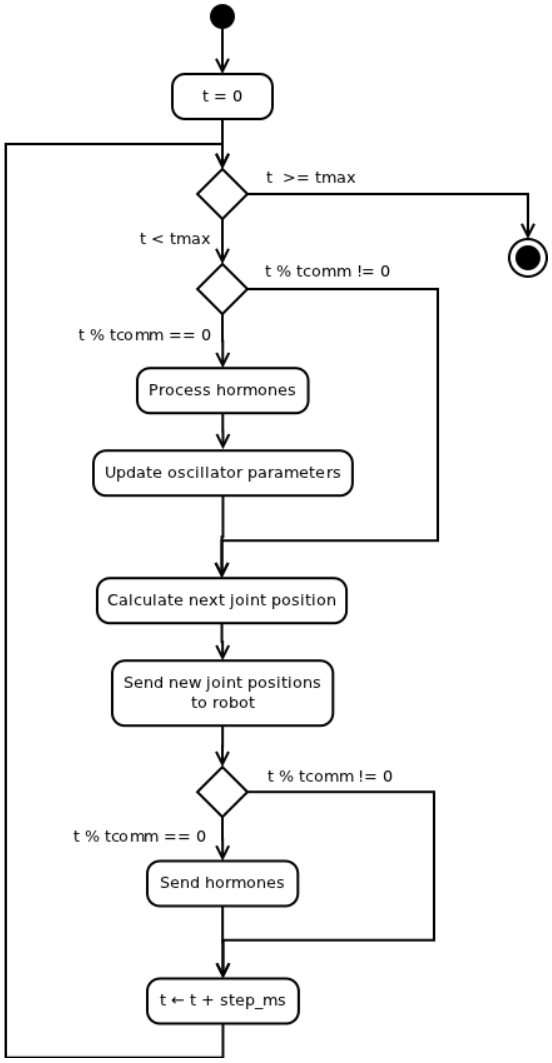


Figure 3.4: ModularRobot::run() flowchart

The whole function runs in a loop, repeated until the maximum runtime, t_{max}, is reached. Each t_{comm} ms the hormone functions are executed. These functions process the input hormones, obtaining the identifiers for the current module's position, function and global configuration, and setting the required hormones in the output buffers, ready to be sent to the other modules. After the different IDs have been calculated, the oscillator parameters are obtained from the gait tables and set.

The next steps, that are executed every t_{step} ms, are the joint values updates. First, the *Oscillator* calculates the new joint value for all the joints at the current time t and, after that, those values are sent to the robot (simulated or real) using the *ModularRobotInterface*.

Finally, also each t_{comm} ms, the hormones that were placed on the output buffers are actually sent to the other modules. The time counter is incremented and the loop repeats again until t_{max} seconds have passed.

It is important to notice that each of these tasks are executed for each of the modules on the modular robot before moving on to the next task, emulating this way a distributed, concurrent system.

3.3.2 Class ConfigParser

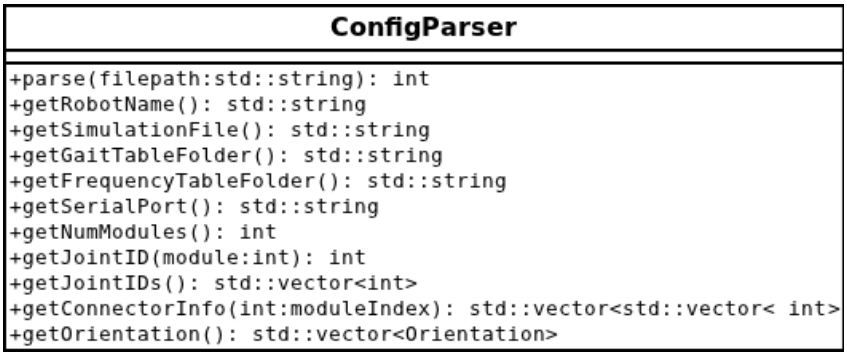


Figure 3.5: ConfigParser class diagram

Since the *ModularRobot* can be executed with different configurations, it is very convenient to have a way to load those different configurations dynamically without recompiling the whole software. The approach typically followed to achieve this is to use configuration files specifying the different aspects to be configured.

For the configuration files, the xml format was chosen, because it is standard, can be read and edited easily by both humans and machines, and there already exist several libraries for parsing xml from different programming languages.

The *ConfigParser* class uses one of those libraries, called "TinyXML2" to parse the configuration files for the different robot topologies and stores the different parameters loaded in a data structure that can be later accessed by the other components of the project.

The tag structure of the configuration files is the following:

- All the xml tags of the robot are enclosed on a parent tag called <*ModularRobot*>. This tag has an atribute *"name"* that contains the name of the robot.

```
<ModularRobot name="TestRobot">

    <!-- Robot config goes here -->

</ModularRobot>
```

- Inside <*ModularRobot*> go the global configuration parameters and the definition of the different modules. The global parameters to be configured are: the path to the openRAVE xml model of the robot for the simulation, with the tag <*simulationFile*>; the path to the folder containing the gait tables, with the tag <*gaitTableFolder*> and the path to the file with the table containing the frequencies of the oscillators for the different configurations, <*frequencyTable*>. The serial port used to communicate with the real robot is configured in the tag <*serialPort*>.

```
<ModularRobot name="TestRobot">

    <simulationFile>../../data/models/REPY-2.1/MultiDof-7-tripod.env.xml</simulationFile>
    <gaitTableFolder>../../data/gait tables/</gaitTableFolder>
    <frequencyTable>../../data/gait tables/frequencies.txt</frequencyTable>
    <serialPort>/dev/ttyUSB0</serialPort>

    <!-- Rest of the configuration goes here -->

</ModularRobot>
```

- Modules are defined after that, using the tag <*Module*>. Inside this tag, the different parameters of the module are to be set, such as the joint index (set with <*Joint*>), the initial orientation of the module and the different connections between modules.

  The initial orientation defined in the configuration file under the tag <*Orientation*> is the one the module takes as if it were the readings of the Inertial Measurement Unit, emulating this piece of hardware that the current module version lacks. The different values for the angles are set in the tags <*Roll*>, <*Pitch*> and <*Yaw*>, respectively.

  Connections between modules are set under the tag <*Connections*>. Each of the local connectors (front, right, back and left) has its own tag for setting the parameters of that connector. Those parameters are atributes of the corresponding tag, such as *connectedTo*, indicating which module is the current connector connected to; *connector*, which represents the index of the remote connector that is connected to the current connector (0 for front, 1 for right, 2 for back and 3 for left) and *orientation*, which represents the relative orientation of the connectors, and that currently it is only used for debugging and testing purposes.

  Here we have an example of a complete xml robot configuration file for a simple 2-module configuration:

```
<ModularRobot name="TestRobot">
    <simulationFile>../../data/models/REPY-2.1/Kusanagi-2.env.xml</simulationFile>
    <gaitTableFolder>../../data/gait tables/</gaitTableFolder>
    <frequencyTable>../../data/gait tables/frequencies.txt</frequencyTable>
    <serialPort>/dev/ttyUSB0</serialPort>
    <Module>
        <Joint>0</Joint>
        <Connections>
            <front connectedTo="1" connector="Back" orientation="0"></front>
        </Connections>
        <Orientation>
            <Roll>0</Roll>
            <Pitch>0</Pitch>
            <Yaw>0</Yaw>
        </Orientation>
    </Module>
    <Module>
        <Joint>1</Joint>
        <Connections>
           <back connectedTo="0" connector="Front" orientation="0"></back>
         </Connections>
        <Orientation>
           <Roll>0</Roll>
           <Pitch>0</Pitch>
           <Yaw>0</Yaw>
        </Orientation>
    <Module>
</ModularRobot>
```

3.3.3 Class ModularRobotInterface

Figure 3.6: ModularRobotInterface class diagram

Although the controller is distributed, the modular robot used to test the gait has a central controller, due to hardware limitations of the modules used in this work. For that reason, the *ModularRobot* class owns a interface with the modular robot, in order to send commands to it, so that it can set the joints of the robot to the desired values or, in the future, receive sensor data from the robot, to implement more complex controllers.

The class *ModularRobotInterface* acts as a bridge between the controller, being run on the computer, and the robot. As the robot to be controlled can be either a simulated one, for gait discovery and testing or the physical one, to verify that the gaits work on the real worl, this class is an abstact class that act defining the interface in a computer science sense, that is, the functions that the classes that follow this interface must define. This way, the different interfaces for the different types of robot can be used indistinctly by the modular robot controller, just having to change which class is instantiated to change the behaviour of the program.

Two different classes have been implemented following the *"ModularRobotInterface"* interface:

- *SimulatedModularRobotInterface*, to control the modular robot simulated on openRAVE.
- *SerialModularRobotInterface*, that controls the physical robot via the computer serial port.

Instances of those clases are created using a factory method pattern, a function that generates objects that follow a certain interface without needing to specify the exact class of object that will be created.

The interface has four main functions to implement for the initialization and destruction of the interface, one for the setup of certain properties or variables, and three for the control of the robot. The purpose of each of the function is the following:

- bool *start()*: this function is used for the initialization of the robot interface, for example, to start the simulation or to connect to the robot with the serial port.
- bool *stop()*: with this funtion the robot interface is stopped. This can be used, for instance, to stop the simulation or to interrupt the communication via serial port.
- bool *destroy()*: this function is meant to be called in order to free all the dynamically allocated memory and to do all the required steps in order to destroy the object.
- bool *reset()*: used to reset the interface, for example, restarting the simulation or the serial connection.
- bool *setProperty(* std::string property, std::string value*)*: this function should be used to configure the different properties or parameters of the interface. The string property specifies which is the property to modify, and the string value contains the new value for that parameter. This can be used, for example, to specify if the simulator viewer is to be visible or not.
- bool *getTravelledDistance()*: returns the distance travelled by the robot. The method used for calculating this distance depends on the implementation of the robot interface.
- bool *sendJointValues(* std::vector<float> joint_values, float step_ms = 0*)*: sends the desired joint values to the robot. The step_ms parameter can be used to specify the amount of time to wait for the joints to reach the position, in order to run the simulation for the time of that step, for example.
- std::vector<float> *getJointValues()*: returns all the joint positions stored in a vector.

3.3.4 Class SimulatedModularRobotInterface

This class implements the *ModularRobotInterface* interface, and it is used to start a simulation and control the simulated robot. For that purpose it uses a *SimulationOpenRAVE* objects that encapsulates all the details of starting a OpenRAVE simulation and controlling it.

If the simulation is to be run continously, it can be started with the *start()* function. If not, the simulation is run step by step, with the step time specified in the *sendJointValues()* method. The *stop()* and *reset()* methods stop and restart the OpenRAVE simulation and with the *destroy()* method the memory for the simulation object can be freed.

The user can decide whether or not the simulation viewer is enabled by calling the *setProperty()* method with the property "viewer" and the value "enabled".

For the travelled distance calculation, the initial position of the robot is stored at the start of the simulation, and when the function *getTravelledDistance()* is called, that distance is calculated by computing the distance between that initial point and the current position. Calculated this way, the distance is always less than actual distance travelled, unless the robot moves in a straight line, which favors that the gaits resulting from the evolutionary algorithm follow a straight line, as this way the fitness value (average speed) is greater.

The joint values are accessed directly through a reference to the OpenRAVE Controller that is set on the robot, both for sending the new joint values, or to read the current ones. A step time can be specified, so that the simulation is advanced that amount of time for the joints to have time to move to their desired positions.

3.3.5 Class SimulationOpenRAVE

In this work, OpenRAVE has been chosen as simulator, but many others exist. Having a class for encapsulating the simulation allows not only to add support to different simulators in the future, but also to offer a simple and generic

interface for interacting with the simulation.

The *SimulationOpenRAVE* class offers a control interface similar to the *ModularRobotInterface* one, with functions for starting, stopping and resetting the simulation. It has also a function *step()* to run the simulation step by step, instead of running it continuously.

It creates a OpenRAVE simulation, loads the environment with the modular robot to be simulated, sets the Servocontroller from the OpenMR plugin for controlling all the joints, gets references to the environment and robots, which can be later obtain with the *getPenv()* and *getRobot()* methods.

The viewer offered by OpenRAVE can be enabled either when creating the *SimulationOpenRAVE* object or later, calling the *showViewer()* method.

Class SerialModularRobotInterface

The *SerialModularRobotInferface* class implements the *ModularRobotInterface*, and it is used to send the joint values to the real robot through a serial connection.

When the *start()* method is called, the serial port is open and the program connects with the modular robot until the communication is interrupted by calling *stop()* or *reset()*. Once the communication is halted, the allocated memory can be freed by calling the *destroy()* method.

The joint values are sent to the robot using the function *sendJointValues()*, which internally converts the values from the range of the oscillator ($[-90, 90]$) to the range of the hardware servos ($[0, 180]$) before sending them through the serial port. Since the robot does not count with any means of measuring the actual joint position, the sent values are stored in the *SerialModularRobotInferface*, and can be requested by the user calling the *getJointValues()* method.

Since the modular robot currently cannot sense its actual position (using internal measurements or with computer vision employing an external camera), the *getTravelledDistance()* method just outputs a warning explaining that the distance travelled measurement is not implemented yet.

Finally, since the controller board has a LED available for visual signaling, a property called "LED" with a value "toggle" can be set to the *SerialModularRobotInferface* using the *setProperty()* method to turn it on and off.

3.3.6 Class Module

The class *Module* represents the controller of each of the modules. This controller is homogeneous for the whole modular robot, each module is an instance of the same Module class.

A *ConfigParser* is passed to the constructor with the parameters read from the configuration file, as well as the index of the module to be created, so that the correct parameters are extracted from the *ConfigParser*.

For generating the oscillations, the *Module* class has a *Oscillator* object, that updates the joint position of the module. For this thesis the oscillators used are *SinusoidalOscillator*, but as the *Oscillator* class defines a interface for oscillators, new kinds of different oscillators can be added very easily to the project, and used without changing the code for the *Module* controller.

The parameters for the oscillator are stored on several *GaitTables*, one for the main parameters (amplitude, offset and phase) for each of the three configurations discussed on this work, and a extra one for the different frequencies for each configuration. These parameters are accessed using the IDs obtained through the hormone communication protocol. How to calculate those IDs is explained on section 5.1.4, and the hormone communication protocol is presented on chapter 6.

One of the required data for calculating the IDs is the initial orientation of the module. There exists a class in this project, called *Orientation*, that stores the data for the orientation angles. It has also several functions to do simple

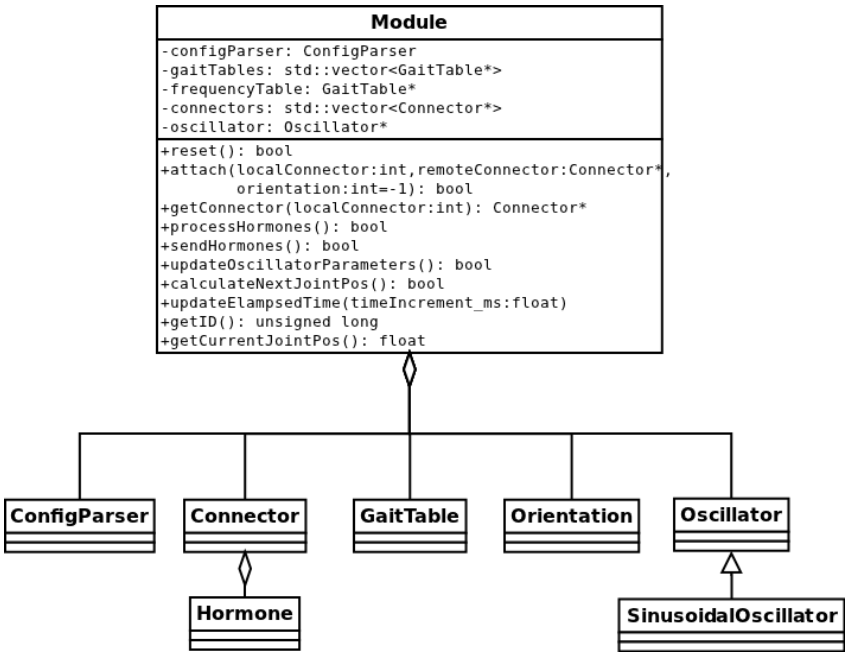


Figure 3.7: Module class diagram

operations with *Orientations*, as well as a function that calculates the relative orientation between the orientations of two modules, a piece of data that is key to the ID calculation.

For the interconnection of modules, and their communication, a class *Connector* was implemented. There are 4 connectors on each module, and each *Connector* has an input buffer and an output buffer to store the incoming and outgoing *Hormones*, and a reference pointing to the remote *Connector* they are attached to. The *Hormone* class models the different types of hormones used in the hormone communication protocol, to be explained in chapter 6.

In order to emulate the distributed controller from a sequential approach, so that the software can be developed and validated in a faster way, the controller of the module is split in several small tasks, that are executed for all the modules before starting the next one.

The first task to be executed is the hormone processing. Hormones are recovered from the connectors' input buffers and processed depending on their type, as explained in section 6.3.3. "Ping" hormones are processed first, obtaining from them the ID related to the position of the module inside the modular robot. Then, the "Leg" hormones are either processed or generated in case of the "leg" modules. From the "Leg" hormones the "head" module will discover which is the global configuration of the robot and will generate "Head" hormones. Finally, if the module is not the "head" module, those "Head" hormones are processed, obtaining the configuration ID obtained by the "head" module. Generated hormones are put in the output buffer in this task, ready to be sent to the connected modules.

Once the IDs are obtained, the next task to be performed consists in querying the new oscillator parameters to the corresponding gait tables, and setting them on the oscillator. With the ID of the current configuration the corresponding gait table for the amplitude, offset and phase of the oscillators is selected, and using the ID related to the function inside the modular robot global configuration the suitable parameters are obtained from the table and set on the oscillator. This task, as well as the previous one, are executed periodically at a different rate than the joint position update, with a period of t_{comm} milliseconds.

After that, the oscillator will update the joint value for the module, calling *calculateNextJoinPos()* with the elapsed time as parameter. The modular robot will recover this joint position value, put it in a vector with the other modules' joint values and send them to the *ModularRobotInterface*.

Finally, the hormones stored in the output buffers will be sent to their destination modules, and the local elapsed time of the module will be updated.

3.3.7 Class Oscillator

In order to generate the oscillations that drive the joints of the modular robot in order to achieve locomotion, the *Oscillator* class is employed. This class is designed as an abstract class that stores the main oscillator parameters, and lefts the actual implementation of the oscillator to the classes that implement this interface. This way, the module controller can work with several types of oscillators without changing the controller code, just switching the *Oscillator* class used.

The calculation of the joint position from the elapsed time is performed on the function *calculatePos()*, which has to be implemented in the classes that follow the *Oscillator* interface.

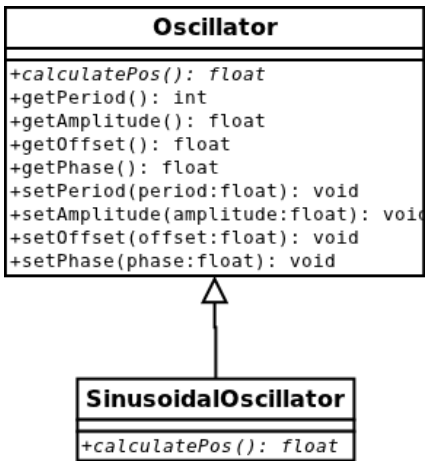


Figure 3.8: Oscillator class diagram

3.3.8 Class SinusoidalOscillator

The only type of *Oscillator* implemented in this project is the *SinusoidalOscillator*, which models the joint position values as an oscillating sinewave.

The *calculatePos()* method calculates the position following a sinusoidal function, characterized by the amplitude, offset, phase and period stored on the *Oscillator* base class. Using the elapsed time, the joint position is calculated the following way:

$$\varphi(t) = A_i \cdot \sin\left(\frac{2\pi}{T} \cdot t + \Phi_i\right) + O_i \tag{3.1}$$

Sinusoidal oscillators are explained in more detail on section 2.2.3.

3.3.9 Class Connector

In order to model the different connectors that allow the modules to be attached to other modules, the *Connector* class is used.

The *Connector* class is a simple data container that has two buffers, one for storing the incoming hormones prior to their processing, and other for placing the outgoing hormones before they are actually sent to the remote connector attached to this connector.

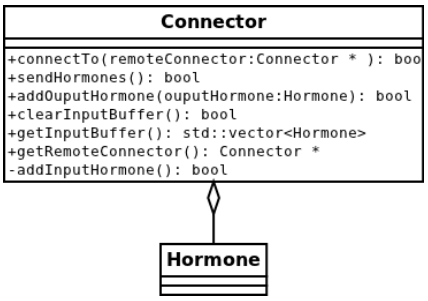


Figure 3.9: Connector class diagram

This class also contains a reference to the remote *Connector* attached to this connector, in order to be able to send the hormones from the ouput buffer. This reference is set with the *connectTo()* method.

For sending out the hormones in the output buffer to the remote module input buffer, the method *sendHormones()* is used, which takes the hormones stored in the ouput buffer and puts them in the input buffer of the remote module using the reference to that module and the *addInputHormone()* function.

The *localOrientation* attribute contains the relative orientation between the local and remote modules calculated by hand. This attribute is no longer in use by the controller, which currently calculates it from the orientation of both modules, dynamically.

3.3.10 Class Hormone

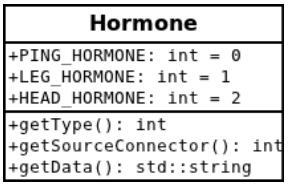


Figure 3.10: Hormone class diagram

The *Hormone* class is the base for the hormone communication protocol, one of the key aspects of this work. The class is a container for the info that is needed to be transmitted to the other modules.

A *Hormone* has 3 fields: the type of hormone ("Ping", "Leg" or "Head"), encoded as an integer; the connector that sent the hormone, also encoded as integer (0, 1, 2 and 3 for the front, right, back and left connectors, respectively) and a data field to attach extra info required for the modules to discover the IDs.

The contents of the data field depend on the type of hormone. "Ping" hormones store in the data field the local orientation of the module that sends the hormone as a string (i.e. "90 180 270", meaning roll=90º, pitch=180º and yaw=270º). "Leg" hormones do not need any extra info, so their data field is empty, and "Head" hormones store in the data field the ID of the global configuration as discovered by the "head" module (0 for *MultiDof-11-2*, 1 for *MultiDof-7-tripod* or 2 for *MultiDof-9-quad*) and, if needed, an extra integer for leg dissambiguation.

The hormone communication protocol is explained in detail on chapter 6.

3.3.11 Class GaitTable

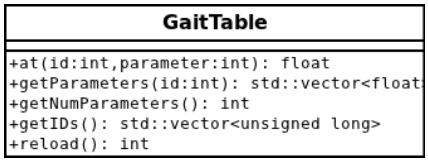


Figure 3.11: GaitTable class diagram

The class *GaitTable* stores the oscillator parameters relating them to a ID. The data is stored as a table, with the different parameters for each ID stored as rows of the table. An example of the internal representation of a gait table for the main oscillator parameters is shown in table 3.1.

ID	Amplitude	Offset	Phase
83506	60	0	0
78896	60	0	120

Table 3.1: Example of the internal contents of a gait table for a 2 module configuration

The *GaitTable* can receive queries for a given ID, and then it returns all the parameters corresponding to that ID. The data is stored and read using text files, in a format that is compatible with the Matlab/Octave text file format.

3.3.12 Class Orientation

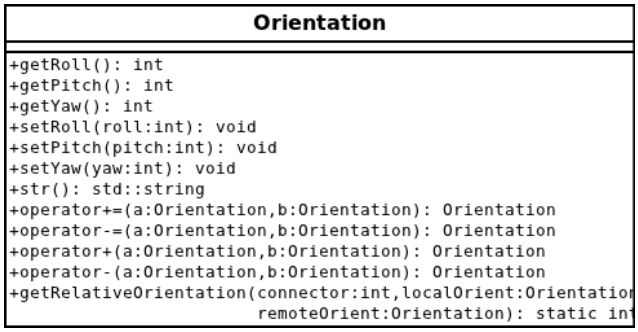
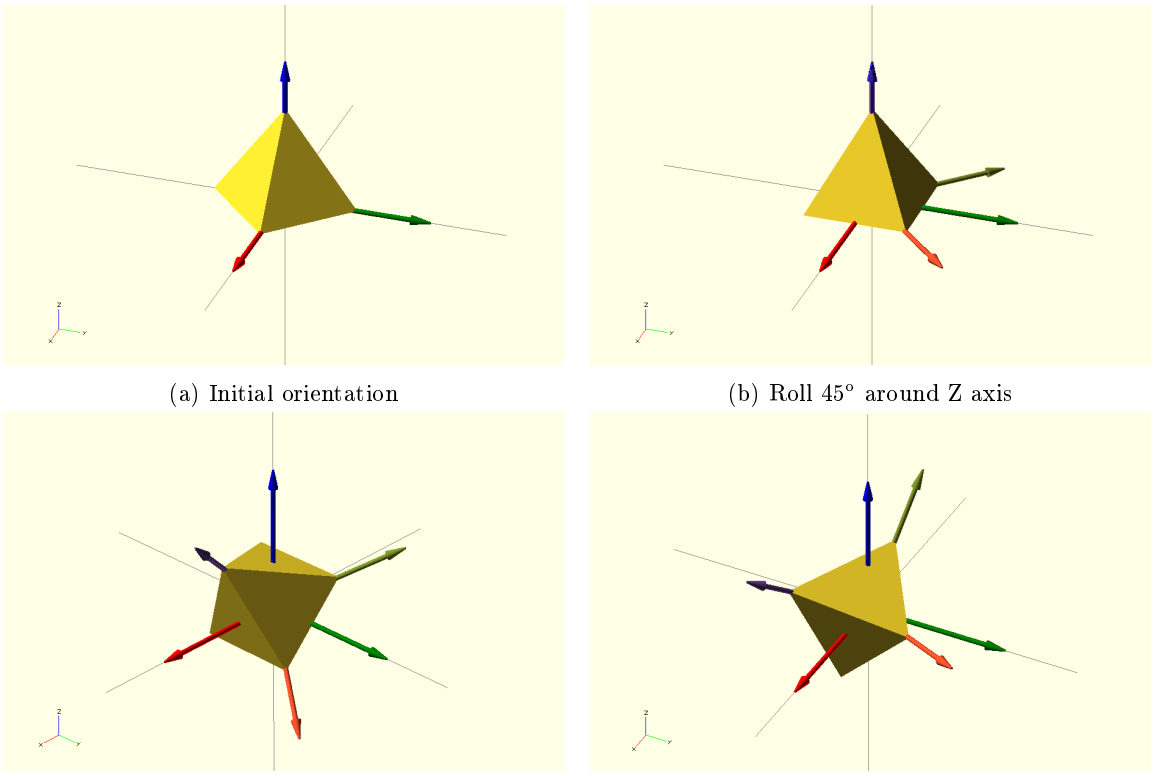


Figure 3.12: Orientation class diagram

For calculating the ID based on the position of the module inside the modular robot, a module needs to know the relative orientation between the attached connectors. To work with those Orientations, the class *Orientation* was implemented. *Orientation* is supposed to be measured by the module with a Inertial Measuremente Unit (IMU) but, as the modules do not have a IMU because of hardware limitations, the IMU is currently simulated by reading the initial values that the IMU would return from the robot configuration file.

For expressing the orientation, we are using Tait-Bryan angles, a representation largely used in aeronautics, and the one that the IMU returns by default. This representation, similar to the Euler Angles, expresses the orientation of an object with three angles: roll, pitch and yaw. These angles represent rotations around the three main axes of a fixed reference frame, roll corresponding to the rotation around the Z axis; pitch being a rotation about the Y axis and finally yaw, a rotation about the X axis. Notice that, unlike Euler Angles, these rotations are performed around a reference frame that is fixed in the world, not in the object, and therefore it does not rotate with the object.

Inside the *Orientation* class, angles are bound to the interval [0, 360), and the class counts with several functions to perform simple arithmetic operations with them, as well as to convert them to a string for storing the *Orientation*



(a) Initial orientation

(b) Roll 45° around Z axis

(c) Pitch 45° around Y axis

(d) Yaw 45° around X axis

Figure 3.13: Steps to obtain a pyramid with a orientation of (45°, 45°, 45°) expressed in Tait-Brian angles RPY

on the data field of the *Hormone* class. For calculating the relative rotation between two modules, the function *getRelativeOrientation* can be used, passing it the ID of the connector that is used as reference (0, 1, 2 and 3 for front, right, back and left, respectively) as well as the local and remote module orientations.

The relative rotation between two modules is expressed as an integer value from 0 to 3 that represents the number of 90° steps we have to rotate the local module around the axis of the local reference system that is in the same direction as the normal vector of the local connector surface, such as the Z axis of the local reference system of both modules coincide. This is explained in more detail, with graphical examples in section 5.1.4.

The algorithm used by *getRelativeOrientation()* to calculate the relative rotation value is the following:

1. Roll, pitch and yaw angles are equivalent to Euler XYZ angles, so we use this property to calculate the transformation matrices for both reference systems (the one from the local module and the one from the remote module) by multiplying the rotation matrices around X (yaw), Y (pitch) and Z (roll).

$$^{0}H_A = Rot_X(\gamma_A) * Rot_Y(\beta_A) * Rot_Z(\alpha_A)$$

$$^{0}H_B = Rot_X(\gamma_B) * Rot_Y(\beta_B) * Rot_Z(\alpha_B)$$

Were $^{0}H_x$ is the homogeneous transformation matrix to go from the absolute reference system to x, A is the local module reference system, B is the remote module reference system and α, β and γ are roll, pitch and yaw angles, respectively. In a matricial form, the previous equations become:

$$^{0}H_A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\gamma_A) & -\sin(\gamma_A) \\ 0 & \sin(\gamma_A) & \cos(\gamma_A) \end{bmatrix} \begin{bmatrix} \cos(\beta_A) & 0 & \sin(\beta_A) \\ 0 & 1 & 0 \\ -\sin(\beta_A) & 0 & \cos(\beta_A) \end{bmatrix} \begin{bmatrix} \cos(\alpha_A) & -\sin(\alpha_A) & 0 \\ \sin(\alpha_A) & \cos(\alpha_A) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$ {}^{0}H_B = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\gamma_B) & -\sin(\gamma_B) \\ 0 & \sin(\gamma_B) & \cos(\gamma_B) \end{bmatrix} \begin{bmatrix} \cos(\beta_B) & 0 & \sin(\beta_B) \\ 0 & 1 & 0 \\ -\sin(\beta_B) & 0 & \cos(\beta_B) \end{bmatrix} \begin{bmatrix} \cos(\alpha_B) & -\sin(\alpha_B) & 0 \\ \sin(\alpha_B) & \cos(\alpha_B) & 0 \\ 0 & 0 & 1 \end{bmatrix} $$

2. We undo the rotation of the local module reference system to make it coincide with the absolute reference frame, and apply the same inverse transformation to the remote module reference system, obtaining the homogeneous tranformation matrix to go from A (local module) to B (remote module):

$$ {}^{A}H_B = {}^{A}H_0 \cdot {}^{0}H_B = ({}^{A}H_0)^{-1} \cdot {}^{0}H_B $$

3. We find the Z vector of the remote reference system ($\vec{k'}$) by multiplying the unit Z vector of the absolute reference system ($\vec{k}$) by the homogeneous tranformation matrix.

$$ \vec{k'} = {}^{A}H_B \, \vec{k} = {}^{A}H_B \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} $$

4. Iteratively, the vector $\vec{k}$ is rotated in steps of 90º until it matches with the vector $\vec{k'}$ calculated in the previous step. The axis of rotation depends on which is the connector of the local module being evaluated, for the front and back connectors, the Y axis is used, whereas for the left and right connectors, it is the X axis. The number of steps required to match them is the relative rotation value returned by the function.

3.4 Applications

Apart from the main framework, the Hormodular software has several applications to perform the optimization of the locomotion parameters and their evaluation either on the simulated robot, or the real robot via serial port. The source code for these applications can be found in the "src/apps" folder inside the project main directory and, once compiled, these applications can be found in the "bin" folder.

3.4.1 evolve-gaits

The program "evolve-gaits" is used for finding the best oscillator parameters in order to achieve an optimal gait on a given modular robot configuration. This program uses the Evolutionary Computing Framework (ECF) library to perform the optimization of those parameters using the Differential Evolution algorithm (Described on section 5.3.2). For the evaluation of the gait generated by the oscillator parameters of an individual of the Differential Evolution population we implemented a very simple controller that just performs the joint position update according to the joint values provided by the sinusoidal oscillators with the parameters to be evaluated.

The usage of the program is not complex, as it only takes an argument for its execution, the evolution configuration file. The configuration file is a xml file containing a set of tags defined by the authors of the ECF library in order to configure the optimization algorithm. Some of the parameters that can be configured with this file are: the type of optimization algorithm and its parameters, the type of genotype to be used, the population size and the number of iterations, among others. ECF also allows the users of the library to define custom tags to configure other aspects of the evolution, such as the path of the robot configuration file, the evaluation time of the robot gait, or the limits of the sinusoidal oscillator parameters.

The program will read this parameters and it will execute the optimization algorithm. A log file is configured to be automatically created to track the progress of the algorithm, and a milestone file is generated with data that can be used to resume the execution of the evolution in case the program crashes or is interrupted. This milestone file also contains the genotype of the best individual, from we will extract the oscillator parameters for the best gait.

These parameters are encoded in the genotype with normalized values in the interval $[-1, 1]$ due to limitations of the ECF that do not allow to set different value limits for different parameters of a single genotype. To obtain the

non-normalized values without doing operations by hand, and set them on a gait table, a Octave/Matlab simple script was developed, and can be found in the "Utils" folder that will generate the required data from the normalized values returned by the evolution program.

3.4.2 evaluate-gaits-sim

Once the best parameters have been found, we would want to evaluate those parameters, as well as the hormone communication protocol, in order to check if a suitable gait was found, and if the hormone controller performs as desired. In order to test it on the simulated robot, the "evaluate-gaits-sim" application can be used.

This program just creates a *ModularRobot* instance with a simulated *ModularRobotInterface* and configures it with the parameters specified by the user. Then, it runs the simulation for the specified run time and reports the results of the evaluation to the user. The resulting gaits can be observed on the simulation viwer offered by OpenRAVE.

The usage of this program is simple. It requires the user to call the program specifying the xml robot configuration file and the time the robot will be simulated (simulation time), with an optional parameter being the simulation step time, which is 0.25 ms by default. At the end of the evaluation, the program shows the real time that it took to simulate the robot during the run time specified by the user, as well as the distance travelled by the robot.

3.4.3 evaluate-gaits-serial

Since the final objective is to control a real modular robot with the controller, it is very useful to test the gaits and the hormone controller on the real modular robot. For that purpose the "evaluate-gaits-serial" program exists.

This program creates a *ModularRobot* with a *SerialModularRobotInterface*, connecting with the robot through the serial port and sending it the commands required to drive the servos to the desired positions. The modular robot, in the current state of development acts as a "dummy" robot: the distributed controller for each of the modules runs on the computer, and it sends to the robot the position of the different joints, which allows the robot to move, and allow us to test if the gaits are effective and optimal in a real-world environment.

The parameters for running the "evaluate-gaits-serial" program are exactly the same as the ones for the simulated robot evaluation, but in this case the step time corresponds to the rate of update of the robot joints, which has been increased to 2 ms by default due to bandwidth limitations of the serial port.

3.4.4 Utils

Apart from the main programs, the repository includes a series of scripts that solve in a fast way some of the repetitive and dull tasks that appear during the development process.

The first one is a Python script called *IDcalculator.py*, that takes a string containing the connection description in a human-readable way as argument and translates it to the actual ID number. An example of its usage would be:

```
$ python IDcalculator.py "((L,90), X, (R, 270), X)"
```

In which each of the four elements of the top level 4-tuple is the information of the remote connector connected to the front, left, right and back connector, respectively, with 'X' denoting that there is nothing connected to that connector. The elements of the inner tuples are the remote connector attached to the local connector, and coded with the initial of that connector (i.e. L for left or R for right) and the relative orientation between connectors. In the example above, that ID would correspond to a module with the left connector of another module attached to the front connector at an relative angle of 90º, and the right connector of other module attached to the back connector at a relative angle of 270º which corresponds to the ID 82644.

The second one is a Octave/Matlab script called *vectorToTable.m* that converts a vector containing the genotype resultant from the optimization process to a gait table. This genotype contains the optimized oscillator parameters normalized in the interval $[-1, 1]$. The parameter limits are specified when calling the function defined in the script, returning a table with the parameters already scaled back to the non-normalized form. If we want to obtain the gait table for a *MultiDof-9-tripod* configuration, optimized with $A_{max} = 80^o$, $O_{max} = 45^o$, $\phi_{max} = 360^o$ and $f_{max} = 1.5Hz$, the function would be called as follows:

```
$ octave

octave:1> v = [ 0.434181, 0.845216, 0.470191, -0.846046, 0.790331, -0.264226, 0.857843, -0.527548, -0.67852,
0.917075, -0.306737, 0.164061, 0.190244, -0.522919, -0.411536, -0.44878, -0.689907, 0.402322, 0.159478,
-0.348137, 0.579257, 0.0460117];

octave:2> vectorToTable( v, 7, 80, 45, 360, 1.5, 1);
```

This will generate two files: the gait table of the main oscillator parameters (A, O, ϕ), called *Tx.txt* and the gait table containing the frequency, named *fx.txt*, where 'x' is the configuration ID in both cases.

3.5 Compiling & running Hormodular

The source code of the *Hormodular* framework is releashed under a open source GPLv3 license, allowing any researcher interested in modular robotics to use this code, study it, improve it and publish their modified version, with a compulsory attribution to the original author. This way the results presented in this work can be evaluated and tested by other researches, reproducing the experiments with the same software to test their validity.

Following the open source approach, the project has been developed using open source software under a GNU/Linux system. Therefore, the code is only tested on the GNU/Linux platform and there are no guarantees that it works on other platforms suchs as Windows or Mac.

This section explains how to install the required dependencies required by *Hormodular*, as well as how to download and compile the source code, and how to run it.

3.5.1 Installing dependencies

Installing CMake

CMake is an open source cross-platform program that automates the compilation and installation of software by generating the corresponding makefiles and environment to be used with the compiler desired by the user. CMake can be found in the repositories of many GNU/Linux distributions, or downloaded from `http://www.cmake.org/cmake/resources/software.html` , either already compiled or as source code.

To be installed from the GNU/Linux terminal on a Ubuntu system, the following command can be used:

```
$ sudo apt-get install cmake
```

Installing OpenRAVE

OpenRAVE is the open source simulation library chosen to simulate the modular robot and evaluate its locomotion gaits for the different configurations.

Detailed instructions for installing OpenRAVE can be found on their website: `http://openrave.org/docs/latest_stable/install/`. On a GNU/Linux, Ubuntu-based system, the commands for installing OpenRAVE from the repository are:

```
$ sudo add-apt-repository ppa:openrave/release
$ sudo apt-get update
$ sudo apt-get install openrave
```

If these commands do not work, or if it is preferred, OpenRAVE can be installed from the sources, following the instructions that can be found in: `http://openrave.org/docs/latest_stable/coreapihtml/installation.html`

Installing OpenMR plugin for OpenRAVE

The OpenMR plugin for OpenRAVE adds the servocontroller to the controllers available by default on OpenRAVE. As well as the 3D models for simulating REPY and REPY-2.0 based modular robots. It was originally developed by Juan Gonzalez-Gomez but his version is no longer maintained, so we recommend to download it from our repository: `https://github.com/David-Estevez/openmr`.

The code is ready to be compiled using CMake. In a GNU/Linux terminal, the commands to install it would be (it is assumed that the user is already in the openMR project folder):

```
$ mkdir build
$ cd build
$ cmake ..
$ make
$ sudo make install
```

Installing the Evolutionary Computing Framework

The Evolutionary Computing Framework provides the optimization algorithms required to obtain the sinusoidal oscillator parameters necessary to achieve an optimal locomotion gait. All the information related to the ECF can be found on its website: `http://gp.zemris.fer.hr/ecf/`.

In order to install the ECF, it is required to download the source code, whose link can be found in their website, under the "Download" section. Instructions for building the ECF can be found in: `http://gp.zemris.fer.hr/ecf/install.html`, but for a GNU/Linux system they can be summarized on running the following commands on the source code directory:

```
$ ./configure
$ make
$ sudo make install
```

In case this procedure fails for any reason, a script (*ecf_install.sh*) is provided for a semi-automatic installation.

Installing TinyXML2

TinyXML2 is a lightweight library required for parsing the robot configuration XML files. The source code can be downloaded from their git repository: `https://github.com/leethomason/tinyxml2` either as a zip file or, if git is installed, with the following command:

```
$ git clone https://github.com/leethomason/tinyxml2
```

Once the source code is downloaded, it can be build and installed using CMake:

```
$ mkdir build
$ cd build
$ cmake ..
$ make
$ sudo make install
```

Installing Eigen

Eigen is an open source library for making linear algebra operations. It is used to calculate the relative rotation between two modules from their Tait-Bryan angles. Eigen is a C++ template library, so no code has to be compiled for its installation, copying the headers to the system libraries folder is enough.

If Eigen is available on the repositories, it can be installed with the following command:

```
$ sudo apt-get install libeigen3-dev
```

Otherwise, the source code can be downloaded from their webpage: `http://eigen.tuxfamily.org/index.php?title=Main_Page` , and installed using CMake:

```
$ mkdir build
$ cd build
$ cmake ..
$ sudo make install
```

3.5.2 Building Hormodular

After all the different depencies have been installed, the source code of Hormodular can be downloaded from our repository: `https://github.com/David-Estevez/hormodular` , either as a zip file or with the following command, if git is installed:

```
$ git clone https://github.com/David-Estevez/hormodular
```

Once the code is downloaded, the procedure to build the project using CMake is simple:

```
$ mkdir build
$ cd build
$ cmake ..
$ make
$ sudo make install
```

3.5.3 Running Hormodular

Once compiled, the different applications should appear on the "bin" folder, ready to be executed. The purpose of each of the applications is explained in section 3.4.

Chapter 4

Hardware

This chapter will present the hardware modules and will explain the tools used to design them, as well as the existing work in which our design is based. The modules are composed of a mechanical part, a electronic control board, and other elements such as the servo or the battery, that will be explained on their corresponding sections on this chapter.

4.1 Module

In this section we will talk about the design of the mechanical part of the module, the two-piece plastic structure that holds the servo and transmits its movement to the other modules, allowing the modular robot to move. We will start discussing the modelling paradigm used to design the model, then we will introduce the previous existing modules from which we took inspiration to design our module, and finally we will present our design, with the corresponding assembling instructions.

4.1.1 Software used

The approach tipically followed for the design and modelling of 3D object is usually a interactive one. Using a CAD software such as SolidWorks the engineer designs the object by applying a series of transformations to a basic shape until the final object is obtained. Other approach, not so frequently employed, is generative design. With this approach, the object is generated by a set of rules or an algorithm, like a computer program. This way, the design can be specified as a function of a set of variables or parameters that can be later modified, modifying the whole design and varying its dimensions, or generating different parts or variations of the same parts.

Using this generative design paradigm, we have designed a module that is parametric, and whose 3D model adapts automatically to the servo and control board, allowing the design to be more accessible, as it does not depend on a single servo model and reusable, as it can adapt to the user needs.

OpenSCAD

OpenSCAD is an open source software for creating solid 3D CAD objects [6]. Unlike most 3D object editors, it is not interactive, but based on script files that describe the geometry from operations with basic primitives.

It provides two main modelling techniques: Constructive Solid Geometry (CSG), where boolean operators are applied to primitive 3D objects to create complex 3D objects; and extrusion of 2D outlines, where a 2D shape is extruded to create a 3D object. This techniques can be mixed, using 3D objects generated by extrusion of 2D outlines as operands of boolean operators to generate even more complex geometries.

CSG modelling has several advantages over the use of polygonal meshes. With CSG modelling the user can model very complex geometries with just a reduced set of simple primitive objects, simplifying the modelling process. If GSG

is procedural or parametric, a complicated design can be changed easily by changing the parameters, allowing designs to be adapted to the new requirements automatically. When the primitive objects used in CSG modelling are "solid" or water-tight it is ensured that the resulting object is also water-tight, which is very important for manufacturing the object. As opposed, when creating geometries using polygonal meshes, consistency checks must be performed to ensure that the mesh represents a valid solid object. Finally, a point can be easily classified as being inside or outside the resulting shape by testing it against all the underlying primitives, and evaluating the boolean expression that generated the shape with the classification values, which is very useful in applications such as collision detection.

The 3D primitives available for CSG in OpenSCAD are three: cube, cylinder and sphere. Cones can be generated from cylinders by specifying a top/bottom radius of 0 units when creating it. These primitive shapes can be translated, rotated or scaled any number of times, and then boolean operators can be applied to form complex shapes. OpenSCAD has boolean operators for performing unions, differences and intersections of objects. Figure 4.1 shows an example of a complex object assembled from simple combinations of primitive objects and boolean operations.

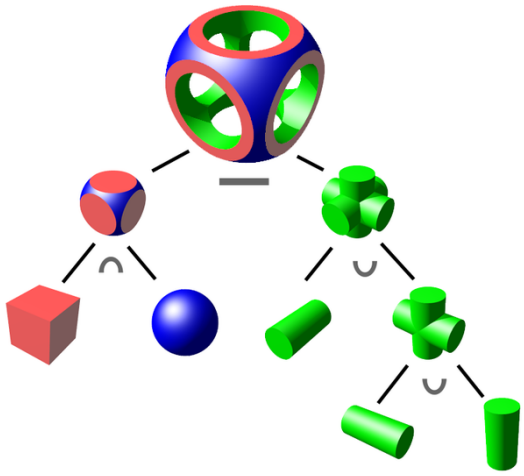


Figure 4.1: Example of a complex shape made from simple primitive objects and boolean operators.

OpenSCAD also counts with some 2D primitives for creating objects through 2D shape extrusion, such as squares, circles and polygons, defined point by point. It also can import CAD files in DXF format for extruding them, or export 2D shapes in DXF format to use them in other CAD programs. The 2D shapes can also be combined using boolean operators, and the extruded to form 3D objects, either with a linear extrusion or with a rotate extrusion, that creates solids of revolution.

Object-Oriented Mechanics Library (OOML)

The Object-Oriented Mechanics Library [52, 53] is an open source C++ library to model 3D objects using C++, in a similar way as the objects are described with OpenSCAD. It was developed by Juan Gonzalez-Gomez and Alberto Valero-Gomez as a way to extend the capabilities of OpenSCAD with the features available in more advanced programming languages such as objects, inheritance or operator overloading, as well as the existing libraries for those languages to, for example, perform mathematical operations.

OOML counts with the same primitive 3D objects of OpenSCAD (cube, cylinder and sphere), plus some additional components such as toroids, prisms, rounded tablets, rounded cubes, rounded cylinders or text strings. It also offers the same 2D primitives, and boolean operations, that can be performed with the classic C++ operators: "+" for union, "−" for difference and "∗" for intersection.

Apart from the basic primitive objects, OOML has also a part library with more complex, basic objects with a mechanical meaning, that can be combined to form even more complex mechanisms, such as robots. These objects are

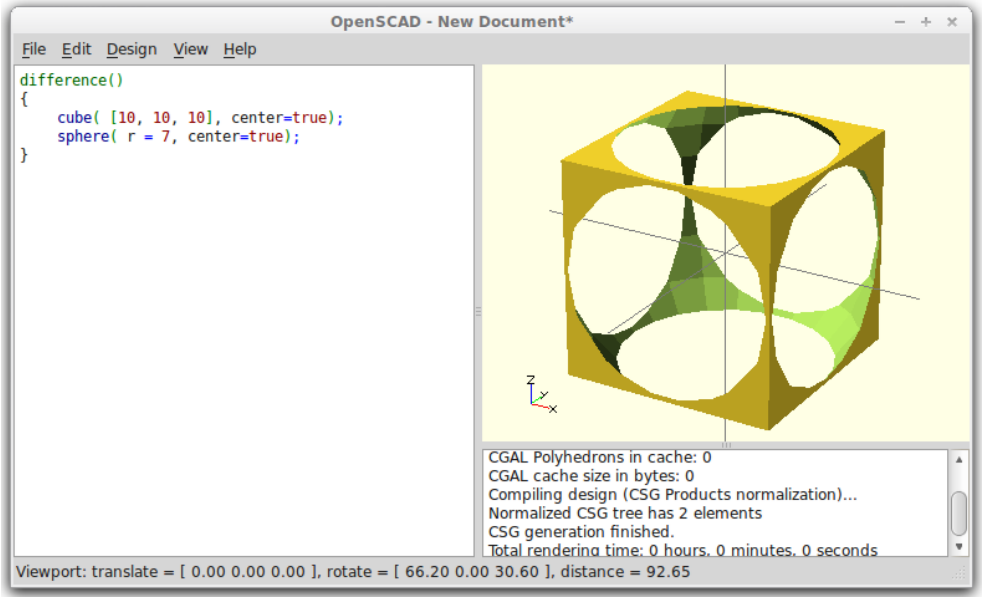


Figure 4.2: OpenSCAD screenshot with a simple object being created.

parametric, so they can be reused across designs, and derivative objects can be made based on these ones by using C++ inheritance. This library contains wheels, battery holders, batteries, servos, sensors, etc.

From the compoment object, OOML automatically generates OpenSCAD code that can be used to render and compile the object in the OpenSCAD development environment. For the code generation OOML has a class, called *Writer*, that is in charge of converting the description of a 3D object into OpenSCAD code.

4.1.2 Previous work

On the state of the art, we introduced several platforms for developing modular robotics. For developing our module we reviewed all of them, and we decided not to reinvent the wheel starting a new design from scratch, so we selected an existing open platform, the Y1 module and its derivatives.

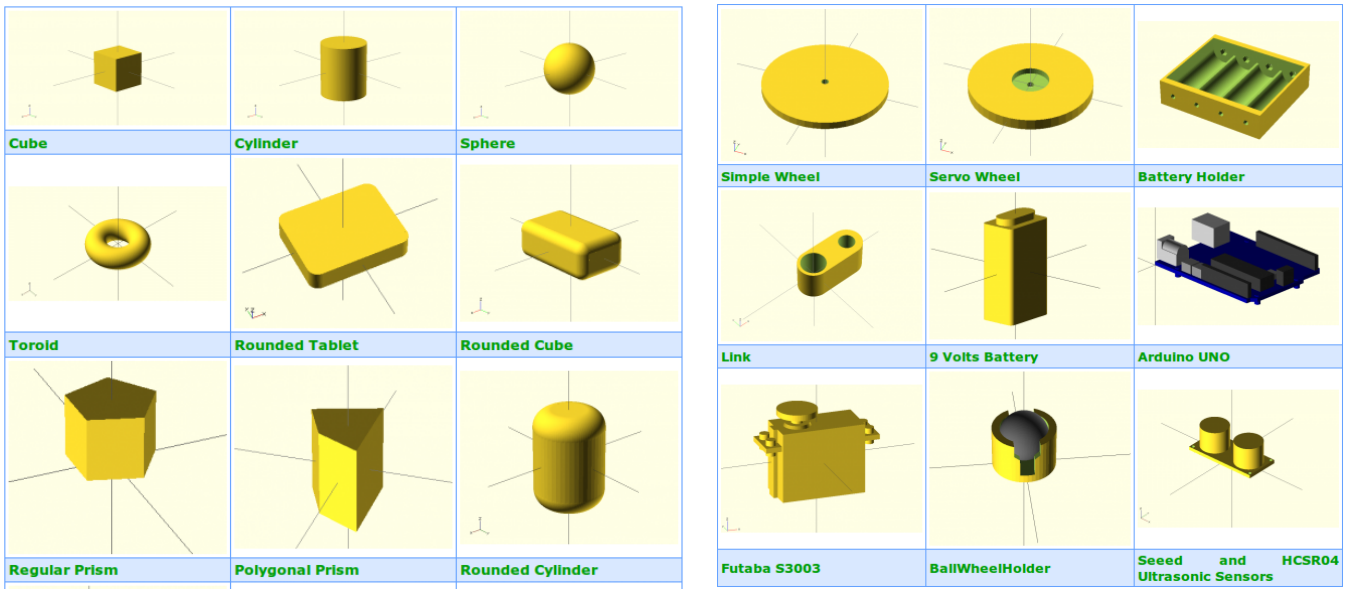
Y1 module

The Y1 module [18] was developed by Juan Gonzalez-Gomez for his PhD thesis, as a open and cheap platform to study locomotion of modular robots. The Y1 modules are based on the G1 modules designed by Mark Yim [56] for the PolyBot robot (the letter 'Y' in Y1 stands for "Yim" in his honor).

Y1 modules are composed of a cheap hobby servo and a two-part housing made from several pieces of laset cut PVC glued together. These modules can be connected in a linear configuration, either pitch-pitch or pitch-yaw, allowing the resulting robots to move in a plane. As the main objective of Gonzalez-Gomez's thesis was to study only the locomotion of modular robots, the resulting design is very simple, and the connections between modules are achieved with screws. As the module does not have any active connector, reconfiguration has to be done by hand. For the same reasons, the control electronics and the power source were supplied off-board.

The main reasons for basing the design of our module mainly on the Y1 module are the following:

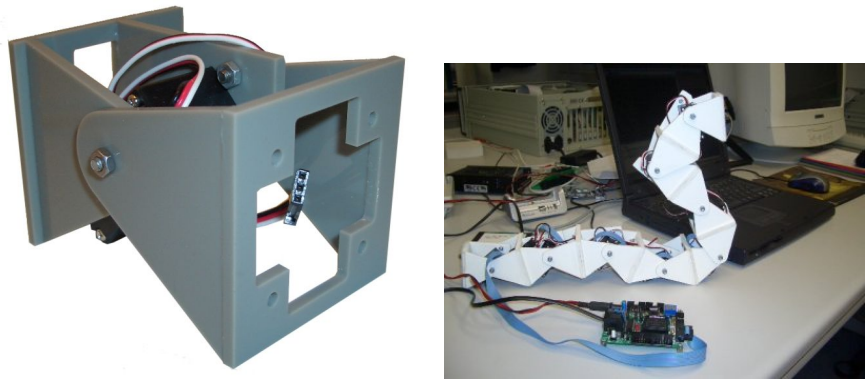
- **Open Source Hardware.** The Y1 module is open source hardware, which means that the plans, the files for manufacturing and the assembly instructions are publicly available online for anyone to access them, study them, and improve them, offering a good starting point for a new design.



(a) Primitive objects (b) Parts

Figure 4.3: OOML's primitive objects and parts

- **Low-cost.** Even though the robots made from Y1 modules can only work tethered and are no self-reconfigurable, their extremely low cost compared with the previously mentioned platforms and the public availability of its sources make the Y1 modules a perfect choice for studying locomotion gaits for modular robots in labs with a reduced budget, or for introducing undergraduate students, or even kids [21], to modular robotics.
- **Availability of materials and easiness of assembly.** The Y1 modules are made of PVC plastic, which is very easy to find in any hardware store. The parts that compose the module housing can be cut from that plastic either by hand or using a laset cutting machine, and later glued together to form the housing. This housing is assembled together using screws, making the whole assembly process very easy to perform.



(a) Y1 (b) Cube Revolutions (Y1 snake)

Figure 4.4: Module Y1

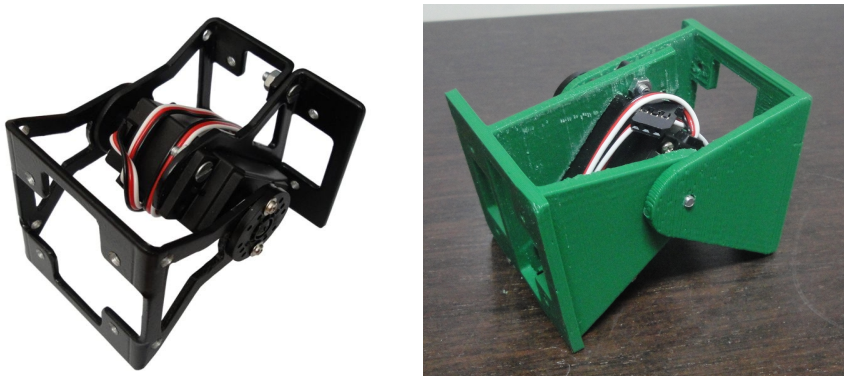
Y1 module derivatives: MY1 and REPY1

Based on the Y1 design, Gonzalez-Gomez developed other modules: the MY1 and the REPY-1. The MY1[22] is a metal version of the Y1 module, manufactured from 2mm thick aluminium with a similar shape to the Y1, and compatible with it. A battery holder and a control board were designed for this module, therefore allowing untethered operation of the robots constructed with this module.

The advantages of the MY1 module are a stronger frame, and the incorporation of the control board and power in the robot structure, requiring only either a serial connection with the computer to be controlled, or having the movements loaded on the controller board. This advantages, however, imply a higher cost and a more complicated manufacturing process, increasing the price of the modules.

The REPY-1[17] module (REPlicable Y1) is a version of the Y1 designed to be printed in a low cost 3D printer, like a RepRap[31]. This simplifies the assembly of the module, as it reduces the number of parts to 2, which can be produced at a low price. If printed on a RepRap 3D printer such as a Prusa Medel 3 model, that costs around 300€, the filament material to print both module halves costs no more than 1-2€.

Apart from the reduced cost and even further simplification of the assembly process, the REPY-1 module has other advantages with respect to the Y1 module. Since the 3D printing process does not imply high fixed manufacturing costs (such as material handling costs, set-up costs, mainteinance costs, etc) there is no need to produce a large batch of modules. Instead, modules can be printed one by one as they are needed. This also allows a faster and cheaper prototyping and improvement process, as modules can be produced, assembled and tested in a short period of time at a low price. Improvements can be done to them, and a new batch of the next iteration of prototypes can be manufactured in a short time and at a low cost, achieving a evolution in the design that previously was only possible to be done in software products.



(a) MY1 (b) REPY-1

Figure 4.5: Module Y1 derivatives: MY1 and REPY-1

The MY1 and REPY-1 modules are also open source, and the files to manufacture them are available in Gonzalez-Gomez's website, with assembly and usage instructions. Having the files publicly available allowed us to download them and print a small batch of 4 REPY-1 modules in order to test them and evaluate their virtues and flaws, in order to improve the design later. We will discuss those improvements made to the original Y1 design on the next section about our design, the REPY-2 module.

4.1.3 REPY-2

Several REPY-1 modules were printed and analyzed on different configurations in order to find if they could be used in this project. After testing them, we realized that the design was a good starting point, but it needed to be improved in order to be used for this thesis.

Based on our analysis and tests of the printed version of the Y1 module, the REPY-1 module, we observed the following problems in the module design:

- The design was difficult to print correctly and it was very fragile. The part that holds the hobby servo was connected to the rest of the body of the module by a small section that broke while it was being printed, and one of the ears of the module was too thin, so it usually broke when assembling both halves together.
- The REPY-1 was designed on OpenSCAD, but it imported the DXF files of the laser-cut Y1 module, extruded them and placed them together, so the module was not parametric. This means that this design only works with the Futaba 3003s hobby servo and the Skycube/Skymega board. This design is therefore not easy to modify or adapt to other hardware different from the original one.
- The module REPY-1, as well as the module Y1 are not symmetric, since they have the hobby servo placed at an angle of approximately 45^o with respect to the base. This results in a more compact module, but allows locomotion of robots only when placed over one of their sides, the opposite one has a straight spine that contains the servo and makes the gait unstable.
- Only 2 connectors are present on the Y1 modules and their derivatives, on the front and the back of the module, so the possible configurations that can be assembled with them are restricted to chain-type, 1D configurations. This makes them only suitable for testing locomotion in apodal robots, such as snake robots.

Other improvements, such as an active connector to enable self-reconfiguration, or a control board and power management system integrated on each of the modules were also desirable, but discarded due to the temporal and economic constraints of this project.

First version: REPY-2.0

We started the design of the module in OpenSCAD, but soon we moved to OOML, since the C++ language allowed us to perform more powerful operations with a clearer syntax. Designing the module with the OOML library instead of directly on OpenSCAD also allowed us to organize the code better, helped by the use of C++ classes.

This first version, called REPY-2.0, solved most of the issues found on the original REPY-1 module. The module was designed not just parametric, but object-oriented. This means that a *BasicServo* class and a *BasicPCB* class exist, containing all the dimensions that define the hobby servo and the control board. Instances of these classes are passed to the constructor of the *REPY_module* class, that uses the stored dimensions to generate automatically a module suitable for being used with that servo and control board. If one wants to build the module with a different model of hobby servo, or with a different control board, he just has to define new servos or PCBs, by inheriting the basic interface from the *BasicServo* and *BasicPCB* classes. If the *REPY_module* constructor is called with these new servos or PCBs, it will automatically generate a module to be used with them.

The module was made symmetrical by placing the hobby servo at a 90^o angle with respect to the base of the module, allowing a smooth movement of the module on either of its opposing sides, solving the instablity problem of the Y1 module when placed over the straight side of the module body.

The overall design was reinforced so that it was easier to print and stronger, being less likely to break it when assembling the module. The thickness of the walls was increased as thin walls were a frequent cause of failure when printing the REPY-1 module and they broke easily. This thicker walls also allowed the horn of the servo to be embedded in the module upper part, eliminating the need of using screws to fix the servo horn to the module.

This module version was printed for the Futaba 3003s servo with a Skymega board and, in order to test that this design could also generate modules to use with other servos and boards, we also print a smaller version of the module to

assemble with the smaller Towerpro SG90 servo, and a small, custom board. Figure 4.6 shows the two kinds of REPY-2.0 modules printed to test the design. It is important that both modules were generated from the same code, only changing the servo and board objects passed to the module constructor.

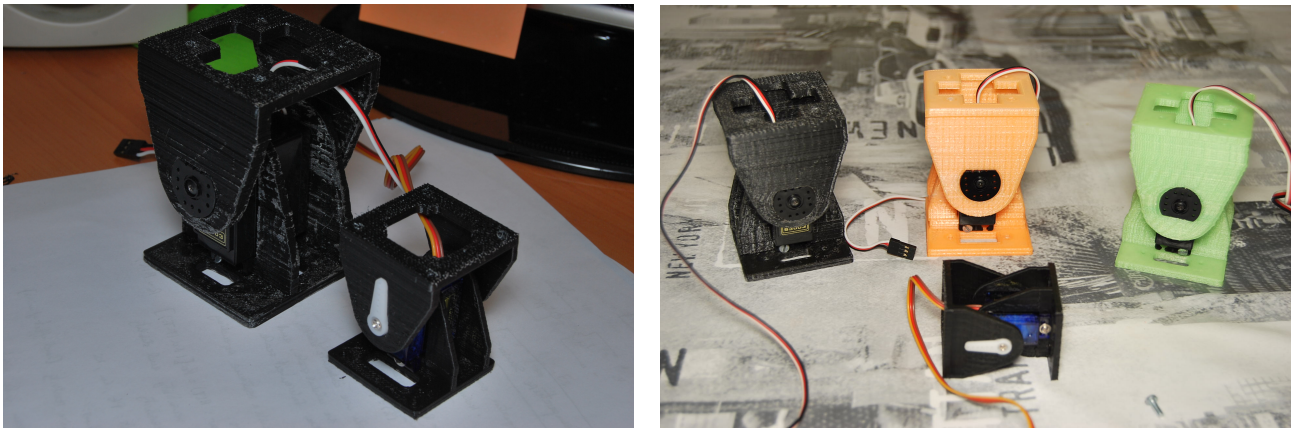
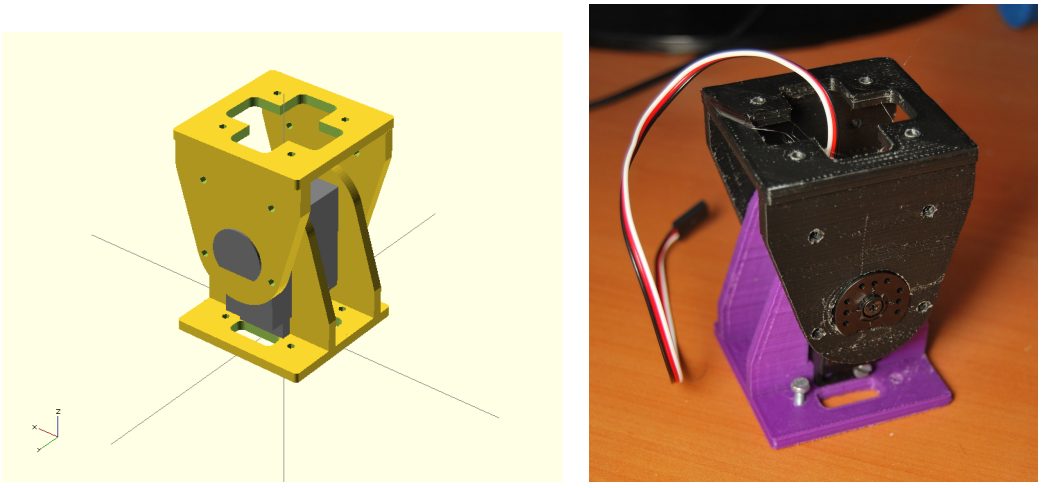


Figure 4.6: Two different REPY-2.0 modules for different servos generated from the same code.

Second version: REPY-2.1

A second version of the REPY-2 modules was developed shortly after testing and validating the first version. This version is mostly equal to the first version, but the tolerances for the different fittings are better adjusted. The last of the required improvements, two side connectors for enabling 2D configurations were also added in this version. These connectors are just 4 holes in the same disposition than the holes in the front and back of the module, that allow the connection with other modules with screws, as the front and back connectors did.

Since both versions are compatible (in both connector and dimensions), and the main difference is the presence of two extra side connectors, both versions were used to assemble the modular robots to test the locomotion gaits, using the second version ones for the places were a 2D connection between modules were used. This way we could use the first version modules already printed to reduce the number of required modules of the second version.



(a) OpenSCAD render of REPY-2.1 module. (b) Printed REPY-2.1 module.

Figure 4.7: REPY-2.1 modules.

4.1.4 REPY-2.1 OOML code structure

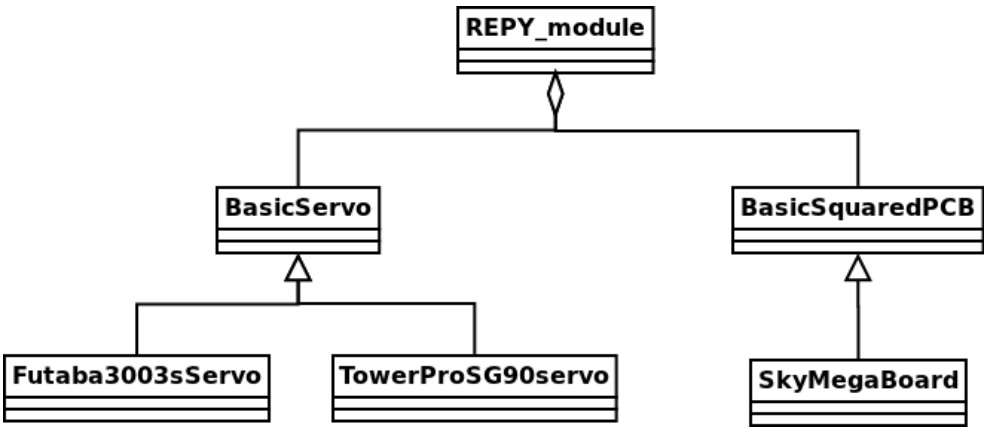


Figure 4.8: Main class diagram

The OOML code for describing the REPY-2 module has a main class, called *REPY_module*, that defines the geometry of the whole module (both upper and lower parts). This class takes as arguments a servo object that follows the *BasicServo* abstract interface and a PCB object that implements the *BasicSquaredPCB* abstract interface. From this objects the *REPY_module* extracts the key dimensions and uses them to calculate its own dimensions and geometry automatically. The main class diagram for the module OOML code is shown in figure 4.8.

This section will explain the main classes used to model the REPY-2 module with the OOML. More detailed information about the code, useful for developers interested in understanding the code and contribute to the project with new servos, boards or improvements can be found documented online on `http://www.dsquaredrobotics.com/wiki/doku.php?id=en:repy-2.0` .

The code of the REPY-2 module is open source, and can be found in the following repository: `https://github.com/David-Estevez/REPY-2.0` .

Class REPY_module

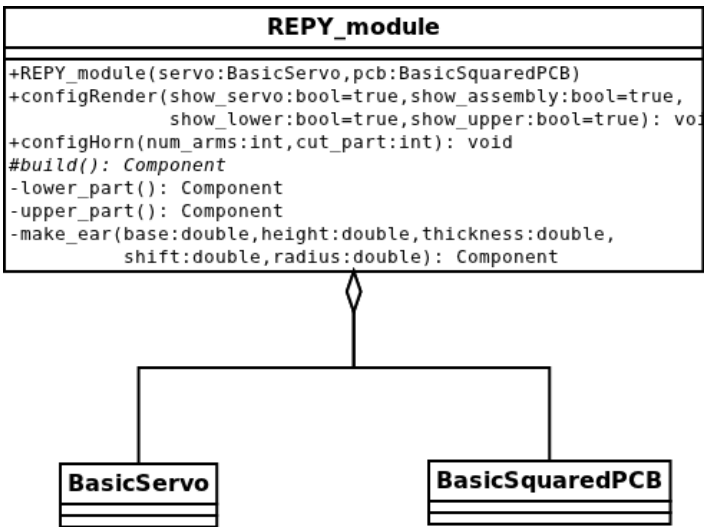


Figure 4.9: Class diagram for REPY_module

The class *REPY_module* generates the geometry of the module. It extracts the required dimensions from a *BasicServo*

and a *BasicSquaredPCB* and calculates its own dimensions from them.

With the function *configRender()* the user can specify if the OpenSCAD code generated from this OOML component will represent the upper part of the module, the lower part of the module or both, apart from other options as if those parts will be shown in an assembly view or in a position ready to be printed as well as if the servo will be shown or no. It is also possible to select the servo horn to be used between the ones available for each servo with the function *configHorn*.

The upper and lower part are defined in different private functions, *lower_part()* and *upper_part()*, that are called by the function *build()*, which generates the whole model depending on the configuration parameters selected. As both parts of the module use a similar shape for their sides, a function *make_ear()* is defined to create them easily.

Class BasicSquaredPCB

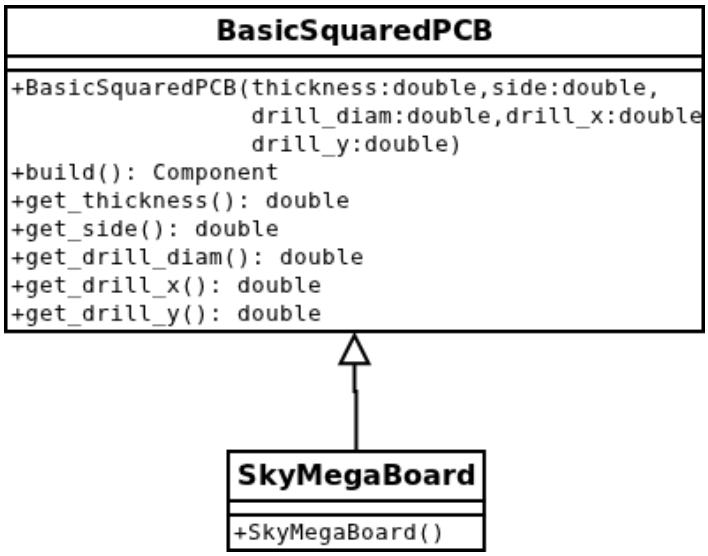


Figure 4.10: Class diagram for BasicSquaredPCB

The class *BasicSquaredPCB* represents a simple squared PCB with four holes for placing screws to attach it to the module. It is defined by the length of the side, the thickness of the PCB, and the location and radius of the drills for the screws.

The SkyMega board dimensions are included in the *SkyMegaBoard*, that is used to create the standard size module. For the small module, a custom board was defined using the constructor of the *BasicSquaredPCB* class.

Class BasicServo

Hobby servos can be created using as a base the *BasicServo* class. The main dimensions of the servo are included in this class, and can be accessed with the getter functions. This way the *REPY_module* class can extract the main dimensions of the servo to calculate its own dimensions. The class diagram of figure 4.11 shows the getter functions available to request the servo dimensions.

Different horns can be defined for a servo, and selected with the *set_horn()* function. This horn is later created on the *build()* method using the *make_horn()* with the rest of the module.

Two different servos were defined, the *Futaba3003sServo* for the standard size module and the *TowerProSG90Servo* for the smaller one. These classes contain the dimensions of the Futaba 3003s servo and the Tower Pro SG90 servo

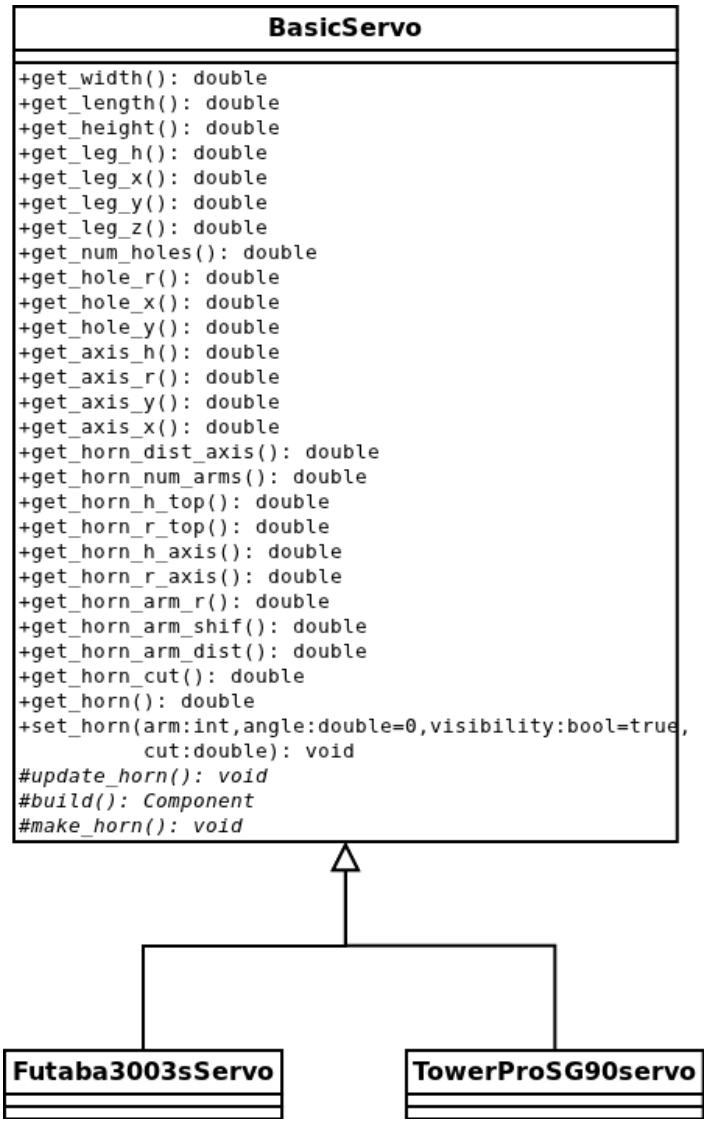


Figure 4.11: Class diagram for BasicServo

respectively, as well as some horns with different number of arms for each one.

4.1.5 Building the REPY-2.1 module

In this section we will explain how to build and assemble a REPY-2.1 module. The files required for manufacturing the two parts that compose the module can be downloaded from the internet, as we are making them publicly available with an open source license.

Generating the 3D files

The first step to build the REPY-2 modules is to obtain the STL files in order to print the parts of the module. These files can be downloaded from the module repository: `https://github.com/David-Estevez/REPY-2.0` , or compiled from the source code.

The requirements for obtaining the files from the sources are the following:

- **CMake.** CMake is used to generate the makefiles used to compile the project C++ code. CMake can be downloaded from `http://www.cmake.org/cmake/resources/software.html`. More detailed instructions can be found on section 3.5.1.
- **OOML.** The Object Oriented Mechanics Library allow us to generate OpenSCAD code from the C++ code, and can be downloaded from `http://iearobotics.com/oomlwiki/doku.php?id=start` . The website counts with very detailed instructions for installing OOML.
- **OpenSCAD.** The OOML only generates OpenSCAD code. For obtaining the STL files, OpenSCAD is required. It can be downloaded from its website: `http://www.openscad.org/` .

Once the dependencies are installed, the code has to be built. The instructions for building the code are:

1. Edit the file "CMakeLists.txt" to include the path to the OOML include directory. For example, in the authors system this path was "/usr/include/ooml".
2. Open a terminal, go to the REPY-2.0 directory and build it using cmake:

```
$ mkdir build
$ cd build
$ cmake ..
$ make
$ make install
```

   Notice that for the installation you won't need to be superuser, as it is installed in a directory called *'bin'* in the *'REPY-2.0 folder'*.

3. Execute the program *REPY-2.0*, that will create the SCAD files and, optionally, the STL files for the REPY-2.0 module.
4. If the STL files were not generated using the *REPY-2.0* program, open the desired SCAD file with OpenSCAD and compile it manually.

Printing the parts

Once the STL files have been downloaded or generated, the next step is printing them on the 3D printer. The concrete steps that are to be followed to print depend on the 3D printer that is going to be used and the software to control it. These low-cost 3D printers used create the object by depositing plastic layer by layer until the object is complete. The plastic used is tipically ABS (Acrylonitrile butadiene styrene) or PLA (Polylactic Acid)

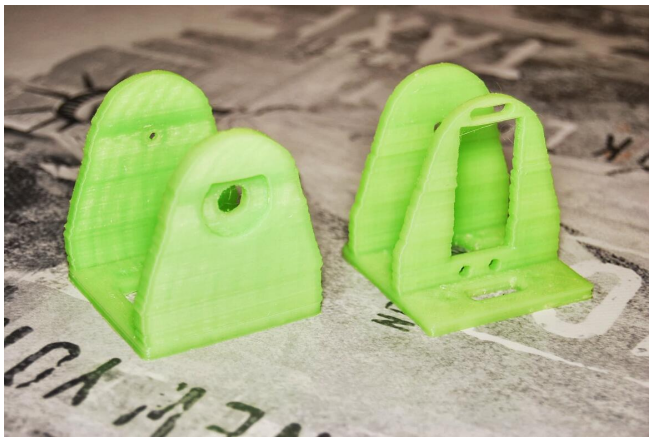
The steps required to print the files involve usually to generate the GCODEs (codes that are control the CNC machine, specifying the velocity and position of its tool, in this case, the plastic extruder required to follow the required path, as well as the amount of plastic to extrude in each position) with a slicer software. The slicer software calculates those toolpaths by slicing the 3D models in several planes, and calculating the trajectories required in each layer to form the 3D object.

For the module to be assembled two parts need to be printed: the lower part and the upper part. As hobby servos usually come with more than one horns with different geometries available, there are more than one upper parts in the repository, each one to be used with a different type of horn. Only one upper part is needed to build the module, that has to be selected according to the horn to be used.

Assembling the module

Each module requires the following materials to be built:

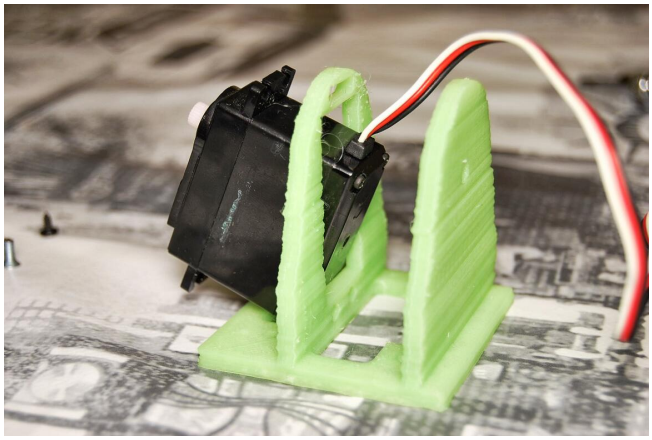
- 1x Upper part of the module (3D printed part)
- 1x Lower part of the module (3D printed part)
- 1x Futaba 3003s servo
- 1x M3x8mm screw
- 4x M3x10mm screws (minimum 2)
- 4x M3 nuts (minimum 2)
- 4x M3 washers (minimum 2)



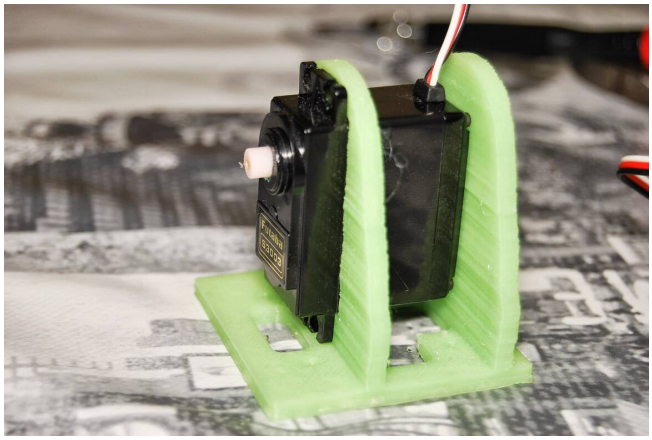
(a) Printed plastic parts needed: upper part and lower part.

(b) Overview of tools and parts needed for assembling the module.

Figure 4.12: Required materials to build the REPY-2.1 module

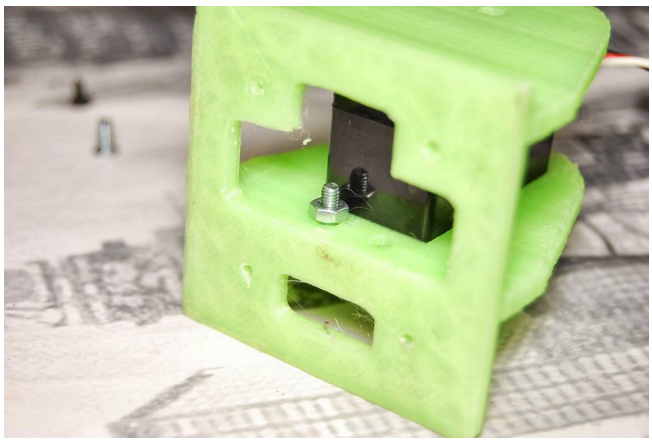


(a) Insert carefully the module in the lower part in the hole prepared to hold the servo.



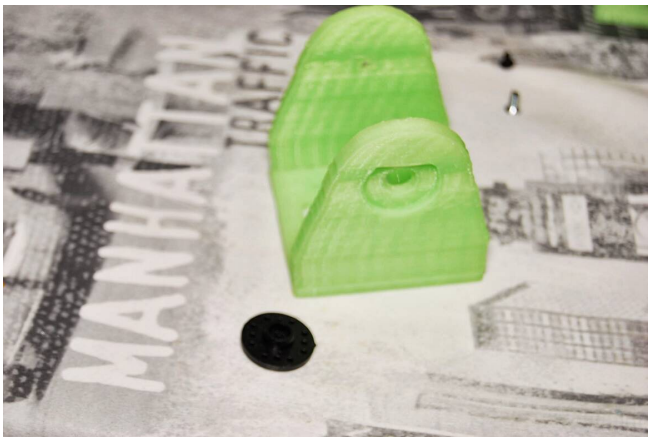
(b) Check that the servo leg holes and the module corresponing holes are aligned.

(c) Fix the servo to the module using the screws, nuts and washers for the top holes.



(d) Fix the servo to the module using the screws, nuts and washers for the bottom holes.

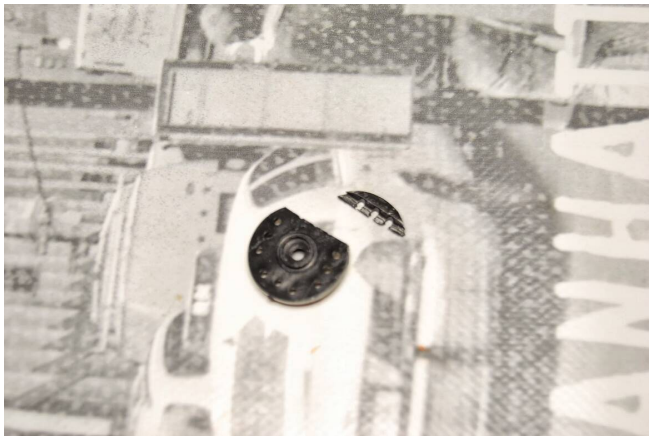
Figure 4.13: Assembly of the lower part



(a) Materials required to assemble the upper part.

(b) If needed, file the hole until the servo horn fits tightly on it.

(c) With the help of some cutting pliers, cut the horn with the shape of the corresponding hole in the printed part.

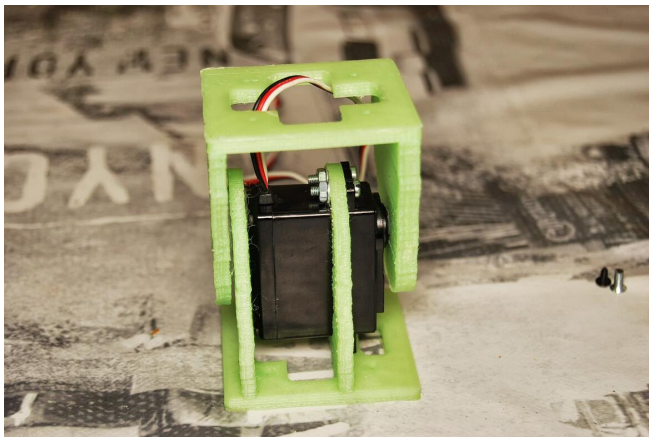


(d) After the cut, the servo horn should look like this.

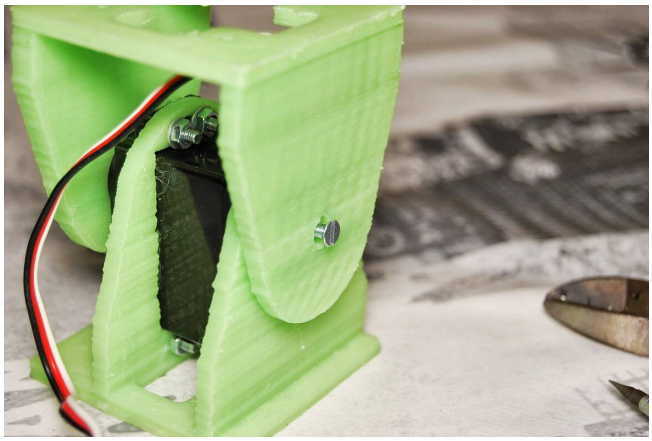
(e) Insert the horn in the corresponding hole of the upper part.

(f) The module should look like this seen from behind after inserting the horn.

Figure 4.14: Assembly of the servo horn

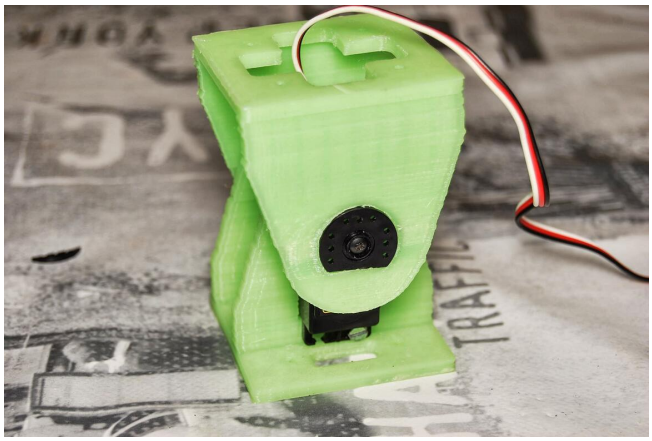


(a) Connect both parts carefully. The servo must be connected to the control board and set to 90° in order to ensure that the 90° position of the servo corresponds to the 90° position of the servo.



(b) Insert the screw for the fake axis.

(c) The screw of the fake axis should not protude from the servo body.



(d) The REPY-2 module is ready to be used!

Figure 4.15: Putting together the module

4.2 Electronic control board

For the control of the robot, a controller board was designed. For this we followed the same approach that with the module, using a previous, open source design as a reference for our board, improving and fixing all the flaws we observed on the original board.

This section explains the software and platforms used for the design of the board software and firmware. Then, it describes the previous existent boards and the discovered weaknesses of the design. Finally, our board design is presented.

4.2.1 Software and platforms used

In this section we will discuss the main software tools and microcontroller platforms used. The modular robot control board, and its firmware, were developed using the Arduino platform and, for the design of our PCB, an open source program called KiCad was employed.

Arduino

Arduino [1] is an open source electronics prototyping platform originally developed for artists, designers and hobbyists that simplifies working with microcontrolling. It is currently very extended its use in education as a first point of contact with microcontrollers, due to its simplicity.

The Arduino platform offers several development boards with different microcontrollers within a wide range of features, that can be selected depending on the project requirements. These boards go from the popular Arduino UNO board, that features an ATmega328, a 8-bit microcontroller with 32KB of flash and 2KB of RAM, to the latest, more advanced boards such as the Arduino DUE, that includes a 32-bit ARM processor running at 84 MHz, with 512 KB of flash and 96 KB of RAM. The schematics and CAD files for these boards are publicly available, which has promoted the appearance of several non-oficial derivative boards, Arduino-compatible, designed for concrete tasks, such as the Ardupilot, a board to control Unmanned Aerial Vehicles or the Skymega, a board for modular robotics, that will be discussed in a later section.

As the Arduino platform has defined a standard physical format for their development boards, some extension boards have appeared, called "shields", that are plugged in the development board, adding extra hardware to the Arduino boards, such as H-bridges for controlling motors or wireless transceivers for communication with other devices.

The principal aspects that make the Arduino easy to use are mainly two. The first one is that this platform is not limited to the hardware, but it also counts with an Integrated Development Environment (IDE) to develop the firmware to be programmed on the microcontroller and manage the board, as well as several libraries to control the microcontroller peripherals and other hardware easily. Most of these libraries have been developed by the community of Arduino users, that have published them with open source licenses for the benefit of all the Arduino community.

The second one is the bootloader for the Arduino boards, that allows the microcontroller to be programmed over a serial connection, and removes the necessity of a external programmer to load the firmware to the microcontroller. The process of compiling the code is also simplified by integrating the compiler toolchain into the Arduino IDE.

The Arduino platform has been selected over other alternatives, such as using microcontrollers from other vendors like Microchip or STMicrocontrollers because of the following reasons:

- Extensive documentation and code examples can be found online due to the openness and popularity of the platform.
- The Arduino libraries reduce the development time by offering a high-level interface to the basic and advanced features of the microcontroller.
- The availability of the schematics and CAD files allows us to develop a compatible board that adapts to our requirements based on a existent and tested design, that ensures us that the hardware will work without problems, and reduces the development time.
- All this tools can be downloaded for free from the Arduino website, as opposed to other propietary IDEs, reducing the cost of the project, as no money has to be spent on development tools.

KiCad EDA Software Suite

KiCad [13] is a multiplatform, open source software suite for Electronic Design Automation (EDA). It was started by Jean-Pierre Charras, a French researcher in the field of electrical engineering, and it is composed of four main programs that perform the different functions required for the design of electronic circuits:

- **Eeschema:** Schematic editor. This is the first step in the KiCad workflow, the creation of a schematic detailing the circuit components and connections.
- **Cvpcb:** Footprint selector for components association. Once the schematic is finished, the netlist containing all the connections is generated, and the component footprints are selected from the libraries with this program.
- **Pcbnew:** Printed circuit board editor. After assigning the footprints to the components, the footprints and netlist are loaded in the PCB editor to make the board layout and routing. Once the PCB is designed, the GERBER files can be generated in order to manufacture the PCB.
- **Gerbview:** GERBER file viewer. With this program the GERBER files can be opened and inspected to check that they are correct before sending them to the manufacturer.

For the design of the PCB in charge of the control of the robot, KiCad was used. Apart from the fact that this program is free, so no cost related to software licences was incurred, KiCad was chosen over other alternatives due to its ease of use and simplicity. Other factors that influenced the decision were that the author had previous experience with this software and the absence of limitations, such as PCB size, number of layers, etc.

4.2.2 Previous work

For the control of the modular robots built from Y1 modules some boards already existed. Instead of designing a new board from scratch, and as the schematics of those boards were available due to them being open source, they were used as reference to design our board. As we had already worked previously with the boards in other projects, we knew the limitations of the boards to be solved in our design. The improvements added, as well as a decription of the existing boards is presented in this section.

Skycube

The Skycube [23] is a board designed by Gonzalez-Gomez for controlling modular robots, featuring a PIC16F876A microcontroller running at 20MHz. It is compatible with the Y1 modules (and derivatives) and, as them, its schematics and CAD files have been released publicly with a open source license.

It allows the control of up to 8 hobby servos, with 4 connectors on each face to make their connection easier. Power and I2C communication connectors are also placed on both faces to allow connections on both sides of the board. The I2C bus can be used for communication between Skycube boards. The board also has a ICSP (In-Circuit Serial Programming) connector for burning the firmware with a programmer, and a connector breaking out the serial pins TX and RX, which is useful for serial communications and to upload firmware using a bootloader.

Even though the board counts with a connector for setting a I2C communication bus, the board is intended as a central controller for the whole modular robot, as the I2C is not a multimaster bus, and its length is limited by the capacitance of the transmission lines, as it was designed as a bus for communication between different integrated circuits inside the PCB.

A LED and a push button are present in the board, and can be used to interact with the program, supplying a simplistic input and output interface with the user. A expansion port is provided to the user to connect sensors or other hardware to extend the funtionality of the board.

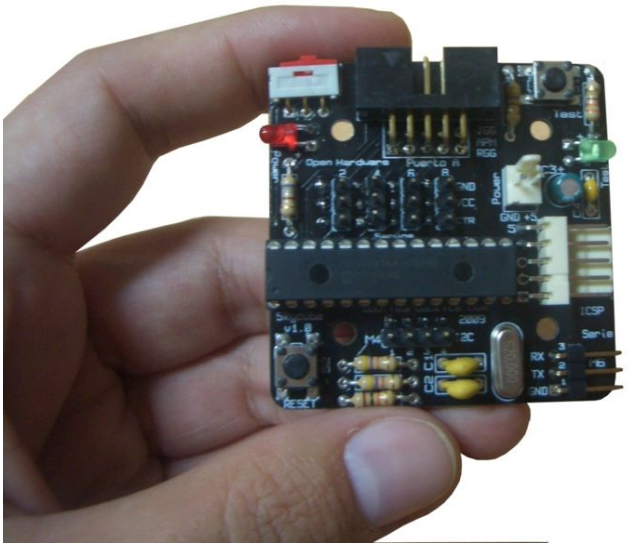
SkyMega

The SkyMega board [24] is a evolution of the Skycube board, designed also by Gonzalez-Gomez. The main improvement over the Skycube board is that this board substitutes the PIC microcontroller by a ATmega328 microcontroller from

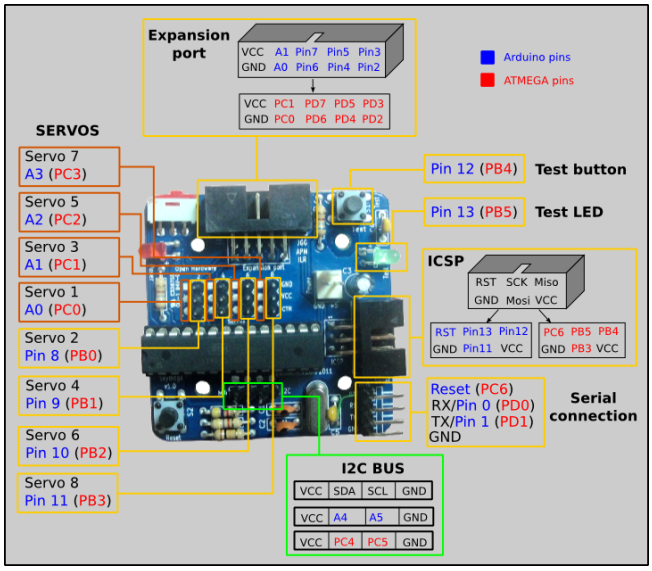
ATMEL, making the SkyMega Arduino-compatible, and allowing to use the bootloader and libraries provided by the Arduino community.

This simplifies the prototyping process, as the firmware is easier to write thanks to the Arduino libraries, reducing the time spent in developing a working robot or testing some gaits. Because it uses the Arduino bootloader, the firmware can be loaded using the serial port, and a programmer is not required.

The SkyMega board comes with several software libraries and example programs to ilustrate its usage with modular robotics. These libraries are very useful, for example, for implementing sinusoidal oscillators embedded in the board, so that the modular robot does not depend in the computer for locomotion. However, this method is only useful when testing a certain gait because, without the use of a more advanced controller, the robot cannot modify this gait according to its configuration or environment.



(a) Skycube board



(b) Skymega board with a description of the available pins and their functions.

Figure 4.16: Skycube and SkyMega control boards.

4.2.3 SkyMega SMD

Based on the SkyMega board, a new board was designed and manufactured for this project. This new board improves the original SkyMega design, and solves several problems found on the original design.

The main improvements added to the new design are:

- The board is populated with surface-mount compoments (SMD) instead of through-hole components (THD). Using SMD components reduces the amount of PCB surface required by each component. This has allowed us to design a cleaner component layout, add extra components and route the board on the top layer, leaving the bottom layer entirely as a ground plane, reducing electrical noise and interferences.
- A linear, Low-Dropout (LDO) regulator was added to the circuit in order to have a stable 5V supply to the microcontroller. A stable supply is required, for example, if using the analog inputs to measure a voltage. In this case, if the supply is not constant, the measured values will not have a common reference and will vary even if the voltage measured is the same.
- The supply for the hobby servos has been split into two different power connectors. One of them supplies the LDO regulator that powers the microcontroller and 4 of the 8 servos that the board allows to use, whereas the other

connector supplies the remaining 4 servos. The reason the supply has been split is because the hobby servos consume a high amount of current, and the inductive component generated by the DC motor introduces a large amount of noise in the power lines, that even resets the microcontroller when the 8 servos are connected to a single supply in the original SkyMega board. To reduce the effect of the motors on the power supply of the microcontroller, a capacitor was added next to each servo connector, and the power for the servos is supplied directly from the connector, not passing through the LDO regulator, so the regulated line is as clean as possible.

- The expansion port has been split into several smaller headers, to reduce the space occupied by the connector. A extra header has been added on the botton of the board for the connection with a optional bluetooth serial transceiver.
- Two LEDs for power indication on both supplies and two LEDs for serial port transmission signalling were added.

The boards were designed with KiCad and sent to be manufactured in SeeedStudio, a chinese manufacturer for very small batches of PCB prototypes. When they arrived, they were assembled by hand, and tested. The resulting assembled boards can be seen in figure 4.17.

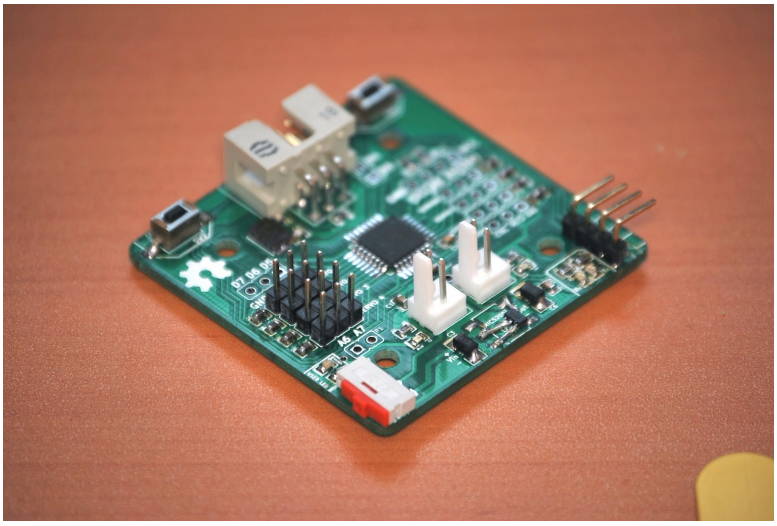


Figure 4.17: Assembled SkyMegaSMD

4.3 Other module components

Apart from the control board, the modular robot requires other elements for actuation, communication and power. In this section, we will discuss those elements.

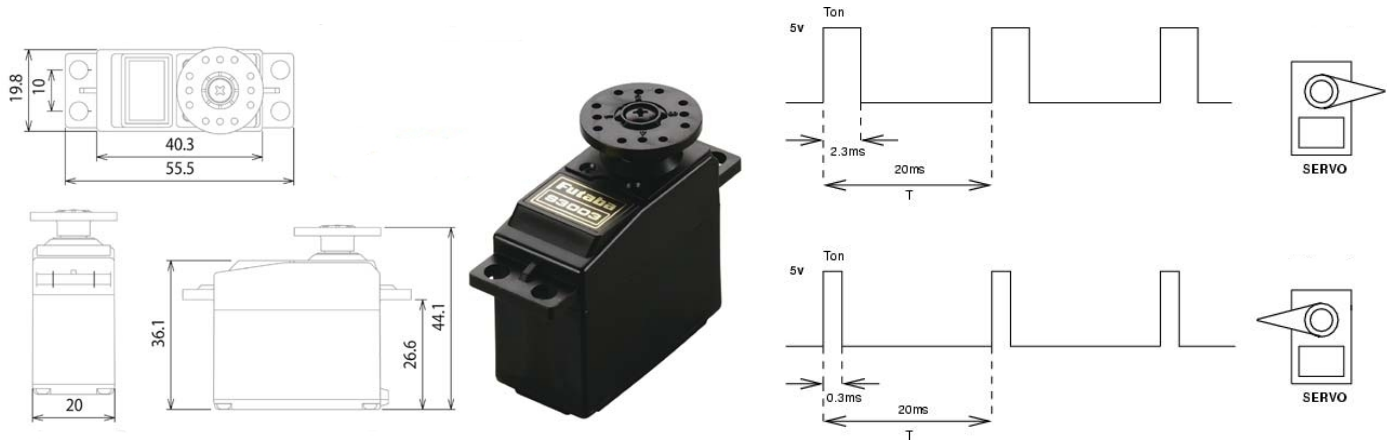
4.3.1 Hobby Servomotor

A servomotor is a motor whose position, velocity or acceleration can be precisely controlled. A servo motor includes, in addition to the motor, a sensor por position feedback and a controller that uses that feedback for controlling the output.

For this robot, we have chosen Futaba 3003s hobby servos, that are cheap servomotors intended for Radio Control vehicles. These servos were selected because of their lower price compared with more advanced servos intended for robotics and their availability.

These servos are composed by a small DC motor coupled to a plastic reduction gearbox. The output shaft is connected to a potentiometer for position feedback, and the motor is controlled by a controller board placed inside the servo, that takes as input signal a pulse-width modulated signal (PWM). These signal controls the position of the servo output, that restricted from 0^o to 180^o, as a function of the width of a periodic pulse of 50Hz. With a pulse of 0.3ms of width, the servo joint is placed at one of its extremes, and with a pulse of 2.3ms the servo axis moves to the other extreme. Intermediate

values place the shaft in a position between the two extremes. For example, to place the joint centered at the middle of the range, a pulse of 1.3ms has to be sent. If no signal is received, the servo is in standby, and the servo does not oppose to external movements of the shaft.



(a) Servo Futaba3003s and dimensions

(b) PWM signal for position control

Figure 4.18: Servo Futaba 3003s

4.3.2 USB-to-Serial converter cable

For firmware upload and communication with the control board, a serial connection with the computer is used. As nowadays most computers do not have a serial connector, a converter is required to connect the serial port of the control board to a USB port on the computer. This converter translates both protocols and allows communication between the computer and the board. As the computer does not have an actual serial port, this port is simulated in software on the computer.

The adapter used in the robot is a cable that integrates a FTDI FT232RL USB/serial chip, which translates the USB communications sent by the computer to the TTL RS-232 signals (RX, TX, CTS, RTS) used by the Universal Asynchronous Receiver/Trasmitter (UART) of the microcontroller. The UART is the peripheral of the microcontroller used for serial communication with multiple communication standards, data formats and transmission speeds.

Figure 4.19: USB-to-Serial converter cable

4.3.3 LiPo Battery

In order to be autonomous, the robot has to carry its own batteries for power. The batteries used for this robot are Lithium Polymer (LiPo) batteries. These batteries are rechargeable, and offer a very good ratio capacity / weight.

Each cell of a LiPo battery outputs around 4.2V when charged, and they are combined in parallel for a higher capacity or in series for a higher voltage. The number of cells is series is indicated in the battery with a number followed by the letter 'S'. A 1S battery has one cell, and will output 4.2V when charged, whereas a 2S has two cells and will output 8.4V when charged. To extend the lifetime and be able to be charged safely, these cells must be balanced (i.e. they must have a similar internal resistance).

The capacity of the LiPo battery is measured in mAh, the amount of milliamps that they can supply in an hour at a certain voltage (usually the battery voltage). For example, a 1200 mAh battery can supply 1.2A during one hour, or 0.6A during two hours, etc. The maximum discharge rate allowed by the battery is expressed as a number followed by the letter 'C', that multiplied by the capacity gives the maximum current that can be drawn. A 10C, 1200mAh battery can supply up to $10 \cdot 1.2 = 12A$.

The advantages of LiPo batteries are their high capacity / weight ratio, as mentioned before, and the high discarge current that they can support, and the main drawbacks are that they have to be charged with a specialized charger, as all the cells must be kept balanced to prevent them to ignite.

For this project, a 3S (12.6V) battery was used, with a capacity of 2200mAh and a maximum discharge rate of 20C ($20 \cdot 2.2 = 44A$).

Figure 4.20: LiPo battery 3S 2200mAh

4.3.4 UBEC

The 3S LiPo battery supplies 11.1V, but the servos and the control board both work at 6V. In order to adapt the supply voltage to the suitable levels, a UBEC is used. A UBEC is a switch-mode DC to DC converter very common in RC planes and helicopters to supply power to the control board, transmitter and servos that actuate the control surfaces from the same battery that powers the motors, eliminating the necessity of having a dedicated battery with a lower voltage for them.

The UBEC chosen for this robot is a Turnigy UBEC that can supply up to 8A (or up to 15A for a very short period of time). It accepts input voltages from 6V to 12.6V, which correspond to 2S or 3S batteries, and has a selectable output of 5V or 6V. It also has LEDs for indication of the current battery level.

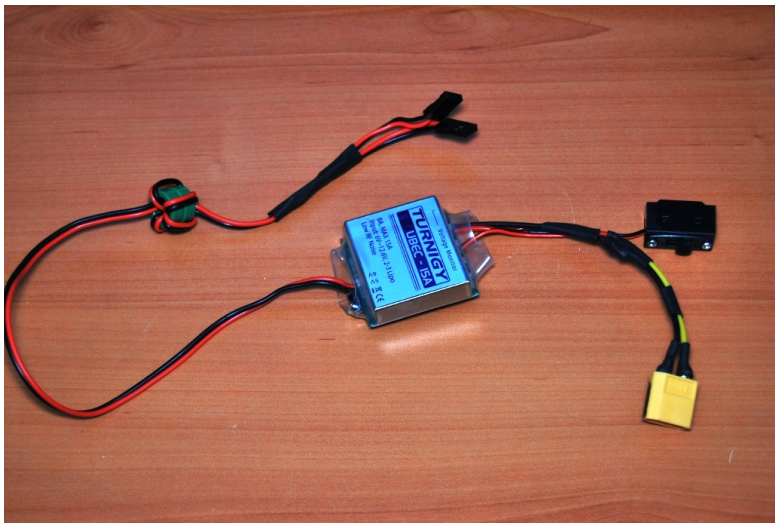


Figure 4.21: UBEC Turnigy 8A

4.3.5 Bluetooth module

The cables connected to the robot for power and communication limit the movement of the robot, affecting to the resulting gait. The cables for power can be removed by adding batteries to the robot and, for removing the cable used for communication, a Bluetooth module was tested.

Bluetooth is a wireless technology standard used for data transmission over short distances, using the 2.4 GHz to 2.485 GHz band. It is used for communication between fixed and mobile devices, and between mobile devices, for building wireless personal area networks (WPANs). It was invented by Ericsson, as a wireless alternative of wired RS-232 communications.

The Bluetooth module used was a JY-MCU, which includes a HC-06 bluetooth transceiver. These modules are very cheap, and they are able to transmit a serial connection over bluetooth using the RFCOMM protocol, taking care of the bluetooth protocol and implementing a virtual serial data stream. On the other side, the communication range spans only a few meters, but that range is enough for our application.

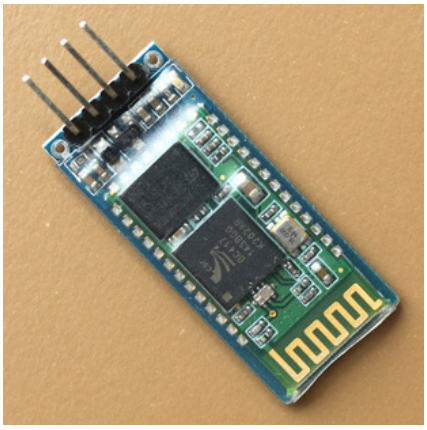


Figure 4.22: Bluetooth module JY-MCU

4.4 Firmware

To control the modular robot from the computer, a very simple firmware was developed. This firmware opens a serial port to connect with the computer and waits for commands. With these commands, the position of the joints can be specified, as well as the state of the onboard LED.

Since the number of servos that a single board can drive is limited to 8, for the configurations with more than 8 servos two boards have to be used. These boards are connected to each other using the I2C bus, with a board acting as a master and the other acting as a slave. The master board is also connected to the computer via serial port to receive the commands. The commands are sent from the computer at a certain rate, updating the joint position according to the oscillator output values. Those oscillations generate the locomotion of the modular robot.

The steps followed by the firmware are the following:

1. The microcontroller UART is configured and the serial port is open. When the robot is ready, it sends a string "Ok!" to the computer, indicating that it is waiting for commands. This string is used by the software on the computer to check that the robot is listening to the commands.
2. The microcontroller waits for a command. The received bytes are stored in a buffer and when a command byte is received, the remaining bytes of the command are processed. The start byte is composed of a pattern of 4 bits (0101) followed by 4 bits indicating the command, so all the available bytes for being used as a command go from 0x50 to 0x5F.
3. The command is processed. Currently there are 4 commands available:
   - **0x50**: Set the position of all the joints. After this byte, the firmware expects as many bytes as joints have the robot, each byte containing the angular position of that joint, from 0º to 180º. For a robot with two joints, to be set at 45º and 30º respectively, the command would be: 0x50 0x2D 0x1E.
   - **0x51**: Set the position on a single joint. This command needs two bytes to be received after the command, the index of the joint to be set and its value. If the third joint of the robot is to be set at 50º, the command would be: 0x51 0x02 0x32 (the joint indexes start with 0, so the third joint has index 2).
   - **0x52**: Send message to another board. Since the number of servos that a control board can drive is limited to 8, for configurations with more servos two boards have to be connected using the I2C bus. When two boards are connected, the master board can relay messages to the slave board using this command. This command requires the size of the message to be sent to be specified after the command. For example, if the would like to set the third joint of the slave board to 50º (command 0x51 0x02 0x32), we would have to send the following command to the master board: 0x52 0x03 0x51 0x02 0x32.
   - **0x5F**: Test command. It just toggles the USER LED on the control board, and can be used to test the connection with the board visually.
4. Waits for another command to execute.

The firmware of the slave board has to slightly different, as the commands are received by the I2C bus, instead of the serial port, and it does not accept the "Send message" command (0x52), since there is no other board connected to the slave board except for the master board. This slave board firmware has not been developed to to time limitations, and is left as future work.

The use of this firmware implies that the modular robot is a dummy robot: the actual controller that generates the joint positions is run on the computer, and the robot control board only sets those values to the actual robot in order to check the resulting locomotion.

Chapter 5

Modular Robot Configurations and Gaits

5.1 Modular Robot configurations

Modular robots have the versatile capability of being able to reconfigure themselves, adapting to the environment and the task they must perform. They could, for example, be configured as a legged robot to step over an obstacle and then reconfigure to a snake robot to travel along a pipe. This ability renders them very useful in unknown and unstructured environments, where they excel other mobile robots.

In order to be studied, this variety of possible configurations require a classification, since the locomotion gaits that each configuration is able to perform and how to perform them change notably among the different configurations. The controller also varies with the configuration, and more complex configurations, such as legged ones, require more complex controllers to deal with coordination between all the limbs, between the limbs and body and in general, between all their joints to achive a optimal gait.

5.1.1 Classification by the arrangement of their basic unit

One common way of classifying the different types of modular robots and their configurations is to base it on the arrangement of their basic unit, classfying them in lattice type, chain type, or hybrid type.

Lattice modular robots have their modules arranged in some regular pattern along 3D space, resembling atoms in crystals and their configurations are usually described using crystallographic displacement groups. These kind of lattice structures are computationally simpler to describe, and their reconfiguration planning can be scaled easier to more complex systems. In fact, locomotion in this kind of modular robots is achieved by reconfiguration, changing the position of individual modules in that lattice in such a way that the global position of the modular robot is displaced towards the goal position.

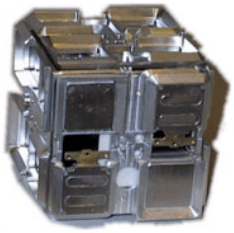
Some examples of modules that can be classified as lattice type are ATRON, Telecube, Digital Clay, or CHOBIE, and can be seen on figure 5.1.

In *chain* (also called *"tree"*) modular robots, the modules are connected forming strings or trees, allowing this kind of robots to reach any point of space. In constrast to this versatility, these modular robots usually need a chain of several modules to reach an arbitrary point, making reconfiguration more complex, and they are more computationally difficult to represent and analyze.

Since reconfiguration is more complex in *chain*-type modular robots, they usually achieve locomotion by means of their own bodies or limbs made of modules, performing oscillating patterns with their joints in order to progress towards the goal.

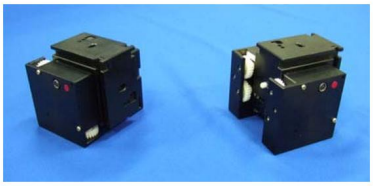
Figure 5.2 contains some examples of chain-type modules, such as CONRO, PolyBot or Y1, on which the modules used in this thesis are based.

(a) ATRON



(b) Telecube

(c) Digital Clay



(d) CHOBIE

Figure 5.1: Examples of lattice-type modular robots

The last category of modular robots according to the arrangement of their basic unit are *hybrid* modular robots. These modules have characteristics of both lattice-type and chain-type, allowing them to perform as any of them when required.

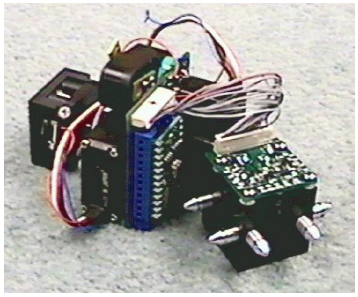
For reconfiguration the robot behaves as a lattice-type modular robot, since it is easier to model and calculate the steps required for reconfiguration in this type of modules, whereas locomotion is achieved as a chain-type modular robot, obtaining higher speeds and maniobrability this way.

Figure 5.3 shows some of the existing *hybrid* modular robots, such as M-TRAN, Superbot or SMORES.

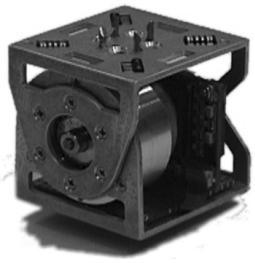
5.1.2 Chain-type configurations

REPY-2.1, the module used in this work, only allows us to build *chain*-type modular robots, and therefore the remaining part of this chapter will focus on them. *Chain*-type modular robots, whose modules form strings or trees, have also several possible configurations, depending on the number of dimensions the trees span (1D, 2D or 3D), that will be addressed in this section.

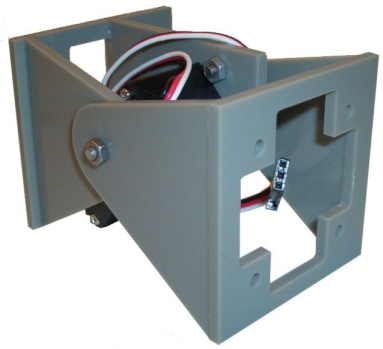
1D chain modular robots resemble snakes or worms and may achieve their gaits with the help of wheels or tracks,



(a) CONRO

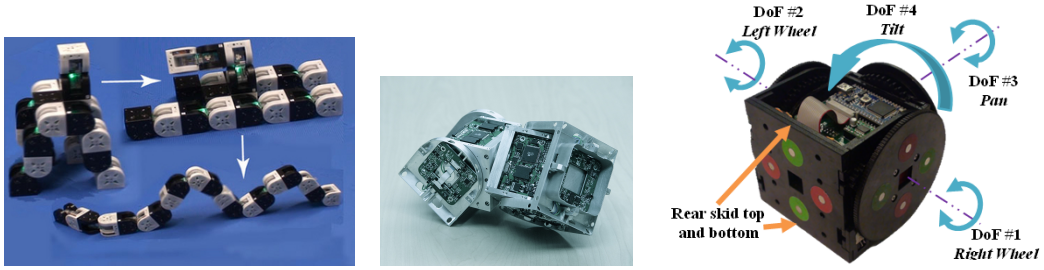


(b) PolyBot (G3)



(c) Y1

Figure 5.2: Examples of chain-type modular robots



(a) M-TRAN III (b) Superbot (c) SMORES

Figure 5.3: Examples of hybrid-type modular robots

either passive of active (i.e. "serpentine" robots), or without them (i.e. "snake" robots), just with their body. The gaits achieved by these configurations can be similar to the side-winding gaits of a snake, to a caterpillar gait or a mix of both, depending on the orientation of their DOFs. They also count with other gaits such as turning or rolling over themselves.

These 1D configurations are useful for accessing complicated and narrow places, such as holes in the debris or all kinds of pipes. With the suitable gait, these robots can also climb pipes at angles up to 90^o though the interior or the exterior of the pipe. One of the most advanced examples of snake robots has been built at the Biorobotics laboratory at the Carnagie Mellon Universy, which can perform all the mentioned gaits, climb trees and even twist around a tree branch when throw onto them.

Modules that possess connectors on their sides allow 2D configurations. This group of configurations include legged robots (like tripods, quadruped, hexapods, miriapods) and other more exotic mesh-like robots. 2D configurations are more complex to describe and model, but they are usually more stable as the count with a larger number of support points over a wider area.

2D legged robots can have 1 or more DOF per limb. This thesis is focused on legged robots with multiple DOFs per limb. This kind of configurations require not only coordination among the different limbs (inter-limb coordination) but also coordination among the distinct DOFs inside of each limb (intra-limb coordination) in order to generate a viable gait. They are also more difficult to control, as the longer the limb, the easier from them to collide with other modules in the robot, so the controller has to take into account these contraints when generating the joint values for each module.

The last type of chain-type modular robot are 3D configurations. These kind of configurations are very rare, since they are much more complex than 2D configurations, and usually more unstable and harder to reconfigure, which makes them less practical than 2D configurations. One of the few examples from this category are the Roombots, developed by Auke Jan Ijspeert at the École Polytechnique Fédérale de Lausanne (EPFL)[48], whose main goal is to develop adaptative furniture than can adapt to the user needs and reconfigure in whatever piece of furniture that is needed by the user at that moment. Note that the Roombots are hybrid-type modular robots, and for the reconfiguration of the different pieces of furniture the lattice mode is used.

Figure 5.4 shows some examples of the three configurations of chain-type modular robots.

5.1.3 REPY-2.1 available configurations

The modules used in this thesis are REPY-2.1 modules, a derivative module from the Y1 module designed by Juan Gonzalez-Gomez. REPY-2.1 are very cheap and simple, since they only have 1 DOF controlled by a hobby servo, a 3D printed plastic structure, and the robot is controlled by a central control board, but they lack the features of other more expensive modular robotic platforms, such as self-reconfigurability, independent in-module control board and autonomy. Despite the lack of those features, these modules are a good platform for researching locomotion gaits for modular robots, since locomotion is achieved in chain-type robots without involving reconfiguration. For the research of the locomotion gaits is enough with the "skeleton" of the modular robot, and the REPY-2.1 provides that "skeleton" in a cheap and simple way.

(a) 1D: CMU Snake Robot (b) 2D: Superbot as mini-humanoid (c) 3D: Roombots configured as a chair

Figure 5.4: Examples of the different configuration types of chain modular robots

These modules are interconnected by hand, using screws, and carry a total of 4 connectors, placed in two pairs located in two ortogonal planes. Therefore, REPY-2.1 modules allow both 1D and 2D configurations and, as the module do not possess any means of self-reconfiguring, only chain-type configurations are of interest. They are also genderless and symmetrical, so each pair of connectors can be connected in 4 different positions, with an offset of 90°, yielding a very high number of possible ways of connecting the modules.

For the study of the locomotion gaits in legged modular robots with multiple degrees of freedom per limb, three of these configurations are selected. The first of them is a configuration with 2 DOF per limb, 4 limbs, and made of 11 modules, called *"MultiDof-11-2"*, shown in figure 5.5.

If we remove the two front limbs (Modules 6, 7, 10 and 11), we obtain a tripod configuration made of 7 REPY-2.1 modules called *"MultiDof-7-tripod"*, and shown in figure 5.6.

The last of the configurations is obtained by attaching a fourth leg to the central module of the *"MultiDof-7-tripod"* configuration, obtaining a quadruped configuration made of 9 modules and called *"MultiDof-9-quad"*, that is shown in figure 5.7.

5.1.4 REPY-2.1 configuration description

As described in the previous section, the REPY-2.1 modules can be connected in a large variety of ways. In order to describe the current location and orientation of a module inside the modular robot, we need to encode all those different

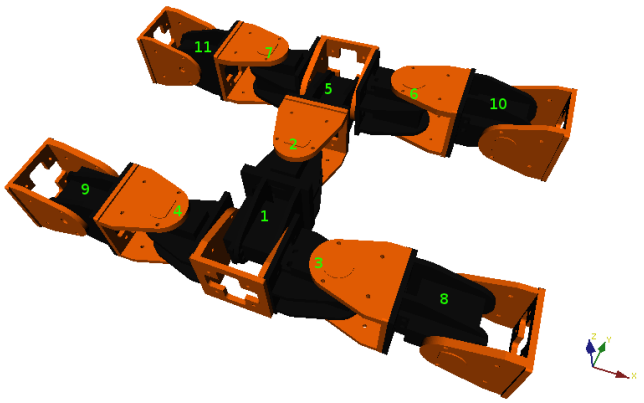


Figure 5.5: MultiDof-11-2

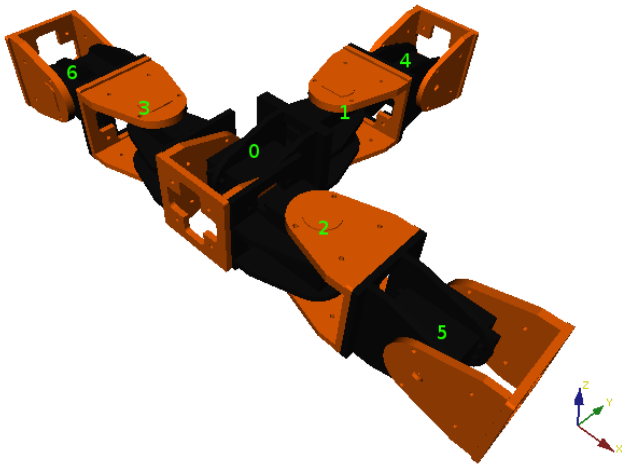


Figure 5.6: MultiDof-7-tripod

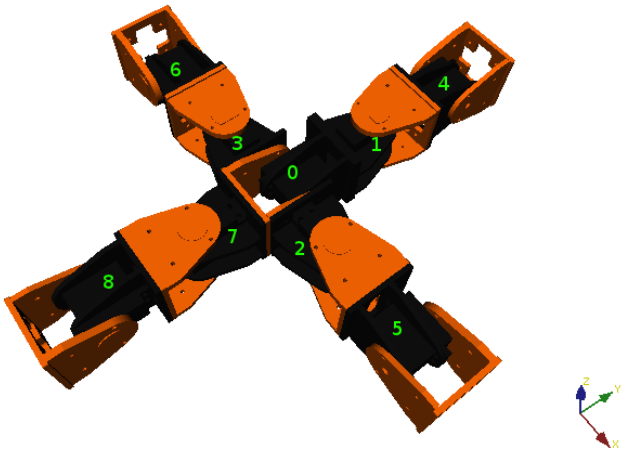


Figure 5.7: MultiDof-9-quad

ways of connecting the modules, to be able to save the locomotion parameters from a module to the gait table, and later assign those parameters to a module in the same position, that is, performing the same function.

The first thing to encode is which of the connectors of the module are being used and which connectors of the neighbor modules are they attached to. Each module has a total of 4 connectors: front, right, back and left, which are encoded in that order to integers from 0 to 3. When two connectors are attached together, as they are symmetrical, they can be connected in four different orientations, each of them obtained from rotating the module to be connected in 90º steps around the vector normal to the connector. Since they can only be connected in steps of 90º, the relative orientation between modules also can be encoded using integers from 0 to 3, representing orientations of 0º, 90º, 180º and 270º.

It is possible to calculate the number of possible combinations between modules in an easy way. Let us consider the possible combinations between a single connector of a module and any of the connectors of other module, we have 4 possible connectors to attach to it, and 4 different orientations in which attach them, plus an extra possibility of not having a connector attached to out connector, which yields $4^2+1 = 17$ possibilities. Since we have 4 different connectors in the module, and each connector has 17 possible ways of connecting a module, we have a total of $17^4 = 83521$ combinations.

If we were to consider the possible combinations considering also the level 1 neighbors (the neighbors of the neighbors of

the considered module), this amount would increase exponentially, and it would be computationally expensive to calculate and store this encoded configurations for each module. That is the reason why in this thesis only level 0 neighbors are considered and, when the ID obtained is ambiguous, some extra info is used to resolve the ambiguities.

To encode the different configurations to a number, the following formula is used:

$$ID = \sum_{i=0}^{3} [(x_0^i \cdot 4^0 + x_1^i \cdot 4^1) \cdot x_2^i + 16 \cdot (1 - x_2^i)] \cdot 17^i \tag{5.1}$$

$$\begin{aligned} ID = {} & [(x_0^0 \cdot 4^0 + x_1^0 \cdot 4^1) \cdot x_2^0 + 16 \cdot (1 - x_2^0)] \cdot 17^0 + [(x_0^1 \cdot 4^0 + x_1^1 \cdot 4^1) \cdot x_2^1 + 16 \cdot (1 - x_2^1)] \cdot 17^1 + \\ & + [(x_0^2 \cdot 4^0 + x_1^2 \cdot 4^1) \cdot x_2^2 + 16 \cdot (1 - x_2^2)] \cdot 17^2 + [(x_0^3 \cdot 4^0 + x_1^3 \cdot 4^1) \cdot x_2^3 + 16 \cdot (1 - x_2^3)] \cdot 17^3 \end{aligned} \tag{5.2}$$

Where:

- i is the local connector to be considered, encoded as integer from 0 to 3. 0 corresponds to the *front* connector, 1 to the *right* connector, 2 to the *back* connector and , 3 to the *right* connector.
- x_0^i corresponds to the connector of the remote module that is connected, also encoded as integer from 0 to 3. The encoding is identical to the local connector: 0 corresponds to the *front* connector, 1 to the *right* connector, 2 to the *back* connector and , 3 to the *right* connector.
- x_1^i corresponds to the relative orientation between the two modules considered, encoded as an integer from 0 to 3. This integer expresses the number of 90^o steps around the normal of the connector face required to achieve the given orientation from the default one. A formal description of this parameter is given later on this section.
- x_2^i can be either 0 or 1, and represents whether the connector i has a module connected or not. 0 represents "no module connected" and 1 that the connector is active.
- 4, 16 and 17 are constants representing the number of possibilites of each parameter. Remote connector and orientation are expressed in base 4, since the number of possible values is 4, whereas 17 represents the number of possible combinations of connector and orientation, plus the possibility of not having a module connected. As those combinations go from 0 to 15, 16 is used when there is no module attached.

Each pair of connectors can be attached together in 4 different orientations, each of them with a difference of 90^o, since the connectors are symmetrical. In order to calculate this orientation, the orientation of the local reference system of each of the modules with respect to an absolute frame is used. This relative orientation is obtained with a Inertial Measurement Unit (IMU) containing an accelerometer, a gyroscope and a magnetic compass, using the Earth's gravity and magnetic field to orientate the sensor with respect to a reference frame fixed on the Earth and returning that orientation expressed in Tait–Bryan angles (Roll, pitch and yaw). These sensors are currently not available in the physical modular robot due to hardware limitations, so the values are set by hand in the configuration file, and later read from it.

Two of the three degrees of freedom that the module orientation has are determined by the connectors used to attach the modules, being that the reason why we only need to know one angle to determine the relative orientation of the connectors. This angle is defined as *"the angle we have to rotate the local module around the axis in the same direction as the normal vector of the local connector surface, such as the Z axis of the local reference system of both modules coincide"*. This axis corresponds to the X axis for the side connectors (left and right) and the Y axis for the front and back connectors, as shown in figure 5.8. This method assumes that the configuration in 5.9a is the default one, and the remaining ones are generated by rotating the remote module by a certain angle.

Figure 5.9 represents the 4 possible orientations of a simple 2-module configuration with their corresponding orientation value for each of them. If we are calculating the orientation from the rightmost module point of view, we observe that the remote module is connected to the front connector of the local module, and therefore the Y axis of the local module the axis of reference. Around this axis, the local module is turn in 90^o steps until both Z axis are coincident. The number of steps required is the value of the orientation parameter.

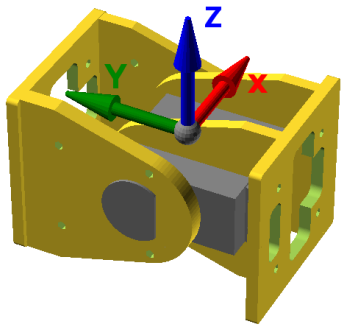


Figure 5.8: Reference system for REPY-2.1 modules

Let us calculate the ID of the third configuration in figure 5.9, configuration c, using equation 5.1:

$$ID = \sum_{i=0}^{3} [(x_0^i \cdot 4^0 + x_1^i \cdot 4^1) \cdot x_2^i + 16 \cdot (1 - x_2^i)] \cdot 17^i$$

Only the front connector is active, so x_2^i equals 0 for connectors 1, 2 and 3. As the front connector is connected to the back connector (encoded as '2') and their relative angle is 180º (encoded also as '2'), the ID would be:

$$ID = (2 \cdot 4^0 + 2 \cdot 4^1) \cdot 17^0 + 16 \cdot 17^1 + 16 \cdot 17^2 + 16 \cdot 17^3 = 83514$$

Figure 5.10 shows another 2-module configuration, but this time the module is attached to the left connector using its back connector. In this case the reference axis is the X axis of the local module, and is important to notice that the axis of rotation is located in the direction of the local refence system, not in the direction of the connector normal vector. By rotating the remote module in steps of 90º, the 4 possible configurations are generated.

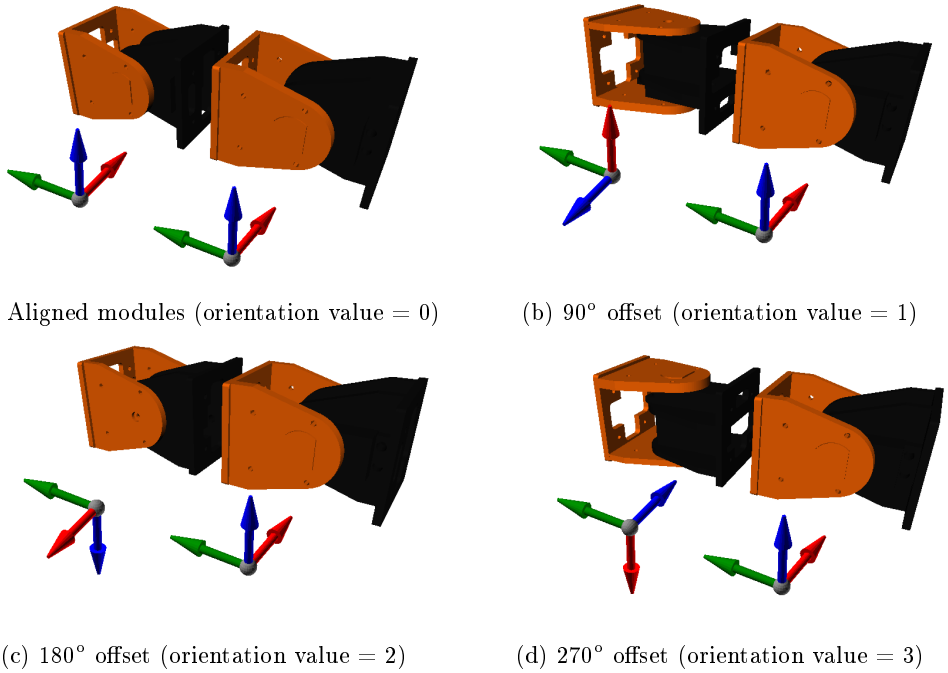
To calculate the ID of the last configuration in figure 5.10, configuration d, we apply again equation 5.1. In this case, the active connector is the left connector, making $x_2^i = 0$ for connectors 0, 1 and 2. If we substitute the remote connector (back connector,'2') and the relative orientation (270º, encoded as 3) it yields an ID of:

$$ID = 16 \cdot 17^0 + 16 \cdot 17^1 + 16 \cdot 17^2 + (2 \cdot 4^0 + 3 \cdot 4^1) \cdot 17^3 = 73694$$

Finally, in figure 5.11 we can observe a 4 module configuration in which the central module has 3 active connections. For finding the relative orientation of the side connectors, the X axis is used, obtaining a orientation of 270º for the module attached to the right connector and a orientation of 90º for the module attached to the left connector, enconding them as 3 and 1, respectively (the central module is upside-down, so the left hand module in the figure corresponds to the right connector of the central module). For the back connector module, the local Y axis is used, obtaining a relative orientation of 90º, encoded as 1.

If we substitute these values in equation 5.1, we can calculate the ID for the central module:

$$ID = 16 \cdot 17^0 + (2 \cdot 4^0 + 1 \cdot 4^1) \cdot 17^1 + (2 \cdot 4^0 + 1 \cdot 4^1) \cdot 17^2 + (2 \cdot 4^0 + 3 \cdot 4^1) \cdot 17^3 = 31466$$



(a) Aligned modules (orientation value = 0)

(b) 90° offset (orientation value = 1)

(c) 180° offset (orientation value = 2)

(d) 270° offset (orientation value = 3)

Figure 5.9: Examples of relative orientation on a 2-module configuration

5.2 Gait generation

Modular robot locomotion, with the large number of degrees of freedom that it usually involves requires highly coordinated gaits in order to be efficient and effective. Several approaches to solve this coordination problem can be found in the literature, and have been already described on section 2.2.

Gait tables are a simple approach, but they are mainly used with a central control paradigm. They also lack flexibility, and their complexity increases with an increase in the number of degrees of freedom of the robot.

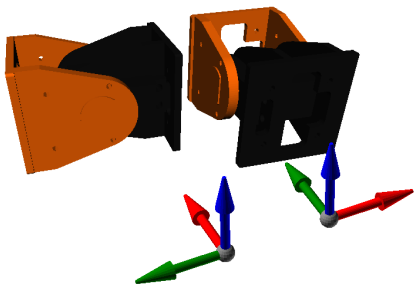
CPGs are very powerful controllers, but they are also very complex to model and implement, and highly redundant. These CPG mathematical models are useful for neurocomputing scientists to study biological CPGs and model neural circuits, as they can be tested either on simulations or on real robots, and therefore validated. In robotics, on the other hand, one is often more interesting in efficiency, in obtaining the best possible gaits using as less resources, computing power and power as possible.

Sinusoidal oscillators, as a simplification of CPGs, are the controller chosen for this work. The main reasons for choosing sinusoidal oscillators are:

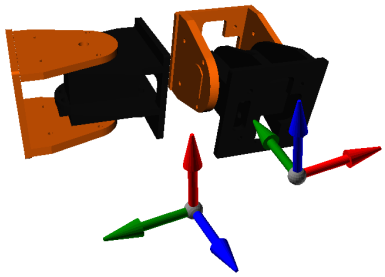
- Sinusoidal oscillators are simple and easy to model and implement. As the joint position follow a sinewave, they do not require a lot of computing power for their execution, so they can be embedded on a simple microcontroller.
- Once their paremeters are selected and set, they can oscillate independently from the rest of oscillators, making them more robust against communication problems. As each step of the joint position is generated by the oscillators, the modules can be synchronized less often, and a greater bandwidth is available for communicating other kind of messages.

For modelling the sinusoidal oscillators, the following equation is used:

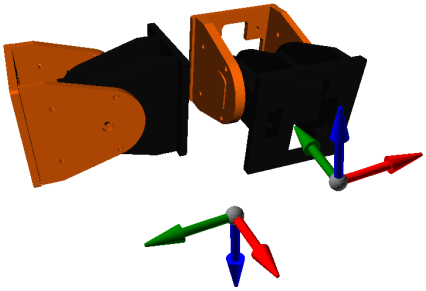
$$\varphi_i(t) = A_i \cdot \sin\left(\frac{2\pi}{T} \cdot t + \Phi_i\right) + O_i \qquad i \in \{1, ..., N\}1 \tag{5.3}$$



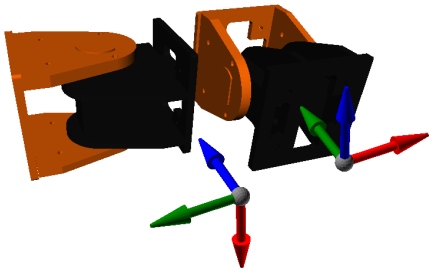
(a) Aligned modules (orientation value = 0)



(b) 90° offset (orientation value = 1)



(c) 180° offset (orientation value = 2)



(d) 270° offset (orientation value = 3)

Figure 5.10: Examples of relative orientation on a 2-module configuration

Symbol	Description	Range
$\varphi_i(t)$	Position of the ith joint	[-90, 90] degrees
A_i	Amplitude of the ith oscillator	[0, 90] degrees
T	Period of the oscillator	T $>$ 0 seconds
t	Elapsed time	t $\leq$ 0 seconds
Φ_i	Initial phase of ith oscillator	[0, 360] degrees
O_i	Offset of ith oscillator	[-90, 90] degrees
N	Number of modules in the robot	$N \geq 2$

Table 5.1: Parameters of the sinusoidal oscillator

The frequency of the oscillators does not affect the coordination, but the speed of the gait. The main parameter behind gait coordination is the phase of the oscillator, defined as:

$$\phi(t) = \frac{2\pi}{T} \cdot t \tag{5.4}$$

Which can be substituted in equation 5.3 to get the joint value of the ith oscillator as a function of the phase:

$$\varphi_i(\phi) = A_i \cdot \sin(\phi + \Phi_i) + O_i \qquad i \in \{1, ..., N\}1 \tag{5.5}$$

As the modules are physically actuated by a hobby servo, with a mechanical restriction of 180 degrees, we have limited the oscillator joint values to a range of [-90, 90] degrees, imposing the following restriction to the oscillator parameters:

$$|O_i| + A_i \leq 90 \tag{5.6}$$

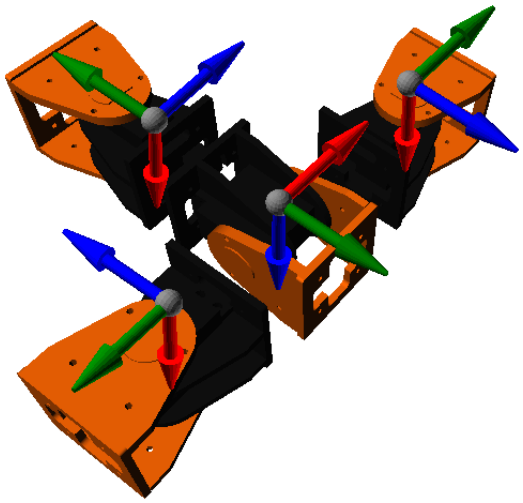


Figure 5.11: Configuration with 4 modules

Figure 5.12 helps to understand the physical meaning of the equation parameters. In this case, the amplitude is 45º and the offset is 0º, so the movement is centered around the 0º joint position. The minimum position reached by the joint is $O - A = -45º$ and the maximum one is $O + A = 45º$, following a sinusoidal waveform as a function of time.

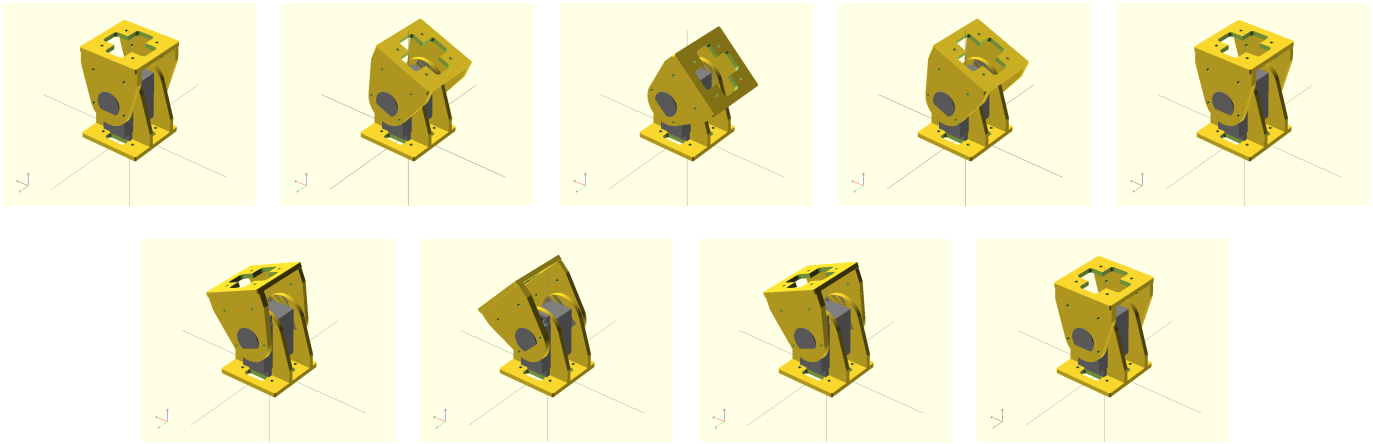


Figure 5.12: Sequence of module oscillation for $A_i = 45º$, $O_i = 0º$

In figure 5.13 the oscillator has the same amplitude as before, 45º, but the offset has been set to -45º. We can observe that in this case the movement is centered around -45º, with a minimum position at $O - A = -90º$ and a maximum one at $O + A = 0º$.

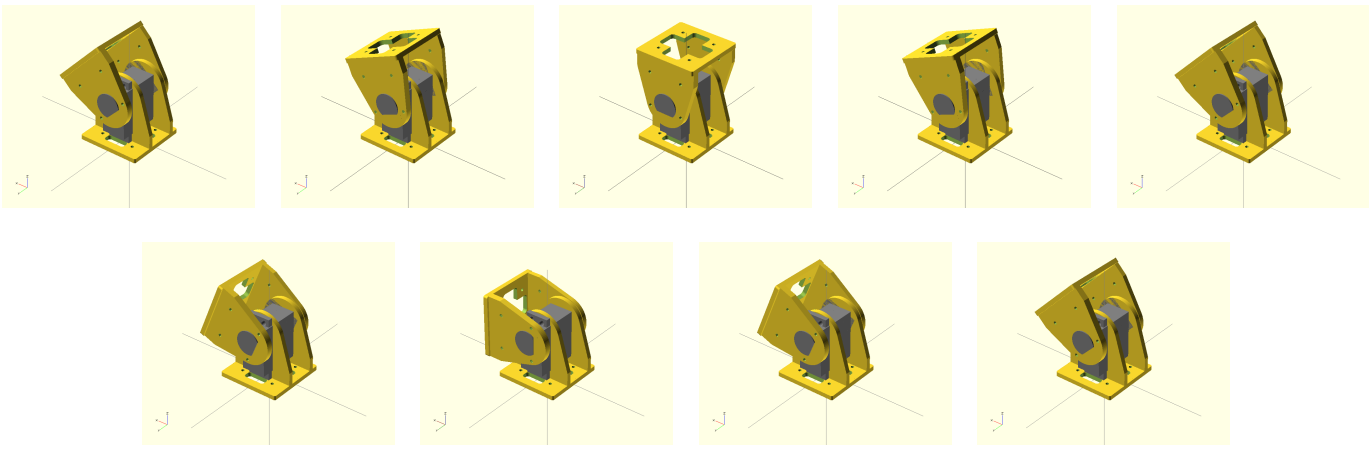


Figure 5.13: Sequence of module oscillation for $A_i = 45^o$, $O_i = -45^o$

5.3 Evolving Gaits

When the number of modules in the modular robot is high, or the distribution of the modules is complex, setting the parameters of the sinusoidal oscillators by hand becomes an almost impossible task, and the gaits obtained by this method are very far from the optimal ones.

As it is very easy to evaluate in simulation how much did a modular robot travell in a certain period of time with a given set of oscillator parameters (A_i, O_i, ϕ_i and T), we can try random values for those parameters and use the ones that yield better gaits. This is the approach taken by the stochastic optimization algorithms, such as simulated annealing, particle swarm optimization or genetic algorithms. Differential evolution, the algorithm used in this thesis for optimizing the oscillator parameters belongs to the later group of stochastic optimization algorithms.

5.3.1 Differential Evolution

Differential evolution (DE) [49] is a iterative method to optimize a multidimensional real-valued function by keeping a population of candidate solutions that is improved over time using simple arithmetic operations between the individuals of the population. Since it takes few or any assumptions on the function to be optimized, and can search over a large space of candiate solutions, this method is called a metaheuristic method. As a drawback, metaheuristic methods do not ensure a optimal solution, but for many applications the solution obtained with these methods is good enough for meeting the requirements.

Differential evolution is inspired by biology and the evolution of living beings, keeping a population of candidate solutions from which only the fittest survive and produce offspring by mixing the candidate solutions, replacing the worst individuals from the population. This way, the population is improved and the parameters to optimize (genotype) are closer to the optimal solution. In order to escape from possible local minima existing in the function to optimize, the concept of random mutation is introduced, so that after a certain number of iterations an individual can have some of its variables randomly modified, in order to create genetic variability in their offspring. The fitness value for indidual is given by the cost function to maximize, that for this thesis is the average speed of the modular robot through the evaluation time. The process finishes when a certain level of fitness is reached (the algorithm as found a "good enough" solution) or after a given number of iterations.

This algorithm, as opposed to other optimization methods such as Gradient Descend, does not use the gradient of the function for locating the minima/maxima of the function, and therefore does not require the cost function to be differentiable. Furthermore, the cost function is seen as a "black box" by the differential evolution algorithm, that uses it to evaluate how good a candidate solution is, and to check whether or not a new candidate obtained by recombination is better than its parents in order to substitute their parents by it on the population, making this algorithm a good choice

for a wide range of optimization problems with a cost function difficult to model analitically, such as the gait optimization problem.

5.3.2 Algorithm

Let us have a cost function $J(\theta) : \mathbb{R}^n \to \mathbb{R}$ to maximize and a population of candidates to be the maximum of the function where $x_{\in}\mathbb{R}^n$ denotes an individual of that population. The algorithm for Differential Evolution is as follows:

Algorithm 1 Differential Evolution algorithm

```
Random initialization of all individual x in the population
while finalization criteria is not met do
    for all x ∈ population do
        Pick three random individuals a, b, c from population different than x
        Generate random number R ∈ {1, ··· , n}, where n is the genotype size
        Compute the candidate to be the new position of the individual y = [y_1 ··· y_n] with this procedure:
        for i := 1 to n do
            Pick uniformly distributed number r_i ≡ U(0, 1)
            if r_i < CR or i = R then
                y_i ← a_i + F · (b_i − c_i)
            else
                y_i ← x_i
        if J(y) > J(x) then
            Replace x with y in the population
Pick the fittest individual as solution
```

Finalization criteria, as explained before, can be either a given number of iterations reached, or that the best individual has a fitness value above a given threshold.

Some important parameters that control the behavior of the algorithm are $F \in [0, 2]$, called *differential weight*, that controls the amplification of the differential variation (a value of 0.8 is suggested) , $CR \in [0, 1]$, called *crossover probability*, the probability of a recombination ocurring (a value of 0.9 is suggested) and $NP \geq 4$, the *population size*.

5.3.3 Application to gait optimization

In order to apply this algorithm to the gait optimization problem, a simple sinusoidal contoller was created for the OpenRAVE simulated modular robot. The oscillator parameters (A_i, O_i, ϕ_i for each oscillator, plus the frequency ($f = \frac{1}{T}$) of all of them) are encoded in the genotype of the individual. As the evolutionary optimization library used (ECF, section 3.1.3) did not support using several gentypes with different limits, the genotype was constraint to have values between $[-1, 1]$ and then later scaled to the suitables ranges for each parameter. In order to avoid collisions between limbs, the ranges for amplitude and offset where constraint to $[0^o, 60^o]$ for A_i and $[-15^o, 15^o]$ for O_i.

Symbol	Description	Constraint range
A_i	Amplitude of the ith oscillator	[0, 60] degrees
O_i	Offset of ith oscillator	[-15, 15] degrees
ϕ_i	Initial phase of ith oscillator	[0, 360] degrees
f	Frequency of all the oscillators $f = \frac{1}{T}$	[0, 1.5] hz

Table 5.2: Values used for gait optimization

For the evaluation of each individual the parameters are extracted from the genome, converted to values within the ranges provided in table 5.2, and set to the corresponding oscillators. Then, the simulation is run for 30s (simulation time), and the average speed is used as fitness value. Therefore, the cost function used for optimization is:

$$J(\vec{A},\vec{O},\vec{\Phi},f) = \frac{\text{Distance travelled}(m)}{\text{Evaluation time}(s)} \qquad \text{where: } \begin{cases} \vec{A} = [A_1\ A_2\ \cdots\ A_N] \\ \vec{O} = [O_1\ O_2\ \cdots\ O_N] \\ \vec{\Phi} = [\Phi_1\ \Phi_2\ \cdots\ \Phi_N] \end{cases} \tag{5.7}$$

Once a suitable gait has been discovered for a configuration, the parameters are stored in a table for that configuration, similar to a gait table, but with the three oscillator parameters (A_i, O_i, ϕ_i) assigned to their corresponding module ID. The frequency is also extracted and stored in another table, shared by all the configurations, and assigned to the ID of the configuration. This process is repeated for the three configurations to be evaluated on this thesis.

Chapter 6

Hormone Communications

6.1 Biological hormones

A hormone is a biochemical that a multicellular organism generates for regulation and communication between its different organs and tissues [5].

Hormones are generated by glands inside the living being body, and transported by the circulatory system to the receptors on the distinct organs and tissues. Some of them are soluble in water, and are delivered to their destination through the bloodstream, meanwhile other need to be bonded to carrier proteins in order to reach the receptors.

The use of hormones is the main method of communication between organs and tissues in multicellular organisms, and their function is to regulate distinct physiological functions, such as digestion, respiration, growth, circadian rythms (related to sleep), mood swings, inmune system control, hunger, sexual arousal, etc.

Organs and tissues have proteins that act as receptors for hormones, and when a hormone is bond to them they produce a signal that leads to the activation of certain genes that regulate protein synthesis. This synthesis may produce other hormones that trigger different reactions in other organs or tissues, such as the gland that generated the first hormone, working as a homeostatic negative feedback mechanism for controlling hormone generation. The hormone concentration required to produce a reaction on the receptor is very small, and big amounts of hormones in the organism usually leads to disorders, such as under/overgrowth.

An interesting feature of hormone communication is that a single type of hormone can trigger different reactions depending on the organ or tissue that receives them. For example, insulin, a hormone produced by the pancreas controls the intake of glucose from blood in liver and muscle, and the intake of lipids and synthesis of triglycerides in adipocytes, as well as other anabolic effects.

6.2 Concept of digital hormone

Digital hormones are a nature-inspired communication method first used by Wei-Min Shen, Behnam Salemi and Peter Will, researchers from the University of Southern California in 2000 [47]. They define a digital hormone as *"a signal, based on biological hormones, that is able to trigger different actions at different receivers, delegating the execution of those actions to the receiver subsystems"*. In this aspect they behave just as biological hormones, like insulin, that can produce different reactions, like the intake of glucose or lipids depending on the organ that receives them.

A control based on digital hormones is a control that lies between a master and a masterless control. The robot can be controlled entirely by the flow of hormones, in a masterless way, or with the help of the hormones, with any module assuming the role of master module whenever is required by the robot, depending on what method is more eficient at that certain moment.

The main properties of digital hormones, as described by Shen, Salemi and Will are the following ones:

1. **Hormones do not have a fixed destination, but float in a distributed system**.

   Since the modules do not posess a unique ID or address to identify them, hormones can not be sent to a particular module as it could be done with a standard communication protocol, such as TCP/IP. They are instead released in the system and relayed, modified or destroyed by the different nodes that receive the hormones.

   Even though hormones can not be sent to a concrete module, they can be sent to a module with a certain role, for example, a head module, a limb module or a spine module, and reach them by travelling through the distributed system, without any need for an ID or address, and even without knowing beforehand the route that the hormone must traverse to reach those modules.

2. **Hormones have a lifetime.**

   In order to prevent the hormones from circulating indefinitely along the distributed system, they have a limited lifetime.

   A hormone can be terminated in three possible ways: when they reach their destination, when their lifetime expires, or when they reach a module with no outlinks and, therefore, they can not be relayed again to any other module.

   If the modular robot were to have a configuration with loops, a hormone without lifetime could be trapped in those loops indefinitely, or returned to the module that generated it, producing undesired or unexpected effects to the robot.

3. **The same hormone can trigger different actions in different receiving sites.**

   Like their biological counterparts, digital hormones can trigger diffrent actions depending on the receptor they arrive to. These actions can be either hormone modification and relay, execution of local actions, or destruction of the hormone.

   For example, a hormone with a limb module as destination, generated at the spine module, can trigger a relay action if it arrives to a spine module, and later trigger a movement of the module joint when the same hormone arrives to a limb module.

In the original work of Shen, Salemi and Will, they define three kinds of hormones, that perform three different tasks: action specification, synchronization and dynamic grouping.

As mentioned in chapter 5, the locomotion of modular robots is almost always controlled by gait tables, containing the joint values for each DOF of each module in the robot at each step of the gait. Depending on the configuration and the number of modules, this table can become very large, and sometimes redundant. By using hormones for action specification, these gait tables can be simplified to a great extent. For example, for achieving linear locomotion in a caterpillar configuration, the module next to the first one sets the same value for its joints as the last value the first module had at the previous step, and so on. Instead of storing a gait table for all the different modules, we can just store the values for the joints of the first module, and then propagate them along the caterpillar using hormones. This method also allows to add new modules to the configuration dynamically, with no need of modifying the gait table to add any new entry.

Synchronization is a inherent problem of a distributed system, in which many machines coexist with their own clocks and local times that have to colaborate. This problem is more evident in modular robotics, as the different modules need to be synchronized in order to generate a suitable gait for its locomotion.

In a distributed system with a master node, synchronization usually involves a high communication cost, since a part of the limited bandwidth is consumed in synchronization messages. On the other hand, masterless control often makes the unrealistic assumption that all modules' internal clocks are synchronized. As hormones can wait at a module until the occurrence of a certain event, such as that all local actions are finished, they can be employed as synchronization tokens.

For instance, in order to synchronize the steps of a caterpillar configuration, a synchronization hormone can be defined, that is sent to the next module when all the local actions have been performed. This method ensures that all modules have completed the tasks of each step before relaying the hormone to the next module. When the last module receives this hormone and finishes its job, it can send back a hormone that can be used by the head module to generate another synchronization hormone, starting again the process.

6.3 Hormones in Hormodular

Modular robots in this thesis achieve locomotion through a set of sinusoidal oscillators with a certain amplitude (A_i), offset (O_i) and phase (ϕ_i), and a global oscillation period (T_{osc}) shared by all the modules. The optimal values for those oscillator parameters are obtained using evolutionary optimization algorithms, maximizing the point-to-point distance travelled by the robot during a certain evaluation time, that is, the average speed of the locomotion gait.

Those values are later set in gait tables, one per configuration. Therefore, each module needs to know what is its current function, based on their location inside the configuration, as well as their global configuration in order to select the appropriate parameters in the gait table corresponding to that configuration.

For that purpose the modules will be using digital hormones as their communication method because of the characteristics mentioned in the previous sections. This will also allow us to implement a homogeneous controller, identical for all the modules, reducing the complexity of the system, and making it much easier to maintain, control and debug.

The algorithm for module function and configuration discovery is based on three different types of hormone: a "Ping" hormone, in charge of the local configuration discovery and "Leg" and "Head" hormones, whose function is the global configuration discovery and communication. The next sections will describe these hormones used in the robot, as well as the communication protocol based on digital hormones, used to calculate the IDs required to select the correct gait table and oscillator parameters A_i, O_i, ϕ_i and T_{osc} from that gait tables.

6.3.1 Structure of a hormone

The structure of the hormones used in this thesis is the following:

```
class Hormone
{
	int type; //-- Posible values: PING_HORMONE, LEG_HORMONE, HEAD_HORMONE
	int sourceConnector; //-- Possible values: 0, 1, 2, 3
	string data;
};
```

(This is a simplified implementation to show the hormone general structure. More details about the actual implementation on the Hormone.hpp file of project Hormodular.)

The variable **type** defines the type of hormone, that can be either a "Ping" hormone, a "Leg" hormone or a "Head" hormone. These three types are explained in section 6.3.2.

The variable **sourceConnector** stores information about the connector that sent the hormone. This information is encoded as follows: Front connector: 0, Right connector: 1, Back connector: 2, Left connector: 3.

The variable **data** stores all the remaining information that might be needed, stored as a string:

- "Ping" hormones store in **data** the local orientation of the sender module. For a sender module with roll=90º, pitch=180º and yaw=270º, **data** would contain the string: "90 180 270".
- "Leg" hormones do not need to store any extra information on **data**.
- "Head" hormones store two values: the first one is the ID of the configuration (e.g. 1 for "Tripod" configuration) and the second one is a integer value used to discriminate the ID of each of the "Leg" modules.

6.3.2 Types of hormones

The three types of hormones used in this work differ from those three types defined by Shen, Salemi and Will in their publications, as they are used for discovering the function of each module in the locomotion gait, based on the general configuration and on the position inside that configuration to later select the oscillation parameters. These hormones are named "Ping" hormones, "Leg" hormones and "Head" hormones, and will be described in this section.

"Ping" hormones

"Ping" hormones are used by the modules to discover their orientation with respect to the neighbor modules, as well as how are they connected between them. They are also used to know which connectors are active, that being the reason we have named them "Ping" hormones.

These hormones are generated by all modules and have a very short range, they only travel to the nearest neighbors of the module. Apart from the information about the connector that sent them, they include the initial orientation of the sender module in their **data** field.

Section 6.3.3 explains in detail how the "Ping" hormones are used to calculate the local ID of the module.

"Leg" hormones

"Leg" hormones are generated at the "leg" modules, and are used to recognize the global configuration of the modular robot. "Leg" modules are those who are connected to just one module, and in all the three configuration correspond to the modules in the extreme of the limbs.

"Leg" hormones travel through the robot body until they arrive to the "head" module, which then discovers the current configuration from the amount of "Leg" hormone received. This process is explained in detail in section 6.3.3.

"Head" hormones

"Head" hormones are generated by the "head" modules, the module which receives "head" hormones at all its active connectors.

They flow in the opposite direction than the "leg" hormones, from "head" to "leg" modules and accomplish a dual function: they communicate to the rest of the modules the current global configuration discovered by the "head" module, and help the "leg" modules to distiguish among them, something that cannot be done only with the neighbors info (for the considered configurations).

The complete explanation of how these hormones are used can be found on section 6.3.3.

6.3.3 Hormone communication algorithm

This section explains the hormone conmmunication algorithm used by the modules to know which values they have to select from the gait table in order to achieve the optimal locomotion gait previously obtained through evolutionary algorithms. This algorithm has three main parts, one per type of hormone: local topology discovery, global configuration discovery and global configuration communication and leg discrimination.

Local topology discovery

For local topology discovery, i.e., discovering how a module is connected to its neighbor modules, the modules use "Ping" hormones. The procedure is as follows:

Each module sends "Ping" hormones through all its connectors. These hormones have information about the connector used to send them, as well as the local orientation of the sender module, obtained from the Inertial Measurement Unit (IMU) at the robot startup.

Only the connectors that have other modules attached to them (i.e. active connectors) will succeed in sending and receiving the hormones.

Each T_{com} seconds, the module looks for "Ping" hormones at each of its connector input buffers, and calculates the module ID of each module as described in section 5.1.4.

Let us consider, for instance, two modules with the configuration of figure 6.1, in which the front connector of module 0 is connected to the back connector of module 1, and vice versa.

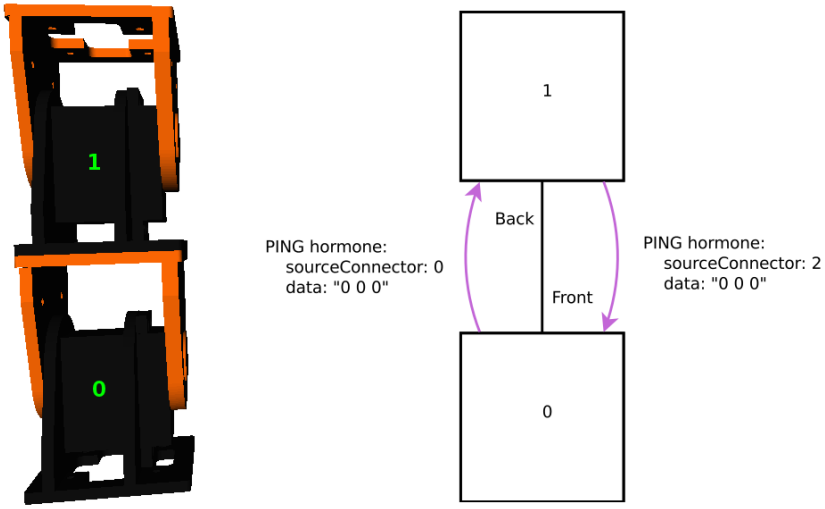


Figure 6.1: Local topology discovery in a two module configuration

For the discovery of the local topology, the module 0 would send hormones from all its connectors but, as only the front connector is attached to other module, only the hormones on the output buffer of the front connector would be delivered to module 1. The same is true for module 1, only the hormones in the back connector are able to arrive to other modules.

As the orientation of both modules is the same (roll=0º, pitch=0º and yaw=0º), the **data** field in the hormones generated by both modules would be also the same: "0 0 0".

Each T_{com} seconds, module 0 will look for "Ping" hormones at its input connector buffers, finding only the hormone sent by module 1. Using the information about which connector did receive the hormone (the front connector, encoded as '0'), which connector sent the hormone (the back connector, encoded as '2') and which was the remote orientation, (0º, 0º, 0º), the module can then identify its local topology, and calculate its ID:

$$ID = (x_0^0 \cdot 4^0 + x_1^0 \cdot 4^1) \cdot 17^0 + (x_0^1 \cdot 4^0 + x_1^1 \cdot 4^1) \cdot 17^1 + (x_0^2 \cdot 4^0 + x_1^2 \cdot 4^1) \cdot 17^2 + (x_0^3 \cdot 4^0 + x_1^3 \cdot 4^1) \cdot 17^3$$

$$ID = (2 \cdot 4^0 + 0 \cdot 4^1) \cdot 17^0 + 16 \cdot 17^1 + 16 \cdot 17^2 + 16 \cdot 17^3 = 83506$$

Module 1 will find only the hormone from module 0, with the receptor connector info (back connector, '2'), the sender connector info (front connector, '0') and the orientation of the module, (0º, 0º, 0º). The corresponding ID for this module will be:

$$ID = (x_0^0 \cdot 4^0 + x_1^0 \cdot 4^1) \cdot 17^0 + (x_0^1 \cdot 4^0 + x_1^1 \cdot 4^1) \cdot 17^1 + (x_0^2 \cdot 4^0 + x_1^2 \cdot 4^1) \cdot 17^2 + (x_0^3 \cdot 4^0 + x_1^3 \cdot 4^1) \cdot 17^3$$

$$ID = 16 \cdot 17^0 + 16 \cdot 17^1 + (0 \cdot 4^0 + 0 \cdot 4^1) \cdot 17^2 + 16 \cdot 17^3 = 78896$$

The same algorithm applies to modules with more than one module connected, such as the configuration on figure 6.2. In this case, the module 0 will receive three hormones from the neighbor modules: a hormone sent by module 1 from the back connector ('2') and received at the back connector ('2'), with a orientation of (0º, 90º, 0º), another sent by the module 2 from the back connector ('2'), received at the left connector ('3'), with a orientation of (270º, 0º, 90º) and a third one sent by the module 3 from its back connector ('2'), received at the right connector ('1'), with a orientation of (90º, 0º, 270º).

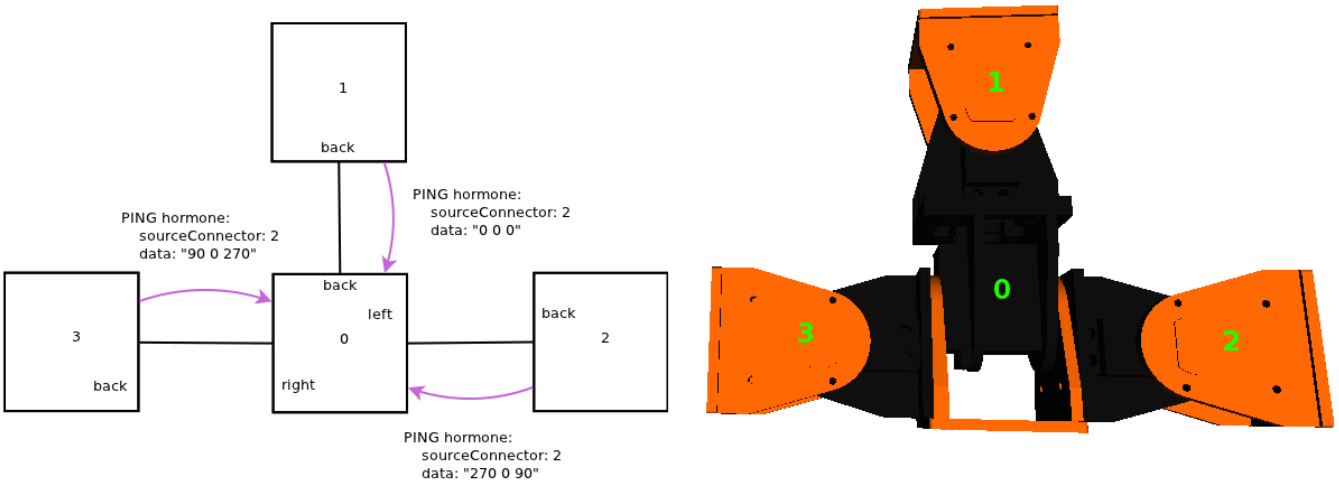


Figure 6.2: Local topology discovery in a four module configuration

With all this data the module 0 can calculate its ID:

$$ID = (x_0^0 \cdot 4^0 + x_1^0 \cdot 4^1) \cdot 17^0 + (x_0^1 \cdot 4^0 + x_1^1 \cdot 4^1) \cdot 17^1 + (x_0^2 \cdot 4^0 + x_1^2 \cdot 4^1) \cdot 17^2 + (x_0^3 \cdot 4^0 + x_1^3 \cdot 4^1) \cdot 17^3$$

$$ID = 16 \cdot 17^0 + (2 \cdot 4^0 + 3 \cdot 4^1) \cdot 17^1 + (2 \cdot 4^0 + 1 \cdot 4^1) \cdot 17^2 + (2 \cdot 4^0 + 1 \cdot 4^1) \cdot 17^3 = 31466$$

This calculated ID is unique for all modules in the three different configurations considered, except for the "leg" modules, which all share the same ID. For this reason we need more data in order to discriminate among the different "leg" modules and know their exact position inside the robot.

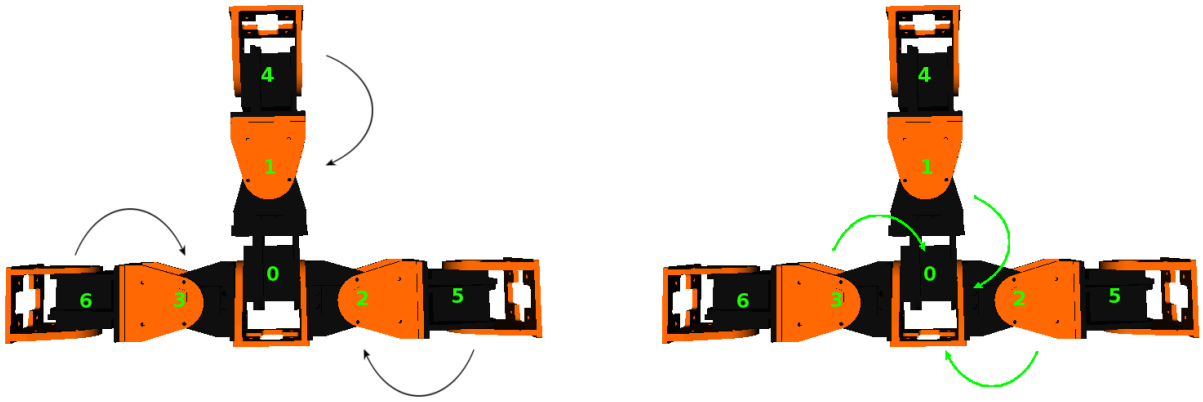
Global configuration discovery

"Leg" hormones are in charge of the global configuration discovery, i.e. recognize which is the general arrangement of the modules: a tripod configuration, a quadruped configuration, etc.

These hormones are generated at the "leg" modules (those who have only one active connection with other module) and they travel along the robot by the child connectors of each module (those who did not receive any "leg" hormone) until they arrive to the "head" module. The "head" module will then discover which is the global configuration, and will communicate it to all the modules.

The "head" module is defined for all the configurations as *"the module which receives "leg" hormones at all its active connectors"* and, therefore, does not have any child connector to relay the hormone to the next module. Depending on the amount of hormones that arrives to the "head" module, the robot will be in one configuration or another, and by counting the number of "leg" hormones received the "head" can discover this configuration. If the "head" module receives 2 hormones, the robot will be in the *MultiDof-11-2* configuration, if it receives 3 hormones it will be considered a *MultiDof-7-Tripod* configuration and if it receives 4 hormones it will be discoverd as a *MultiDof-9-Quad* configuration.

In figures 6.3, 6.4 and 6.5 we can observe this hormone flow throught the modular robot. Figure 6.3 represents the steps required by a hormone departing from a "leg" module to arrive to the "head" module in a *MultiDof-7-Tripod* configuration. After being generated, they are transmited from the modules 4, 5 and 6 to the modules 1, 2 and 3, respectively. In this figure we can also observe how at the second step the module 0 receives 3 "leg" hormones at the same time at all its active connectors, and thus it will become a "head" module, identifying the configuration by the number of hormones received.

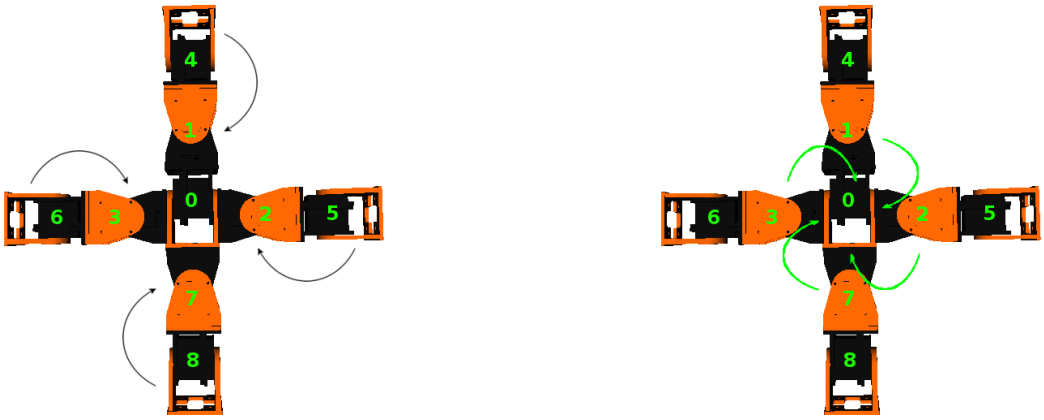


(a) Step 1 (b) Step 2

Figure 6.3: "Leg" hormone flow on the *MultiDof-7-Tripod* configuration

Figure 6.4 shows the "leg" hormone flow through the *MultiDof-9-Quad* configuration. This flow from the "leg" modules to the "head" module is completed in two steps as well, one from the modules 4, 5, 6 and 8 to the modules 1, 2, 3 and 7, and another from the modules 1, 2, 3 and 7 to the module 0, which performs the function of "head" module of this configuration. As it will receive 4 "leg" hormones in all its 4 active connectors, it will discover the module configuration to be the *MultiDof-9-Quad* configuration.

The hormone flow through the *MultiDof-11-2* configuration, shown in figure 6.5, is completed in one step more that the previous configurations, a total of three steps. In the first step, the modules 10, 9, 8 and 7 will send hormones to the modules 2, 3, 5 and 6. Those modules will relay the hormones to the "shoulder" modules, 0 and 4, and they will finally arrive to the module 1 from them, so that module 1 will become the "head" module of the configuration. Notice how "head" is a function that does not depend on the particular module, but in the configuration, and can be performed by any module when needed, as the controller is identical for all the modules. In this case, module 0 acts as "head" module for the *MultiDof-7-Tripod* and *MultiDof-9-Quad*, whereas the "head" of the *MultiDof-11-2* configuration is the module 1.



(a) Step 1 (b) Step 2

Figure 6.4: "Leg" hormone flow on the *MultiDof-9-Quad* configuration

Global configuration communication and leg discrimination

Once the "head" module has figured out which of the three possible configuration it belongs to, it will start the global configuration communication and leg discrimination algorithm.

As at this point only the "head" module knows what is the global configuration of the modular robot, so it has to communicate this information to the rest of the modules. For that purpose, it will start to generate "head" hormones that contain information about the current global configuration, and that will be propagated through all the modules until they arrive to the "leg" modules.

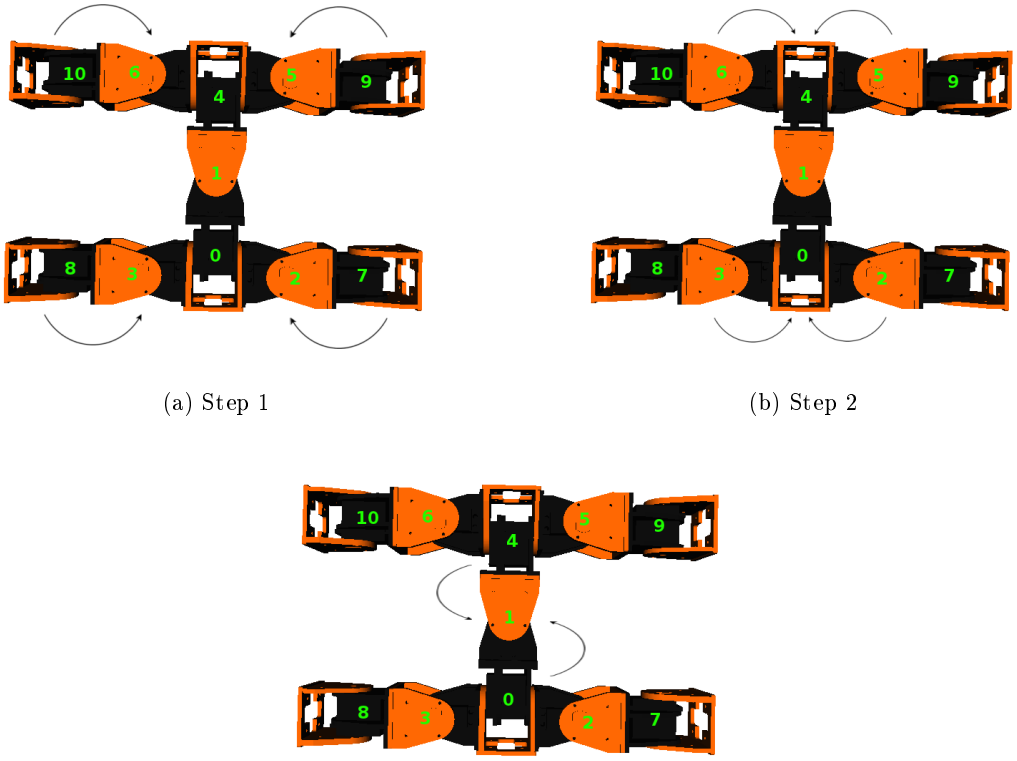
As all the "leg" modules have the same ID based on their connection with the neighboring modules, they also need some extra information in order to discriminate what is the position they occupy in the robot configuration. This information corresponds to an integer value from 0 to 3 that represents the connector of the "shoulder" module (i.e. the module that connects the limb with the body) that sent the hormone, which for the *MultiDof-7-Tripod* and *MultiDof-9-Quad* configurations correspond to the "head" module, but in the *MultiDof-11-2* configuration this function is performed by the modules placed at the end of the spine (0 and 4).

Other important aspect to notice is that the "leg" modules are continuously generating hormones each step, i.e. they do not wait until the "leg" hormone arrives to the "head" module , even though not all the hormones circulating in a given step are represented on the figures, to make them clearer. This way the robot can adapt faster to changes in its global configuration, as the "head" module is receiving continuously information about the limbs.

The flow of "head" hormones can be observed in figures 6.6 6.7 and 6.8. Figure 6.6 shows this flow for the *MultiDof-7-Tripod* configuration. In this configuration, the "head" module, module 0, will generate 3 "head" hormones, each one of the will have in its **data** field both the code of the configuration ('1' for *MultiDof-7-Tripod*) as well as another value telling the leg which receives this hormone who has the "head" connector that sent the hormone.

The procedure for the *MultiDof-9-Quad* configuration, shown in figure 6.7, is identical. The "head" module, also module 0, will generate in this case 4 "head" hormones, with the code of the configuration ('2' for *MultiDof-9-Quad*) and the connector that sent each hormone.

For the *MultiDof-11-2* configuration, shown in figure 6.8, the procedure changes slightly. In this case the modules connected to the limbs are module 0 and 4, so the "head" module will generate "head" hormones with just the information about the configuration ('0' for encoding the *MultiDof-11-2* configuration) on the first step. In the next step, shown in figure 6.8b, and before sending the hormone to the next modules, the modules 0 and 4 will add the extra info to distinguish between the different "leg" modules to the hormone.



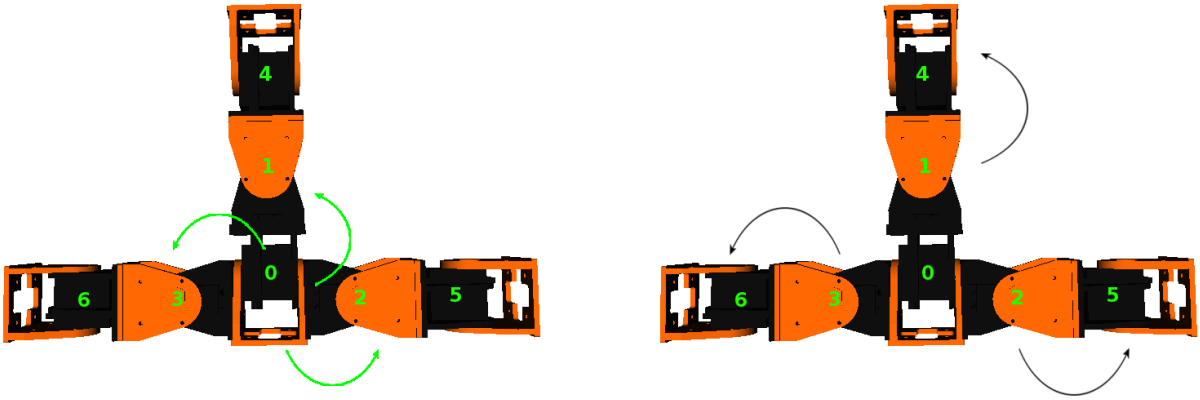
(a) Step 1

(b) Step 2

(c) Step 3

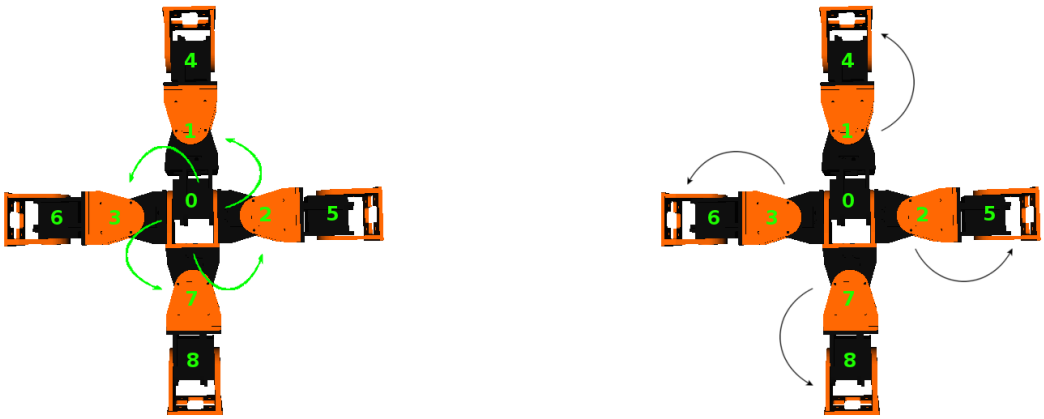
Figure 6.5: "Leg" hormone flow on the *MultiDof-11-2* configuration

As with the generation of "leg" hormones, the generation of "head" does not wait until the "head" arrives to the "leg" modules, but is continuously generated each time the "leg" hormones arrive to the "head" module, allowing a faster recognition of the possible changes in the global configuration.



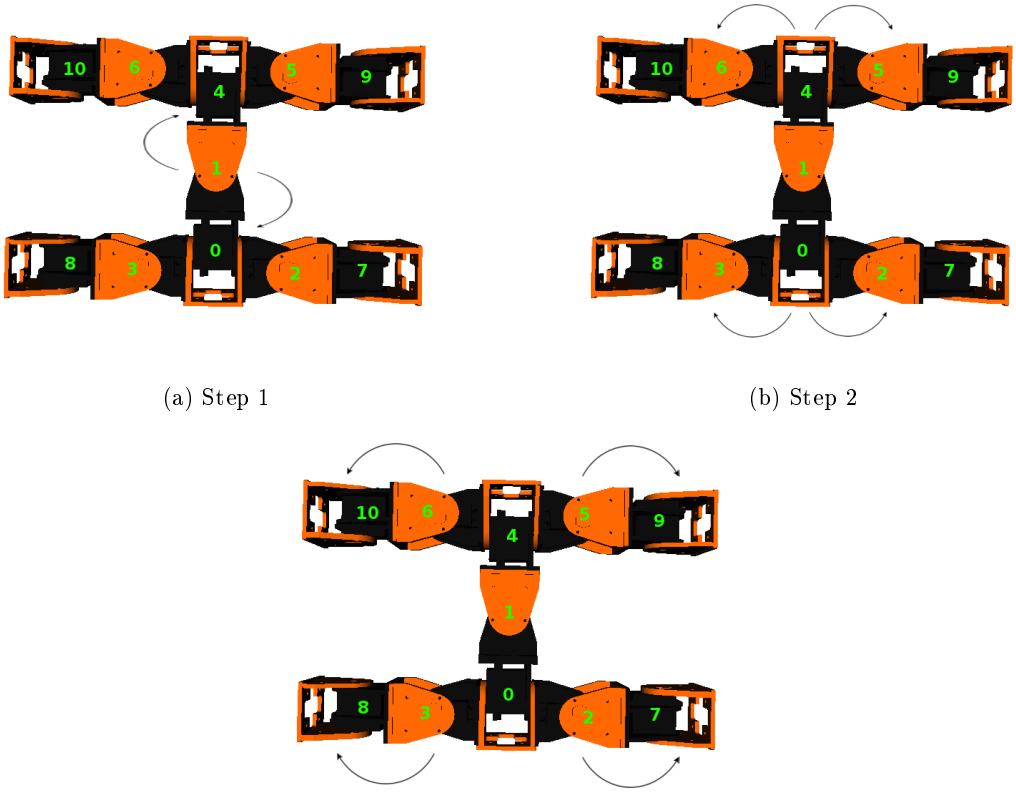
(a) Step 1 (b) Step 2

Figure 6.6: "Head" hormone flow on the *MultiDof-7-Tripod* configuration



(a) Step 1 (b) Step 2

Figure 6.7: "Head" hormone flow on the *MultiDof-9-Quad* configuration



(a) Step 1

(b) Step 2

(c) Step 3

Figure 6.8: "Head" hormone flow on the *MultiDof-11-2* configuration

Chapter 7

Results

In this chapter we will present the locomotion gaits obtained with the evolutionary optimization algorithm and we will analyze them for each of the configurations. We will also explain the results obtained after testing the digital hormone-based distributed controller in all the different configurations and its performance compared with our expectations.

7.1 Evolution results

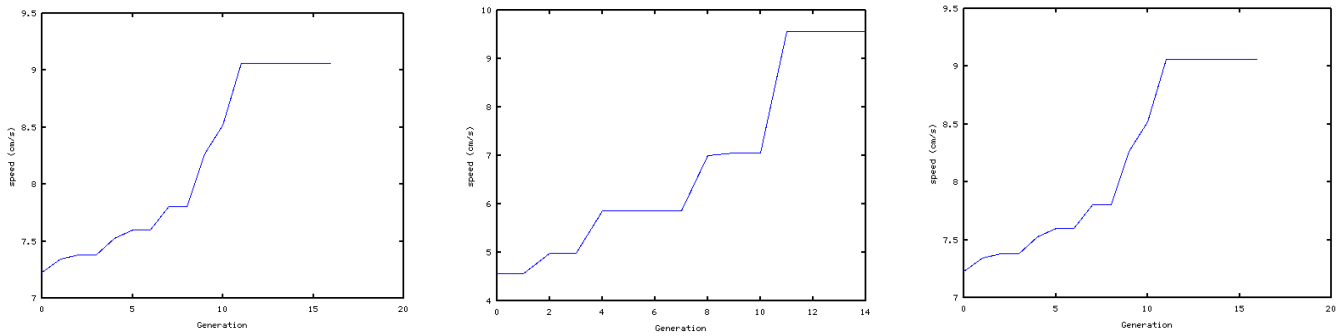
After developing the basic software framework for simulating the locomotion of the modular robot, the locomotion gaits for each of the three configurations were optimized using a Differential Evolution algorithm. The Differential Algorithm main parameters F (the scaling constant) and CR (crossover rate) were left with the ECF default values: 1 and 0.9, respectively.

For the rest of the parameters, we used a configuration with a single population of 40 individuals, in order to have a wider search space. The evaluation time of each locomotion gait was configured to 30s, with a simulation resolution of 250µs per step, and the oscillator parameters (A_i, O_i, ϕ_i, T) were restricted to avoid collisions between the different limbs of the modular robot. Table 7.1 shows a summary of the main parameters used for the evolution.

Parameter	Description	Value
F (Scaling constant)	Amplification of the differential variation	1
CR (Crossover rate)	Probability that a recombination occurs	0.90
Population size	Number of individuals on the population	40 individuals
Evaluation time	Simulation duration	30 s
Simulation step	Time of each step of the simulation	250 µs
A_{max}	Maximum amplitude allowed	80º for *MultiDof-7-tripod* , 60º for the others
O_{max}	Maximum offset (absolute value) allowed	45º for *MultiDof-7-tripod* , 15º for the others
ϕ_{max}	Maximum phase allowed	360º
f_{max}	Maximum frequency allowed	1.5 Hz

Table 7.1: Evolution main parameters

The evolution program was run for appoximately 16h, until stagnation of the best individual fitness value was reached. As evaluating 30s of simulation time results in about 80s / 100s of computing time, depending on the configuration, in those 16h that the program was run a small number of generation were produced (16 for *MultiDof-7-tripod* , 14 for *MultiDof-9-quad* and 11 for *MultiDof-11-2*). However, in that number of generations all the configurations had already reached a stagnation point. A plot the fitness value (the robot speed in cm/s) for the different configurations best individual is shown in figure 7.1 as a function of the number of generations. The best individual was found at generation 11 for *MultiDof-7-tripod* and *MultiDof-9-quad* , and at generation 8 for *MultiDof-11-2* , with values of speed around 9 cm/s.



(a) MultiDof-7-tripod (b) MultiDof-9-quad (c) MultiDof-11-2

Figure 7.1: Fitness value (robot speed in cm/s) of the best individual as a function of the number of generations

7.2 Analysis of resulting gaits

In this section we will analyze the locomotion gaits generated by the optimized sinusoidal oscillator parameters obtained with the differential evolution optimization, their speed and trajectory and their relationship with those parameters for the three configurations considered in this thesis.

7.2.1 MultiDof-7-tripod

Table 7.2 shows the optimal sinusoidal oscillator parameters for the MultiDof-7-tripod configuration.

Parameter	0	1	2	3	4	5	6
A_i	54.06	72.14	24.76	7.21	33.67	32.00	48.96
O_i	34.88	-31.77	36.03	30.05	39.82	24.64	-38.32
ϕ_i	36.34	206.12	133.29	62.80	112.13	191.15	234.73

Table 7.2: MultiDof-7-tripod oscillator parameters

The resulting gait of this configuration is one in which the robot expands and contracts to displace. The lateral limbs remain almost straight, rolling and using the 'leg' modules to advance. The tail modules have an offset towards the limb at the left connector of module 0, and with their movement they contribute to the forward locomotion.

As we can appreciate, the amplitude and offset values are similar for the two lateral limbs, except for module 3. Since the tail shoulder module (1) has a negative offset, a higher value for the amplitude of module 3 would result in a collision between this limb and the tail, so this low value makes sense.

The phase difference between the oscillators is very important for the coordination of the gait. In this case, modules 1, 5 and 6 have a very similar phase, and with their synchronized movement are the modules that contribute the most to the forward movement. Module 0 has a phase difference of approximately 180º with these modules, but as the module is placed upside down, the resulting movement is also in phase with the other modules, favoring the movement of the other modules.

The speed of the best individual with this configuration was 9.06 cm/s. Comparing this value with the speed of the best individual in the initial generation, composed by 30 individuals randomly initialized, that is 7.23 cm/s, results in an increase of 125%.

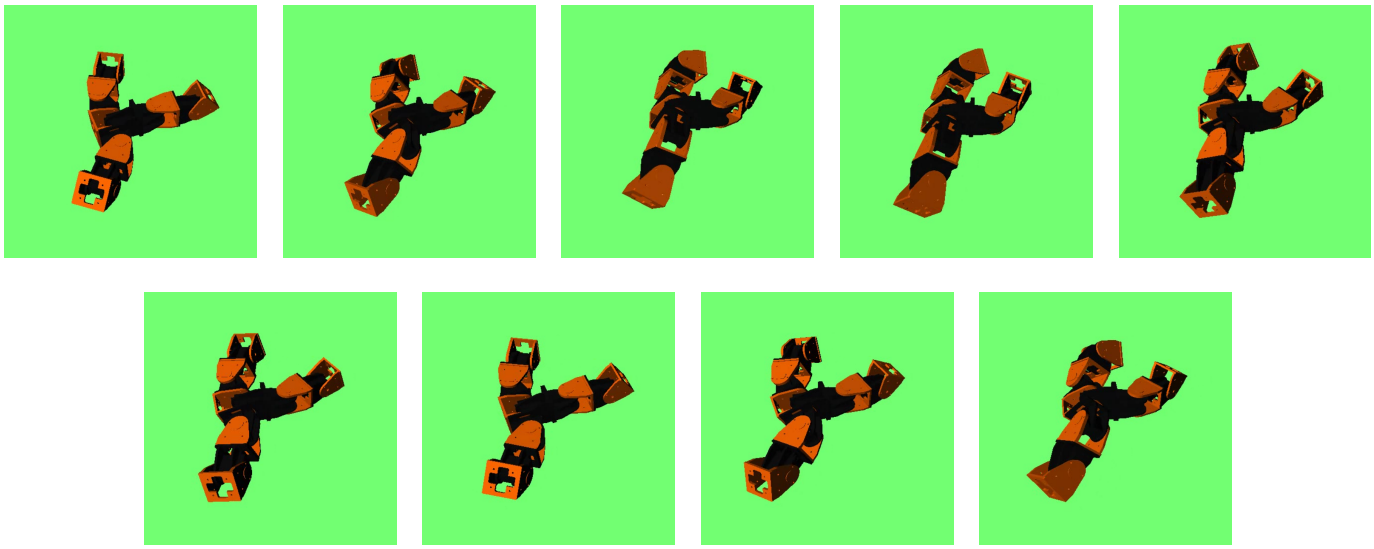


Figure 7.2: Sequence of the optimal gait obtained for the MultiDof-7-tripod configuration

The trajectory followed by this configuration is a straight line directed to the third quadrant, with small oscillations due to the oscillatory nature of the gait, and it is shown in figure 7.3.

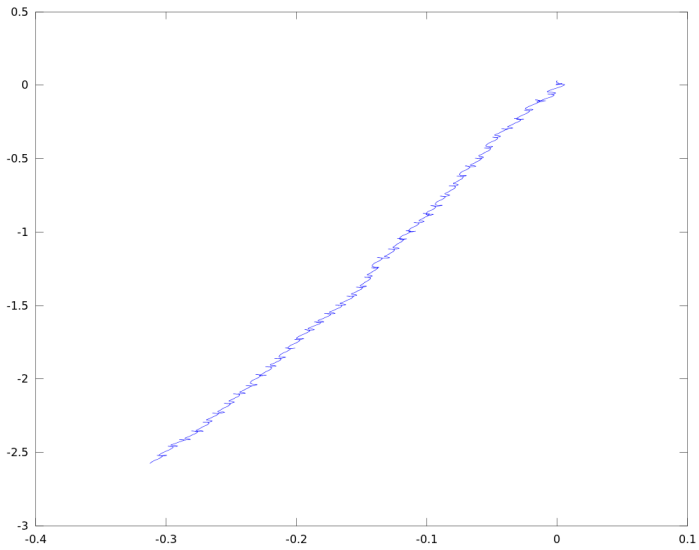


Figure 7.3: Trajectory followed by the MultiDof-7-tripod configuration

This gait was tested on the physical robot, obtaining the same gait with a similar performance. We tested the gait on different surfaces, and the best results were obtained on the surfaces with a higher friction coefficient, such as carpet and rubber mat. Other surfaces with a low friction coefficient with the robot, such as the floor, made the modular robot slip when performing its gait, resulting in a very small or null advance of the robot.

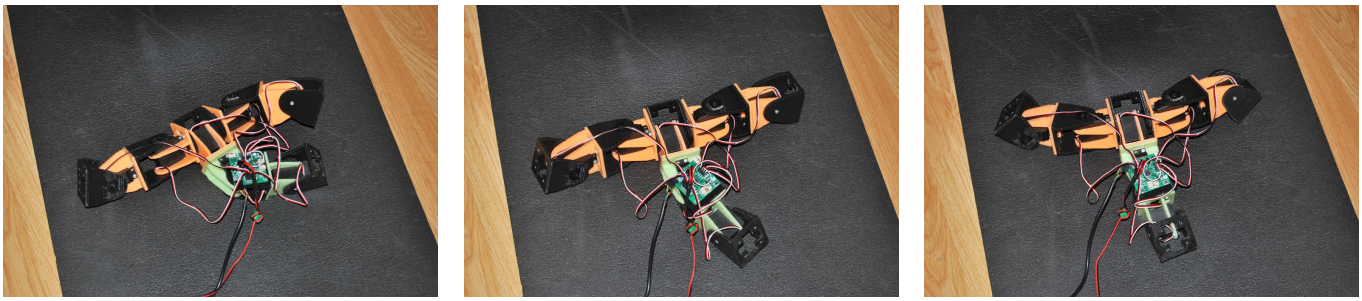


Figure 7.4: Testing gait on the modular robot, MultiDof-7-tripod configuration

7.2.2 MultiDof-9-quad

Table 7.3 shows the optimal sinusoidal oscillator parameters for the MultiDof-9-quad configuration.

Parameter	0	1	2	3	4	5	6	7	8
A_i	37.68	32.99	54.92	37.45	4.20	19.56	56.61	38.37	13.53
O_i	14.06	-4.18	-10.52	9.57	-14.64	1.64	-14.73	14.89	-2.05
ϕ_i	76.23	26.06	255.95	37.22	123.14	110.70	1.30	155.58	109.63

Table 7.3: MultiDof-9-quad oscillator parameters

This configuration moves in the direction of the limb connected to the back connector of the central module, composed of modules 1 and 4. The limbs connected to the front and back connectors of the central module have a movement that resembles a snake or worm robot, and this movement is helped with the other lateral limbs, which act as arms 'rowing' and contributing to the forward movement.

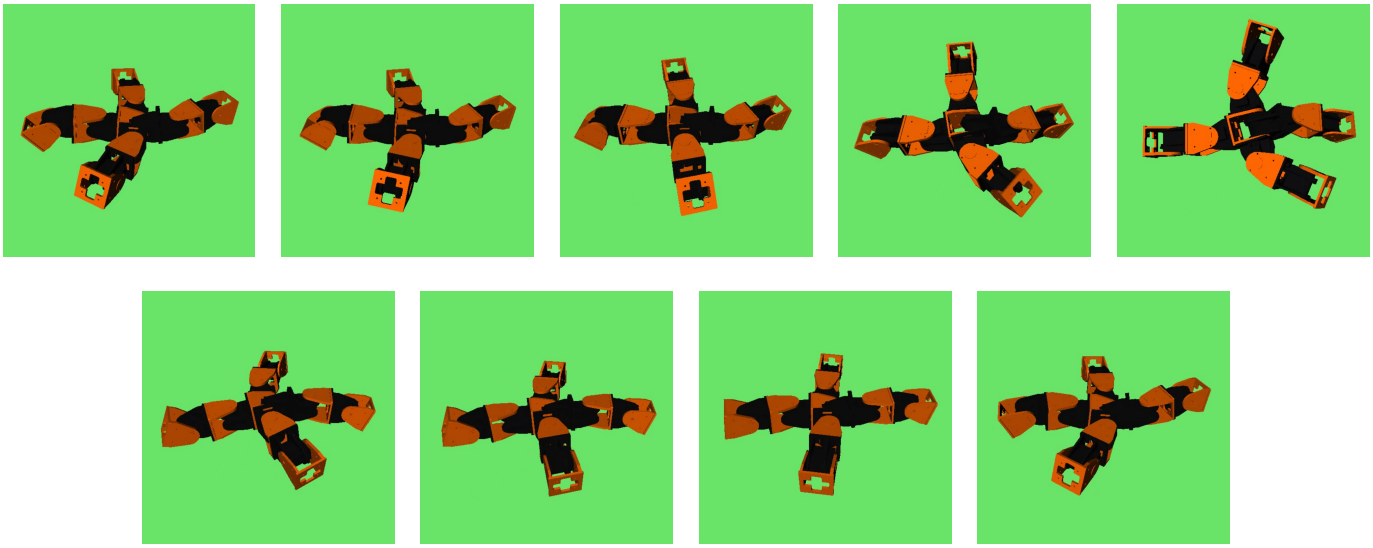


Figure 7.5: Sequence of the optimal gait obtained for the MultiDof-9-quad configuration

The amplitude and offset values are low due to the constaints imposed to avoid collisions between limbs. Modules 2 and 3, the shoulders of the lateral limbs, have a phase difference of 180°, but as they are placed as mirror images of each other, this results in a movement of these limbs towards the same direction in phase. The other limbs make this 'rowing'

movement more effective by pushing the robot forward when the lateral limbs are not in contact with the ground.

The speed of the best individual with this configuration was 9.57 cm/s. If we compare this value with the speed of the best individual from the 30 individuals randomly initialized of the first generation, 3.32 cm/s, results in an increase of 288%, almost 3 times better than the random solution.

This configuration gait follows a straight line almost parallel to the y axis, with small oscillations due to the oscillatory nature of the gait, which is shown in figure 7.6.

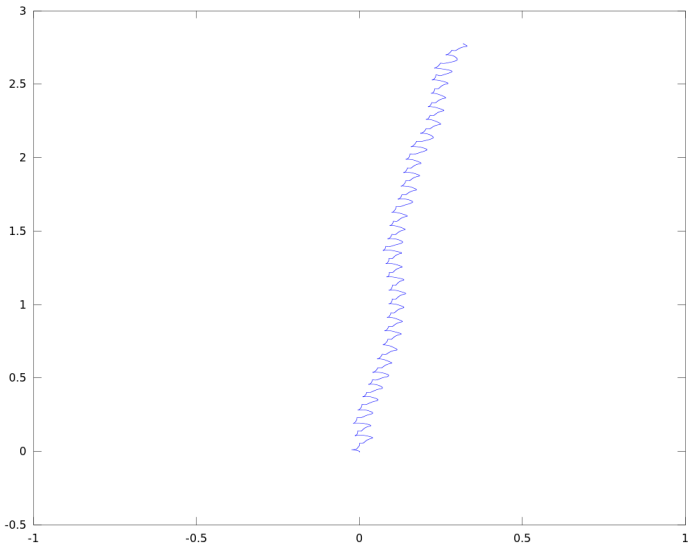


Figure 7.6: Trajectory followed by the MultiDof-9-quad configuration

Since this configuration is made of 9 modules, and the modular robot hardware current limit is 8 modules, this gait could not be tested on the real world. Testing this gait requires the use of 2 interconnected control boards, feature that is left as future work. However, the expected results are similar to the previous configuration: a speed close to the values obtained in the simulations, and a better performance over surfaces with a high friction coefficient between them and the robot.

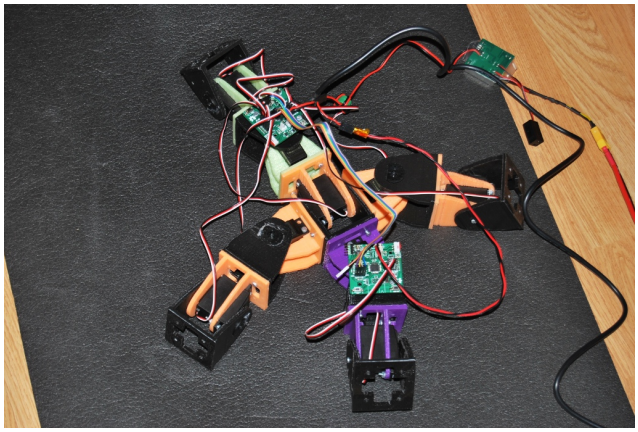


Figure 7.7: Modular robot configured as MultiDof-9-quad

7.2.3 MultiDof-11-2

Table 7.4 shows the optimal sinusoidal oscillator parameters for the MultiDof-11-2 configuration.

Parameter	0	1	2	3	4	5	6	7	8	9	10
A_i	33.66	28.21	37.05	49.91	50.91	16.83	46.78	20.17	9.82	26.69	29.42
O_i	2.91	4.19	1.63	-3.04	-8.53	-3.82	-9.07	-5.51	-2.93	3.14	-13.14
ϕ_i	108.12	342.78	112.62	218.17	326.64	207.69	306.48	239.12	218.58	120.80	140.99

Table 7.4: MultiDof-11-2 oscillator parameters

This configuration has a gait similar to the typical quadruped gait, with the limbs in the same diagonal moving in phase and in opposite phase as the limbs in the other diagonal to generate a forward movement towards the module 0 direction.

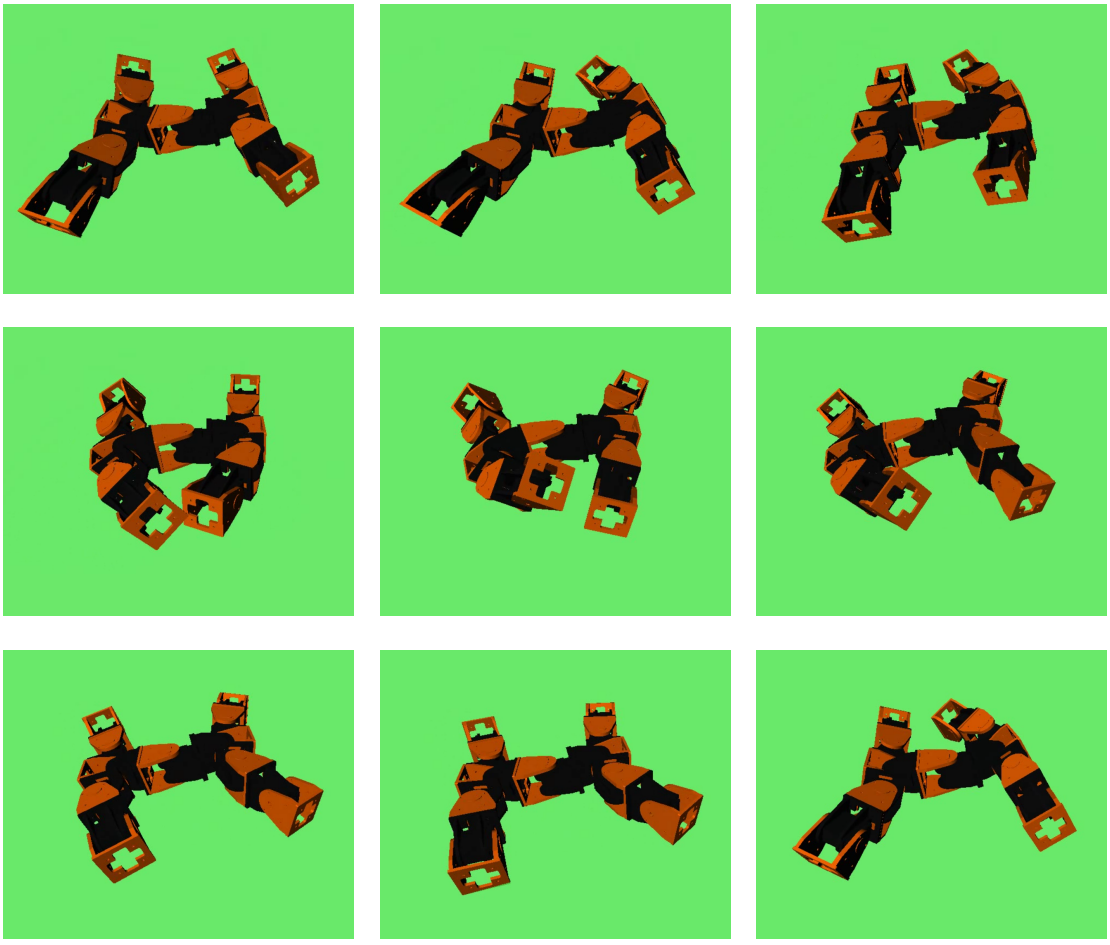


Figure 7.8: Sequence of the optimal gait obtained for the MultiDof-11-2 configuration

As with the previous configuration, the amplitude and offset values are low because they were constrained in order to avoid collisions between limbs. Looking at the phase difference between modules 2 and 6, and 3 and 5, we can observe that it is around 180º, but as the modules are mirrored, this results in a movement in phase in the same direction. We can also observe intra-limb coordination, with a phase difference of approximately 120º between the 'leg' module and the 'shoulder' module of each limb.

The best individual with this configuration had a speed of 8.53 cm/s and the best individual among the 30 individuals randomly initialized that compose the initial population was 1.52 cm/s. Comparing them we obtain an increase of 562%, more than 5 times the speed of the best random individual, which indicates that optimization algorithm generates better gaits than the ones that could be obtained randomly.

The trajectory followed by this configuration is also a straight line directed to the third quadrant, with small smooth oscillations due to the oscillatory nature of the gait, and it is shown in figure 7.9.

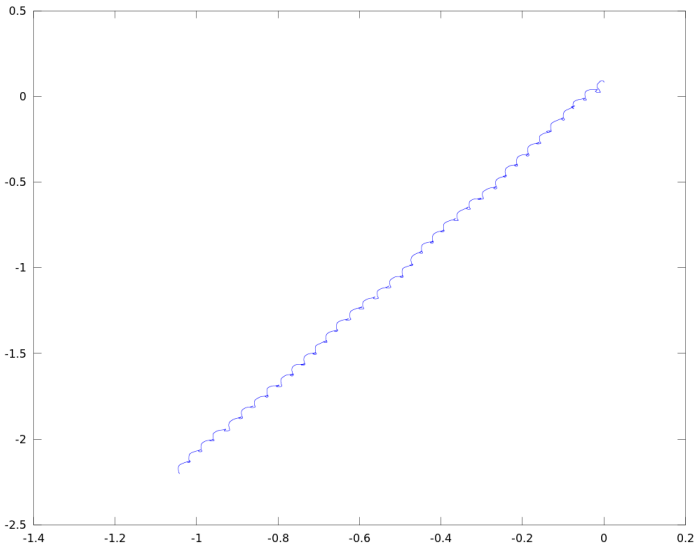


Figure 7.9: Trajectory followed by the MultiDof-11-2 configuration

This configuration requires 11 modules and therefore, as the previous configuration, it could not be tested on the real world with the current robotic platform. As with the MultiDof-9-quad configuration, implementing the communication between the 2 control boards, and testing gaits on this configuration is left as future work.

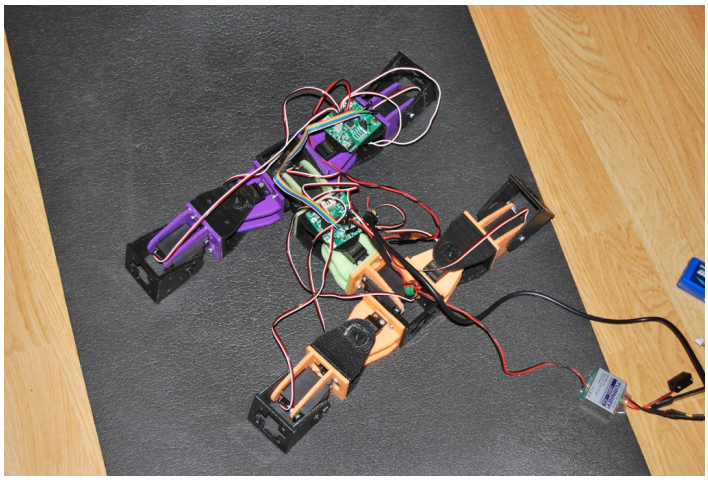


Figure 7.10: Trajectory followed by the MultiDof-9-quad configuration

7.3 Analysis of hormone-based communication protocol

Finally, we will analyze the performance of the hormone-based distributed communication protocol, checking if the controller is able to discover the global configuration of the modular robot as well as the current function of the module inside that configuration. The behavior of the modular robot with this controller will be compared to the behavior of the modular robot with only sinusoidal oscillators with the optimal parameters to check if the speed of the robot decreases when using the hormone-based controller.

After running the evolutionary optimization algorithm, the optimal oscillator parameters obtained were extracted and set on the parameter tables, ready to be loaded by the module controller. Once the preparations were finished, the contoller was tested for all the different configurations.

Once a brief period of time has passed, in which the robot is finding its current configuration and therefore the resulting gait is chaotic, each module discovers the global configuration of the robot and its role inside that configuration, and sets its oscillator parameters to the corresponding ones according to the values stored in the parameter tables, achieving the optimal locomotion gait for the current configuration.

This period of time depends on the communication period being used (i.e. how frequently does the module exchange hormones with the neighboring modules) and the number of communication steps required for a 'leg' hormone to travel to the 'head' module and for the 'head' hormone to travel back to the 'leg' module, as explained on section 6.3.3. This number of steps is different for the three configurations, and it is equal to 6 steps for the *MultiDof-11-2* configuration and 4 steps for the *MultiDof-7-tripod* and *MultiDof-9-quad* configurations. The value for the communication period (T_c) used for testing the controller was 100 ms, so the initial delay in discovery the configuration and function, and selecting the correct parameters is 400 ms for the *MultiDof-7-tripod* and *MultiDof-9-quad* configurations, and 600 ms for the *MultiDof-11-2* configuration.

Since the communication period is very large compared to the joint update period (100 ms vs 250 µs), this communication does not affect the performance of the robot, which after finding the appropiate gait achieves the same speed as if only the gait is evaluated without controller, by assigning the appropiate parameters to the oscillators by hand.

Even though reconfiguration is not possible currently, due to limitations in the software framework and in the hardware modules, since the modular robot is able to select gaits correctly for all the three configurations considered with the exact same controller, we can induce that this controller will also work with a reconfiguring modular robot. In that case, the maximum delay for the detection of the new configuration would be the same as the one calculated before for the configurations starting from the initial state, 400 ms for the *MultiDof-7-tripod* and *MultiDof-9-quad* configurations, and 600 ms for the *MultiDof-11-2* configuration.

Chapter 8

Conclusions and Future Work

This final chapter includes a brief summary of all the topics covered on this thesis, highlighting the most important aspects of it, as well as some suggestions of topics to be developed as future work.

8.1 Conclusions

We began the thesis presenting the main topic, modular robots, and their main terminology and applications. We also explained what are the most frequent problems found when working with modular robots: their high number of degrees of freedom, that make very difficult the design of efficient gaits with analytical methods; their distributed nature, that require a distributed controller, in which each module only has information about its neighbours and control of the whole modular robot emerges from the sum of all the individual decisions; and the high cost of modular robots, due to the large number of modules required and the high cost of each of them.

We presented our main objectives, that correspond with solutions to those main problems. The first one is to obtain optimal locomotion gaits with a bio-inspired approach, using sinusoidal oscillators and optimizing their main parameters (amplitude, offset, phase and frequency) with a evolutionary optimization algorithm called Differential Evolution, which is based on biological evolution. The second one is to design a homogeneous distributed controller that is able to discover the current global configuration of the modular robot and the role of the module inside that configuration using digital hormones. The last one is to design a cheap modular robotic platform to test the gaits and controller obtained and validate them in a real robot, apart from the simulated one.

Once these objectives were presented, we reviewed the state of the art in the topics covered by the main problems. We introduced the main existing modules and their features, including the module Y1 designed by Juan Gonzalez-Gomez, in which we based the design of our module and electronic control board. We also reviewed the different techniques used for locomotion in modular robots, such as gait tables, CPGs and sinusoidal oscillators. Finally, we introduced the problem of coordination in modular robotics, and the existing solutions in the literature.

All the software framework implemented for modular robots, named *'Hormodular'*, and the methodology used to develop it, called 'Test-Driven Development' were presented in the next chapter. This methodology consists in developing code with the help of unit tests, that test the requirements of the software to be implemented, and help the programmer to develop code faster, in a more maintainable way, and writing the minimum amount of code necessary to accomplish the requirements of the project. We also presented the different software dependencies of the project, a detailed explanation of each class belonging to the 'Hormodular' framework and the instructions to download and run the software, which has been released publicly under a open source license.

Afterwards, we presented the hardware platform developed to test the locomotion gaits and the distributed controller. We explained the main features of the Y1 module and their derivatives, as well as the Skymega board, in which we based our work. We introduced the drawbacks found on these platforms and explain how we solved them on our designs. Our module, the REPY-2, can be produced at a low cost and is easy to manufacture using a 3D printer. Improves the previous design with a symmetrical, more resistent design, that allows not only linear configurations, but also 2D configurations.

Detailed instructions to assemble this module were also included. The electronic board designed, the SkymegaSMD, solves some flaws of the previous Skymega design by using SMD components, which allowed us to include a 5V regulator, split the power supply of the servos and improve the circuit layout, reducing the electrical noise. We also talked about the other components required to assemble the modular robot, such as the batteries or the USB/serial adapter, and about the firmware implemented to control the robot from the computer.

The next topic discussed was the modular robot configuration and gaits, in which we introduced the different types of configurations allowed by the REPY-2 modules design, as well as our method for describing these different configurations. This method encodes the connections of one module in a ID that represents them, using for that purpose the IDs of the connectors involved in the connection, as well as the relative rotation between them. We also explain the sinusoidal oscillators, selected to achieve the locomotion and coordination of the modular robot by using a simplified version of the CPG model, which is more complex and demands more computing power. Finally, we introduced the Differential Evolution algorithm that we used to find the parameters of the sinusoidal oscillators which yield optimal locomotion gaits for the three configurations considered in this work.

Next, we introduce the concept of digital hormone, comparing them to the biological hormones, as a message that does not have a fixed destination but floats on the distributed system, has a lifetime, and triggers different actions depending on its receiver. We explain the structure of our digital hormones, as well as the hormone communication protocol. This protocol, based in three types of hormone ('ping', 'leg' and 'head') is used by the modules to discover the global configuration of the modular robot, as well as the role of each module inside that configuration. With this information, each module can select the appropiate parameters for their sinusoidal oscillators, from the ones learned with the evolutionary algorithms.

To conclude, we present the results obtained when testing the locomotion gaits and the homone-based controller. The resulting gaits are then analyzed, as well as their trajectory and speed, and their relationship to the sinusoidal oscillator parameters used to produce them. We observe how these parameters have generated stable and fast gaits, that can be reproduced in the real modular robot with a similar performance for surfaces with a friction coefficient high enough. Finally, we tested the hormone controller for the three configurations and we checked how after a short period of time the modules are able to discover their global configuration and their role inside that configuration, and they are able to select the appropiate parameters that have been previously found with the differential evolution optimization.

This way we checked that all the objectives proposed for the thesis have been accomplished: we found optimal gaits for the three configurations using sinusoidal oscillators whose parameters were optimized through differential evolution; we designed a homogeneous distributed controller based on digital hormones that is able to discover the modular robot global configuration as well as the function of the module inside that configuration, and select accordingly the parameters for the sinusoidal oscillators; and we developed a cheap modular robotic platform that was used to test our work, and that can be reused for other modular robotics researches.

8.2 Future work

In this section we will present some suggestions of possible improvements to the current work that could not be developed for this thesis due to limitations in time or resources, and that could be developed as future work.

1. **Enable MultiDof-9-quad and MultiDof-11-2 configurations on hardware plaftform.**
   Due to hardware limitations, the current platform only allows using up to 8 servos with a single SkymegaSMD board, which only allows us to test the gaits on the MultiDof-7-tripod configuration. For the remaining configurations, that count with 9 and 11 servos respectively, two SkymegaSMD boards have to be used.

   These boards have to be connected using the I2C bus, and the current firmware has to be extended to allow this communication through the I2C bus, and to select from the computer controller which board is to receive the joint values to be set to the servos.

2. **Development of more advanced modules.**
   The current platform is cheap and useful for testing different gaits and controllers on a real modular robot, but it is also very limited for other topics related to modular robotics. If a distributed controller or communications between modules have to be implemented, they mush be emulated on the computer, since the current robot has a central controller that can only receive the joint position values and set them on the different servos. Reconfiguration of the modular robot can only be achieved manually and, since the connectors use screws, this manual reconfiguration is very slow and tedious.

   One possible improvement to the current work would be to develop a better modular robotic platform. This platform would need to have the control electronics, communications and power on each of the modules, so that the algoritms developed for them can be tested on the actual modules without the need for a computer. It would also have to feature a new connector that allows self-reconfiguration or, at least, that eases the manual reconfiguration process providing a simple lock/unlock mechanism.

3. **Improvement of the current hormone-based communication protocol.**
   The current hormone protocol works correctly with the three proposed configurations, but it relies too much on the particular aspects of each of them and, if a new configuration is to be added to the controller, it is not trivial to modify it to add the new configuration and its corresponding parameters to the controller.

   The hormone protocol could be improved to a more generic one, in which locomotion gaits for new configurations can be added by adding the corresponding new parameter tables to the controller, and the hormone protocol can discover this new configuration and use its parameters without further modifications.

4. **Add support for reconfiguration to the Hormodular framework.**
   Hormodular currently does not support changing from one configuration to other one while the simulation is running, that being the reason why we only tested the hormone controller on each of the configurations individually.

   In order to ensure that the homone controller also works when the robot configuration has changed, and to allow research related to self-reconfiguration on modular robots, this support to reconfiguration should be added to the software framework and simulator.

5. **Add support for concurrent execution of controllers.**
   To simplify the development of the controllers, the current version of Hormodular executes them sequentially, emulating the concurrency that it would exist if run on the different modules. This simplifies the process of developing and testing a controller, but the resulting controller is not realistic, and lacks some of the problems of actual distributed controllers, such as the need for synchronization and a robust communication protocol.

   If the controller of each module is run concurrently in simulation, all those aspects can be evaluated in conditions closer to the ones existing on the real life modular robot.

6. **Add sensors to the modules or to the modular robot.**
   In the current version of the controller, the sinusoidal oscillators are running using the optimized parameters in open loop, with any kind of sensorial feedback. Different kinds of sensors could be added to the modules, such as potentiometers or encoders to measure the actual joint position, inertial sensors (IMUs) to measure the movement of the module or IR / ultrasonic rangefinder to measure the distance to the possible obstacles the robot may face.

   Using the information received from the sensors, and integrating it to the hormone communication stream, it would be possible to develop a reactive controller to modify the modular robot gaits to avoid obstacles or adapt the gait to changes in its performance due to changes in the terrain conditions (changes in friction coefficient, slope, etc).

Appendices

Appendix A

Cost Estimation

A.1 Detailed cost estimation

Code	Units	Description	Quantity	Unit price (€)	Total Price (€)
1	unit	**Modular Robot**	1		209.62
1.1	unit	**Module REPY-2**	11	12.3	135.30
1.1.1	unit	**Upper part** Upper part of the REPY-2.0 module, 3D printed with PLA plastic. Includes manufacturing and preparation costs.	1	1.5	1.5
1.1.2	unit	**Lower part** Lower part of the REPY-2.0 module, 3D printed with PLA plastic. Includes manufacturing and preparation costs.	1	1.5	1.5
1.1.3	unit	**Servo Futaba 3003s** Futaba 3003s hobby RC servomotor.	1	9	9
1.1.4	unit	**M3x8mm screw** M3 screw, length 8mm	1	0.02	0.02
1.1.5	unit	**M3x10mm screw** M3 screw, length 10mm	4	0.02	0.08
1.1.6	unit	**M3 nut** M3 nut.	4	0.03	0.12
1.1.7	unit	**M3 washer** M3 washer.	4	0.02	0.08
				Subtotal:	12.3
1.2	unit	**Control board SkymegaSMD**	2	13.54	27.08
1.2.1	unit	**SkymegaSMD PCB** Two layer PCB manufactured at Seeedstudio, a low cost chinese manufacturer of PCB protypes. PCB includes green soldermask and white silkcreen on both PCB sides.	2	1.395	2.79
1.2.2	unit	**VLMS1300-GS08 Super Red Clear** LED Red 0603	2	0.064	0.128
1.2.3	unit	**LTST-C193TBKT-5A Blue 470nm** LED Blue 0603	2	0.08	0.16
1.2.4	unit	**LG Q971-KN-1 Green, 570nm** LED Green 0603	1	0.04	0.04
1.2.5	unit	**ATMEGA328P-AU** ATMEL microcontroller, 32KB In-system Flash	1	2.5	2.5

Code	Units	Description	Quantity	Unit price (€)	Total Price (€)
1.2.6	unit	**RR0816P-102-D** Resistor 1K 1/16W 0.5% 0603	5	0.064	0.32
1.2.7	unit	**CRCW060310K0FKEA** Resistor 10K 1/10W 1% 0603	3	0.016	0.048
1.2.8	unit	**C0603C104K3RACTU 25volts 0.1uF** Ceramic capacitor 0.1µF 0603	4	0.016	0.064
1.2.9	unit	**C0603C105Z8VACTU 10volts 1uF** Ceramic capacitor 1µF 0603	8	0.024	0.192
1.2.10	unit	**C0603C270J5GACTU 50volts 27pF** Ceramic capacitor 27 pF 0603	2	0.016	0.032
1.2.11	unit	**F931A106MAA 10volts 10uF** Tantalum capacitor 10µF 1206	2	0.112	0.224
1.2.12	unit	**7A-16.000MAAJ-T** 16.000MHz 30ppm SMD crystal.	1	0.752	0.752
1.2.13	unit	**FSMSM 3.5X6 SMT TACT** Tactile switch, SMD package.	2	0.231	0.462
1.2.14	unit	**MIC5205-5.0YM5 TR** 5V linear LDO voltage regulator, SMD package	1	0.5	0.5
1.2.15	unit	**MOLEX 22-27-2021** 2 pin MOLEX polarized power connector	2	0.235	0.87
1.2.16	unit	**EOZ 1K2 09.10201.02** 2 position switch, right angle, 2.54mm pin spacing	1	1.492	1.492
1.2.17	unit	**SEK-18 SV ML LP ANG29 06P PL2** 2.54mm 2x4 right angle shrouded header.	1	1.08	1.08
1.2.18	unit	**FCI 68004-236** 2.54mm 36 male straight pins strip	1	0.824	0.824
1.2.19	unit	**929835-01-36-RK** 2.54mm 36 male right angle pins strip	1	1.06	1.06
				Subtotal:	13.54

1.3	unit	**Other components**			47.24
1.3.1	unit	**USB-to-Serial converter cable** USB-to-Serial converter cable based on the FTDI FT232RL USB/serial chip to convert USB communications to TTL RS-232 signals.	1	14.73	14.73
1.3.2	unit	**ZIPPY Flightmax 2200mAh 3S1P 20C** LiPo Battery for RC vehicles, 3 cell (11.1V), 2200Ahm capacity, 20C discharge rate.	1	10.63	10.63
1.3.3	unit	**TURNIGY 8-15A UBEC for Lipoly** UBEC for RC vehicles, 8-15A output current, 5V/6V selectable output voltage, 6V-12.6V (2-3 cell LiPo) input voltage	1	11.22	11.22
1.3.4	unit	**JY-MCU Bluetooth Wireless Serial Port Module** Bluetooth wireless serial port module including a HC-06 Bluetooth transceiver for communication with a 5V TTL RS232 serial port.	1	8.66	8.66
1.3.5	unit	**M3x16mm screw** M3 screw, length 16mm	40	0.02	0.80
1.3.6	unit	**M3 nut** M3 nut.	40	0.03	1.20
				Subtotal:	47.24

Code	Units	Description	Quantity	Unit price (€)	Total Price (€)
2	unit	**Software (Hormodular framework)**	1		3,800
2.1	hours	**Developer work time**	190	20.00	3,800.00
3	unit	**Research costs**	1		6,200.00
3.1	hours	**Researcher work time**	310	20.00	6,200.00

A.2 Cost estimation summary

Code	Description	Cost (€)	% Total cost
1	Modular Robot	209.62	2.05
2	Software (Hormodular Framework)	3,800.00	37.22
3	Research costs	6,200.00	60.73
	Total	10,209.62	100

Appendix B

Time Distribution

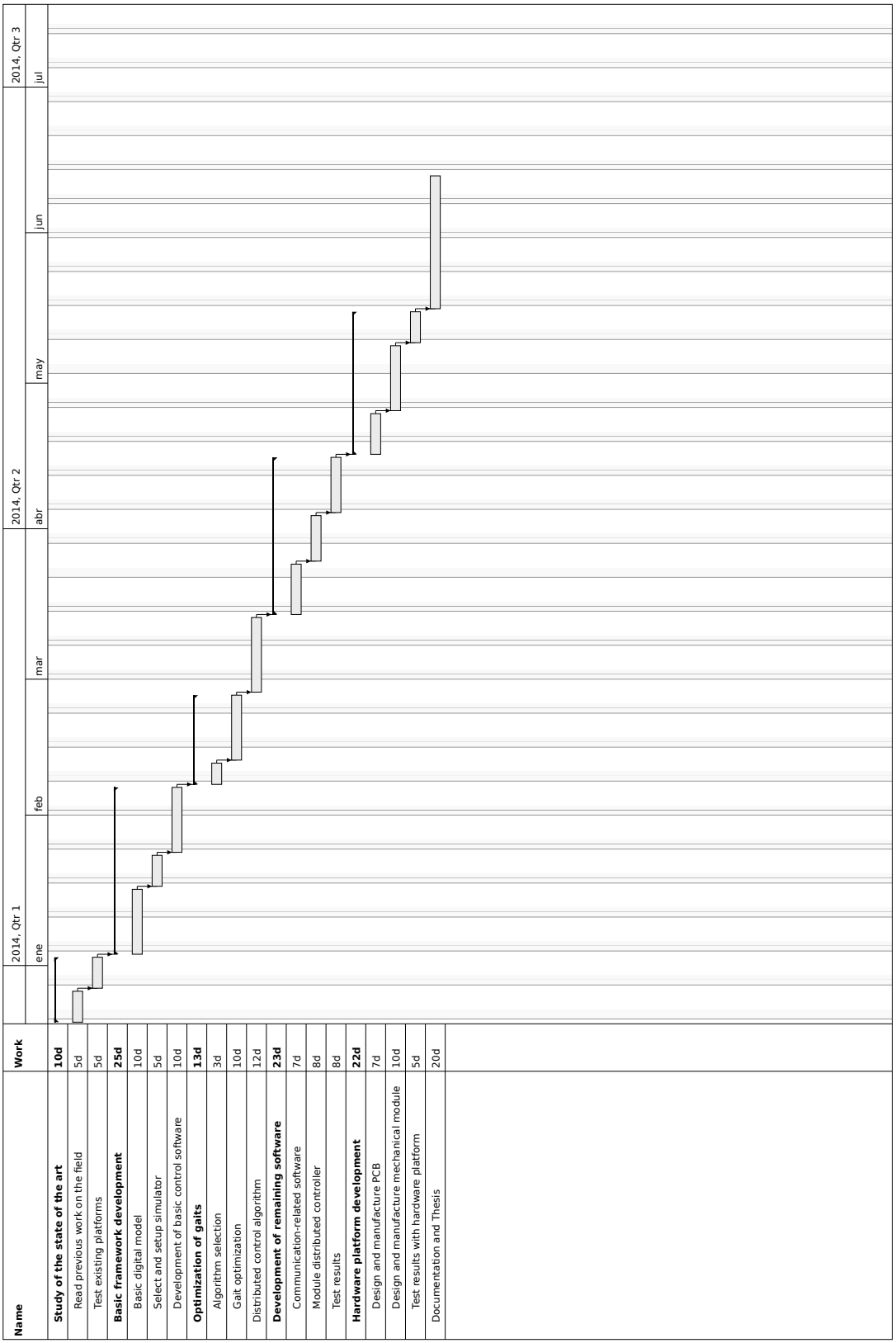
The project started on December 20th 2013, and was estimated to end on June 13th 2014. The estimated task planning during those 6 months is presented in the section B.1, and a Gantt diagram following that planning is included in section B.2.

The critical tasks are the gait optimization and design of the distributed control algorithm, since one cannot foresee the results, and an error in this tasks would suppose an extra workload. Because of that reason, extra time was assigned to those tasks as a buffer to take into accout unexpected outcomes.

B.1 Estimated task planning

Code	Task	Hours	Weeks	Start date	End date
1	**Study of the state of the art**	40	2	20/12/13	03/01/14
1.1	Read previous work on the field	20	1	20/12/13	27/12/13
1.2	Test existing platforms	20	1	27/12/13	03/01/14
2	**Basic framework development**	100	5	03/01/14	07/02/14
2.1	Basic digital model	40	2	03/01/14	17/01/14
2.2	Select and setup simulator	20	1	17/01/14	24/01/14
2.3	Development of basic control software	40	2	24/01/14	07/02/14
3	**Optimization of gaits**	50	2.5	07/02/14	26/02/14
3.1	Algorithm selection	10	0.5	07/02/14	12/02/14
3.2	Gait optimization	40	2	12/02/14	26/02/14
4	**Distributed control algorithm**	50	2.5	26/02/14	14/03/14
5	**Development of remaining software**	90	4.5	14/03/14	16/04/14
5.1	Communication-related software	30	1.5	14/03/14	26/03/14
5.2	Module distributed controller	30	1.5	26/03/14	04/04/14
5.3	Test results	30	1.5	04/04/14	16/04/14
6	**Hardware platform development**	90	4.5	16/04/14	16/05/14
6.1	Design and manufacture PCB	30	1.5	16/04/14	25/04/14
6.2	Design and manufacture mechanical module	40	2	25/04/14	09/05/14
6.3	Test results with hardware platform	20	1	09/05/14	16/05/14
7	**Documentation and Thesis**	80	4	16/05/14	13/06/14
	Total	**500**	**25**	**20/12/13**	**13/06/14**

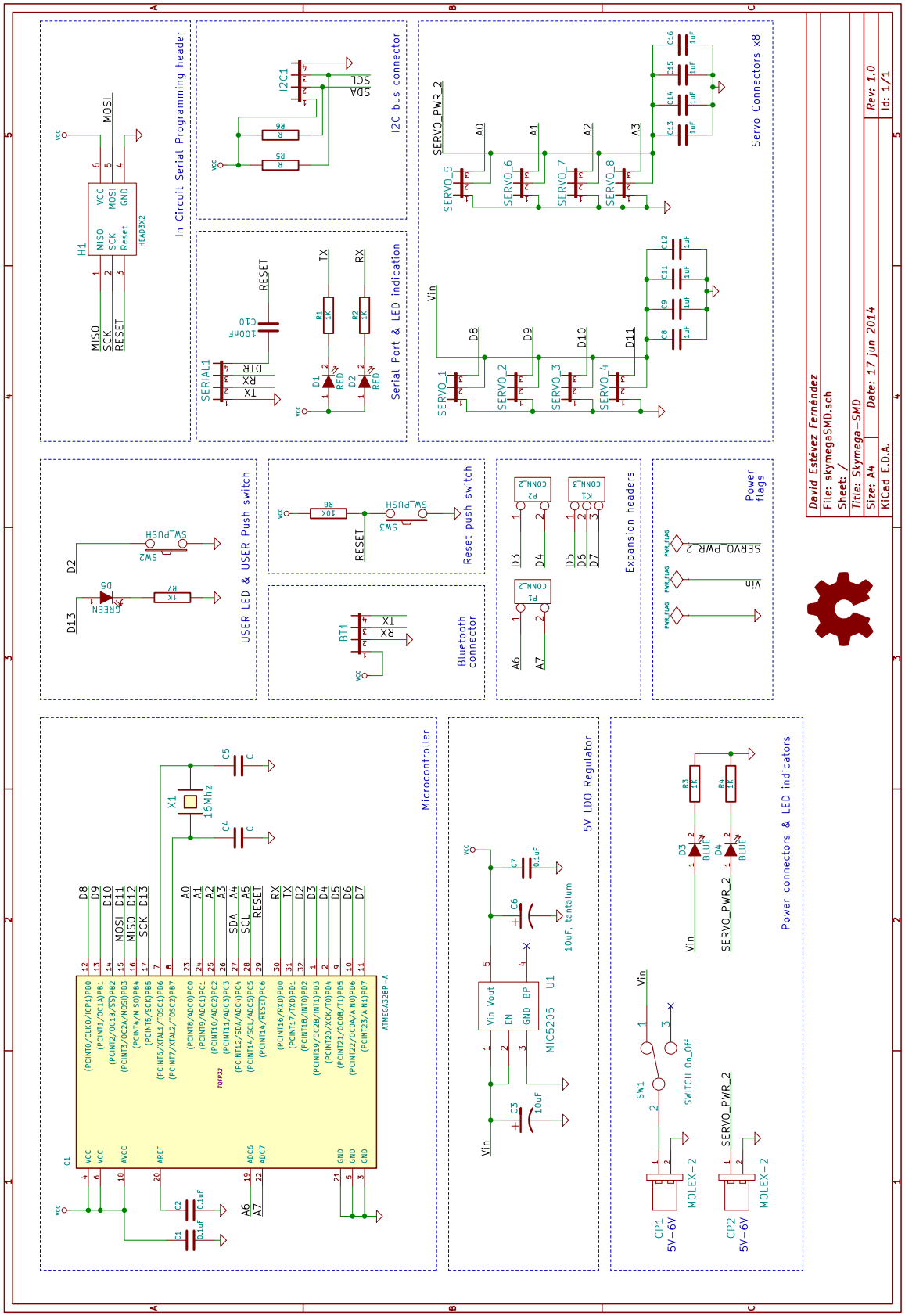
B.2 Gantt diagram



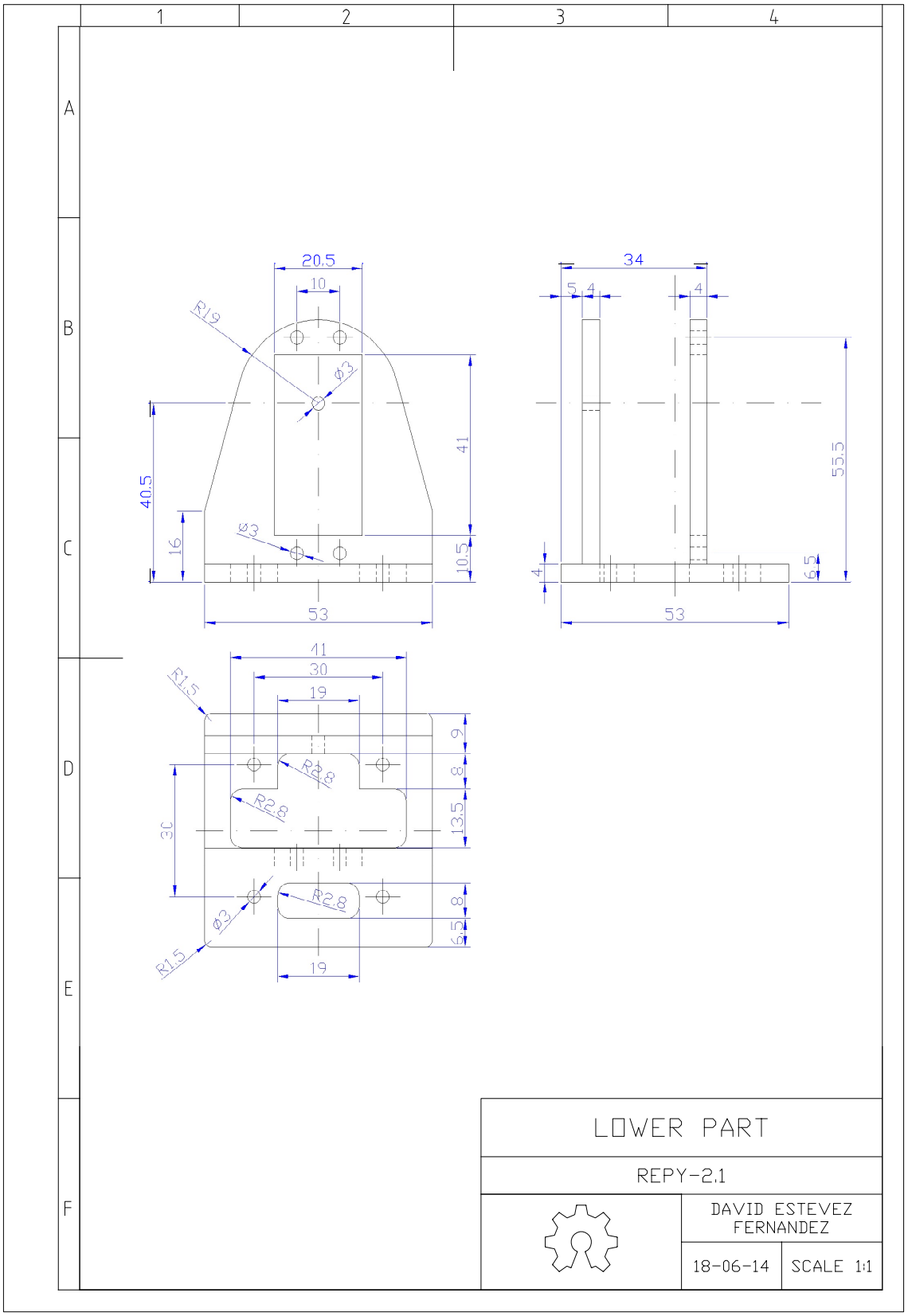
Appendix C

Schematics and Plans

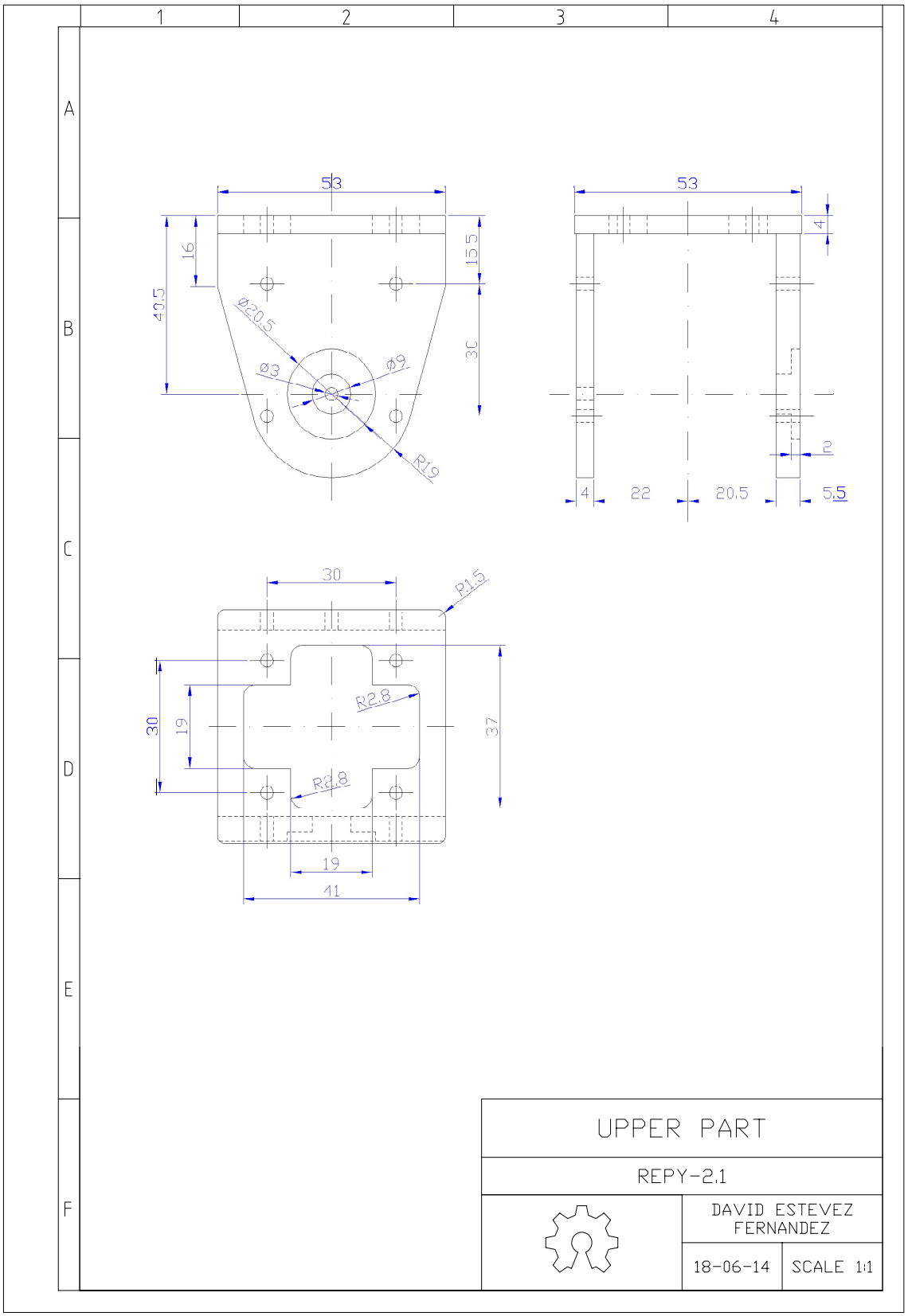
C.1 SkymegaSMD schematic



C.2 REPY-2.1 lower part



C.3 REPY-2.1 upper part



Bibliography

[1] Arduino website. http://www.arduino.cc. Accessed: 2014-06-22.

[2] CMake - Cross Platform Make. http://www.cmake.org/. Accessed: 2014-06-22.

[3] Decerebration - Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Decerebration. Accessed: 2014-06-22.

[4] googletest- Google C++ Testing Framework. http://code.google.com/p/googletest/. Accessed: 2014-06-22.

[5] Hormone - Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Hormone. Accessed: 2014-06-22.

[6] OpenSCAD - The Programmers Solid 3D CAD Modeller. http://www.openscad.org/. Accessed: 2014-06-22.

[7] A. Alonso-Puig. Application of Waves Displacement Algorithms for the Generation of Gaits in an All terrain Hexapod. *Climbing and Walking Robots*, pages 343–348, 2005.

[8] K. Beck. *Test-Driven Development by Example*. Addison Wesley - Vaseem, 2003.

[9] David Brandt, David Johan Christensen, and Henrik Hautop Lund. ATRON Robots: Versatility from Self-Reconfigurable Modules. *2007 International Conference on Mechatronics and Automation*, pages 26–32, August 2007.

[10] Z. Butler. Generic Decentralized Control for Lattice-Based Self-Reconfigurable Robots. *The International Journal of Robotics Research*, 23(9):919–937, September 2004.

[11] A. Castano, A. Behar, and P.M. Will. The Conro modules for reconfigurable robots. *IEEE/ASME Transactions on Mechatronics*, 7(4):403–409, December 2002.

[12] A Castano, WM Shen, and Peter Will. CONRO: Towards deployable robots with inter-robots metamorphic capabilities. *Autonomous Robots*, (1995):309–324, 2000.

[13] Jean-Pierre Charras. KiCad EDA Software Suite. http://www.kicad-pcb.org/display/KICAD/KiCad+EDA+Software+Suite. Accessed: 2014-06-22.

[14] R. Diankov. OpenRAVE website. http://openrave.org. Accessed: 2014-06-22.

[15] R. Diankov. *Automated Construction of Robotic Manipulation Programs*. PhD thesis, Carnegie Mellon University, Robotics Institute, August 2010.

[16] S. Funiak, P. Pillai, M. P. Ashley-Rollman, J. D. Campbell, and S. C. Goldstein. Distributed Localization of Modular Robot Ensembles. *The International Journal of Robotics Research*, 28(8):946–961, June 2009.

[17] J. Gonzalez-Gomez. Módulos repy-1. http://www.iearobotics.com/wiki/index.php?title=M%C3%B3dulos_REPY-1. Accessed: 2014-06-22.

[18] J. Gonzalez-Gomez. Módulos Y1. http://www.iearobotics.com/wiki/index.php?title=MÃşdulos_Y1. Accessed: 2014-06-22.

[19] J Gonzalez-Gomez. Modular robotics and locomotion: application to limbless robots. *Pdd. Universidad Autonoma de Madrid. Madrid*, 2008.

[20] J Gonzalez-Gomez and Eduardo Boemo. Motion of minimal configurations of a modular robot: sinusoidal, lateral rolling and lateral shift. *Climbing and Walking Robots*, 2006.

[21] J. Gonzalez-Gomez, A. Prieto-Moreno, and R Gomez. Campus científico 2010:taller de robots modulares. `http://www.iearobotics.com/wiki/index.php?title=Campus_cient%C3%ADfico_2010:Taller_de_robots_modulares`. Accessed: 2014-06-22.

[22] J. Gonzalez-Gomez, A. Prieto-Moreno, and R Gomez. Módulos MY. `http://www.iearobotics.com/wiki/index.php?title=M%C3%B3dulos_MY`. Accessed: 2014-06-22.

[23] J. Gonzalez-Gomez, A. Prieto-Moreno, and R. Gomez. Skycube - wikirobotics. `http://www.iearobotics.com/wiki/index.php?title=Skycube`. Accessed: 2014-06-22.

[24] J. Gonzalez-Gomez, A. Prieto-Moreno, I. Lima, and R. Gomez. Skymega. `http://www.iearobotics.com/wiki/index.php?title=SkyMega`. Accessed: 2014-06-22.

[25] J. Gonzalez-Gomez, A. Ranganath, and D. Estevez-Fernandez. OpenMR: Modular Robots plug-in for Openrave. `http://www.iearobotics.com/wiki/index.php?title=OpenMR:_Modular_Robots_plug-in_for_Openrave`. Accessed: 2014-06-22.

[26] Feili Hou and Wei-min Shen. Hormone-inspired Adaptive Distributed Synchronization of Reconfigurable Robots. 2006.

[27] Auke Jan Ijspeert. Central pattern generators for locomotion control in animals and robots: a review. *Neural networks : the official journal of the International Neural Network Society*, 21(4):642–53, May 2008.

[28] Auke Jan Ijspeert, Alessandro Crespi, Dimitri Ryczko, and Jean-Marie Cabelguen. From swimming to walking with a salamander robot driven by a spinal cord model. *Science (New York, N.Y.)*, 315(5817):1416–20, March 2007.

[29] B. Jacob. Eigen. `http://eigen.tuxfamily.org/index.php?title=Main_Page`. Accessed: 2014-06-22.

[30] D. Jakobovic. ECF - Evolutionary Computation Framework. `http://gp.zemris.fer.hr/ecf/`. Accessed: 2014-06-22.

[31] Rhys Jones, Patrick Haufe, Edward Sells, Pejman Iravani, Vik Olliver, Chris Palmer, and Adrian Bowyer. RepRap – the replicating rapid prototyper. *Robotica*, 29(01):177–191, January 2011.

[32] a. Kamimura, H. Kurokawa, E. Yoshida, S. Murata, K. Tomita, and S. Kokaji. Automatic Locomotion Design and Experiments for a Modular Robotic System. *IEEE/ASME Transactions on Mechatronics*, 10(3):314–325, June 2005.

[33] H. Kurokawa, K. Tomita, A. Kamimura, S. Kokaji, T. Hasuo, and S. Murata. Distributed self-reconfiguration of m-TRAN III modular robotic system. *The International Journal of Robotics Research*, 27(3-4):373–386, March 2008.

[34] H. Kurokawa, K. Tomita, A. Kamimura, S. Kokaji, T. Hasuo, and S. Murata. Distributed Self-Reconfiguration of M-TRAN III Modular Robotic System. *The International Journal of Robotics Research*, 27(3-4):373–386, March 2008.

[35] Haruhisa Kurokawa, Akiya Kamimura, Eiichi Yoshida, Kohji Tomita, Satoshi Kokaji, and Shigeru Murata. M-TRAN II: metamorphosis from a four-legged walker to a caterpillar. In *Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, volume 3, page 2454–2459, 2003.

[36] Jens Liedke. CoBoLD - A Bonding Mechanism for Modular Self-Reconfigurable Mobile Robots. (section III):2025–2030, 2011.

[37] Kevin Lipkin, Isaac Brown, Aaron Peck, Howie Choset, Justine Rembisz, Philip Gianfortoni, and Allison Naaktgeboren. Differentiable and piecewise differentiable gaits for snake robots. *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1864–1869, October 2007.

[38] Daniel Marbach and AJ Ijspeert. Online optimization of modular robot locomotion. *Mechatronics and Automation, 2005 ...*, (July):248–253, 2005.

[39] R Möckel, C Jaquier, and K Drapel. YaMoR and Bluemove—an autonomous modular robot with bluetooth interface for exploring adaptive locomotion. *... and Walking Robots*, pages 1–8, 2006.

[40] S. Murata, E. Yoshida, A. Kamimura, H. Kurokawa, K. Tomita, and S. Kokaji. M-TRAN: self-reconfigurable modular robotic system. *IEEE/ASME Transactions on Mechatronics*, 7(4):431–441, December 2002.

[41] C. M. Rovainen. Neurobiology of lampreys. *Physiological Reviews*, 59(4):1007–1077, 1979.

[42] B Salemi, WM Shen, and P Will. Hormone-controlled metamorphic robots. *Robotics and Automation, 2001. ...*, pages 94–99, 2001.

[43] Behnam Salemi, Mark Moll, and Wei-Min Shen. SUPERBOT: a deployable, multi-functional, and modular self-reconfigurable robotic system. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, page 3636–3641, 2006.

[44] Behnam Salemi, Mark Moll, and Wei-min Shen. SUPERBOT: A Deployable, Multi-Functional, and Modular Self-Reconfigurable Robotic System. *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3636–3641, October 2006.

[45] Wei-Min Shen, Yimin Lu, and Peter Will. Hormone-based control for self-reconfigurable robots. *Proceedings of the fourth international conference on Autonomous agents - AGENTS '00*, pages 1–8, 2000.

[46] WM Shen, B. Salemi, and P. Will. Hormone-inspired adaptive communication and distributed control for CONRO self-reconfigurable robots. *IEEE Transactions on Robotics and Automation*, 18(5):700–712, October 2002.

[47] WM Shen, Behnam Salemi, and Peter Will. Hormones for self-reconfigurable robots. *Proceedings of the 6th International Conference ...*, 2000.

[48] A Spröwitz and S Pouya. Roombots: reconfigurable robots for adaptive furniture. *Computational ...*, (August):20–32, 2010.

[49] Rainer Storn and Kenneth Price. Differential Evolution – A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces. pages 341–359, 1997.

[50] Matthew Tesch, Kevin Lipkin, Isaac Brown, Ross Hatton, Aaron Peck, Justine Rembisz, and Howie Choset. Parameterized and Scripted Gaits for Modular Snake Robots. *Advanced Robotics*, 23(9):1131–1158, January 2009.

[51] L. Thomason. TinyXML-2. `http://www.grinninglizard.com/tinyxml2/`. Accessed: 2014-06-22.

[52] A. Valero-Gomez and J Gonzalez-Gomez. The C++ Object Oriented Mechanics Library. `http://iearobotics.com/oomlwiki/doku.php?id=start`. Accessed: 2014-06-22.

[53] A Valero-Gomez, Juan Gonzalez-Gomez, Mario Almagro, and Miguel A. Salichs. Boosting mechanical design with the C++ OOML and open source 3D printers. *Global Engineering ...*, 2012.

[54] M. Yim. A reconfigurable modular robot with many modes of locomotion. *Proceedings JSME International Conference on Advanced Mechatronics*, pages 283–288, 1994.

[55] M. Yim. New locomotion gaits. *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, pages 2508–2514, 1994.

[56] M Yim, DG Duff, and KD Roufas. PolyBot: a modular reconfigurable robot. *Robotics and Automation, 2000. ...*, 2000.

[57] Mark Yim, Kimon Roufas, David Duff, Ying Zhang, Craig Eldershaw, and Sam Homans. Modular reconfigurable robots in space applications. *Autonomous Robots*, 14(2-3):225–237, 2003.

Figures:

Figure 1.1: https://unit.aist.go.jp/is/frrg/dsysd/mtran3/mtran3.htm
Figure 1.2a: http://biorobotics.ri.cmu.edu/projects/modsnake/pictures.html
Figure 1.2b: http://en.wikipedia.org/wiki/File:SpaceMolecubes.JPG

Figure 2.1a: http://robotics.stanford.edu/users/mark/segnode.gif
Figure 2.1b: http://robotics.stanford.edu/users/mark/tpoly.gif
Figure 2.2a: http://www.iearobotics.com/personal/juan/doctorado/tea/html/img20.png
Figure 2.2b: http://www.iearobotics.com/personal/juan/doctorado/tea/html/img23.png
Figure 2.2c: http://www.inovacaotecnologica.com.br/noticias/imagens/010180020527-polybotg3.jpg
Figure 2.3: http://www.isi.edu/robots/conro/fig/modu1n.jpg
Figure 2.3: http://www.isi.edu/robots/conro/fig/hardw.jpg
Figure 2.4: http://www.isi.edu/robots/superbot/SuperbotModule.JPG
Figure 2.4: http://www.isi.edu/robots/superbot/SuperBotCup.JPG
Figure 2.5: https://unit.aist.go.jp/is/frrg/dsysd/mtran3/mtran123.jpg
Figure 2.5: http://media.g-mark.org/award/2007/07D01004/07D01004_01_880x660.jpg
Figure 2.6a: http://www.iearobotics.com/wiki/images/3/3f/Modulo-Y1-1.jpg
Figure 2.6b: http://www.iearobotics.com/wiki/images/c/c6/Cube-rev-cobra.jpeg

Figure 4.1: http://upload.wikimedia.org/wikipedia/commons/8/8b/Csg_tree.png
Figure 4.3a: http://iearobotics.com/oomlwiki/doku.php?id=primitives:start
Figure 4.3b: http://iearobotics.com/oomlwiki/doku.php?id=parts:start
Figure 4.4a: http://www.iearobotics.com/wiki/images/3/3f/Modulo-Y1-1.jpg
Figure 4.4b: http://www.iearobotics.com/wiki/images/c/c6/Cube-rev-cobra.jpeg
Figure 4.5a: http://www.iearobotics.com/wiki/images/4/44/Modulo-y1-lote1-2.jpg
Figure 4.5b: http://www.iearobotics.com/wiki/images/2/2e/Repy1-v1.1-1.jpg
Figure 4.16a: http://www.iearobotics.com/wiki/images/4/43/Skycube-1.0-modulo-y1-2.jpg
Figure 4.16b: http://www.iearobotics.com/wiki/images/5/57/Skymega-pins.png
Figure 4.18a: http://www.rchobbies.co.nz/images/specs-futm0031.jpg
Figure 4.18b: http://www.iearobotics.com/wiki/images/2/2c/Futaba-pwm.jpg
Figure 4.19: http://www.adafruit.com/images/970x728/70-00.jpg

Figure 5.1a: http://modular.tek.sdu.dk/uploads/images/ATRON/ATRON06.jpg
Figure 5.1b: http://static.ddmcdn.com/gif/real-transformer-6.jpg
Figure 5.1c: http://nlp.stanford.edu/~wcmac/p/i/digitalclay.jpg
Figure 5.1d: Extracted from http://downloads.hindawi.com/journals/jr/2011/794251.pdf
Figure 5.2a: http://www.isi.edu/robots/conro/fig/modu1n.jpg
Figure 5.2b: ttp://www.inovacaotecnologica.com.br/noticias/imagens/010180020527-polybotg3.jpg
Figure 5.2c: http://www.iearobotics.com/wiki/images/3/3f/Modulo-Y1-1.jpg
Figure 5.3a: http://media.g-mark.org/award/2007/07D01004/07D01004_01_880x660.jpg
Figure 5.3b: http://www.isi.edu/robots/superbot/SuperbotModule.JPG
Figure 5.3c: http://modlabupenn.org/wp-content/uploads/SMORES_DoF_w_arrows2.png