Swiss Federal Institute of Technology of Lausanne
University of Fribourg

YaMoR Lifelong Learning

Michel Yerly

Project type	Master Project 2007 Computer Science Department University of Fribourg
Students	Michel Yerly
Supervisors	Alexander Sproewitz, Auke Jan Ijspeert
Responsible professors	Beat Hirsbrunner
Work place	EPFL Lausanne Switzerland
Start/end dates	19.03.2007/ 24.08.2007

Contents

Chapter 1

Introduction

In 2004 the Biologically Inspired Robotics Group (BIRG) started developing the modular robot YaMoR (Figure 1.1) at the Swiss Federal Institute of Technology of Lausanne, Switzerland. The acronym YaMoR stands for "yet another modular robot".

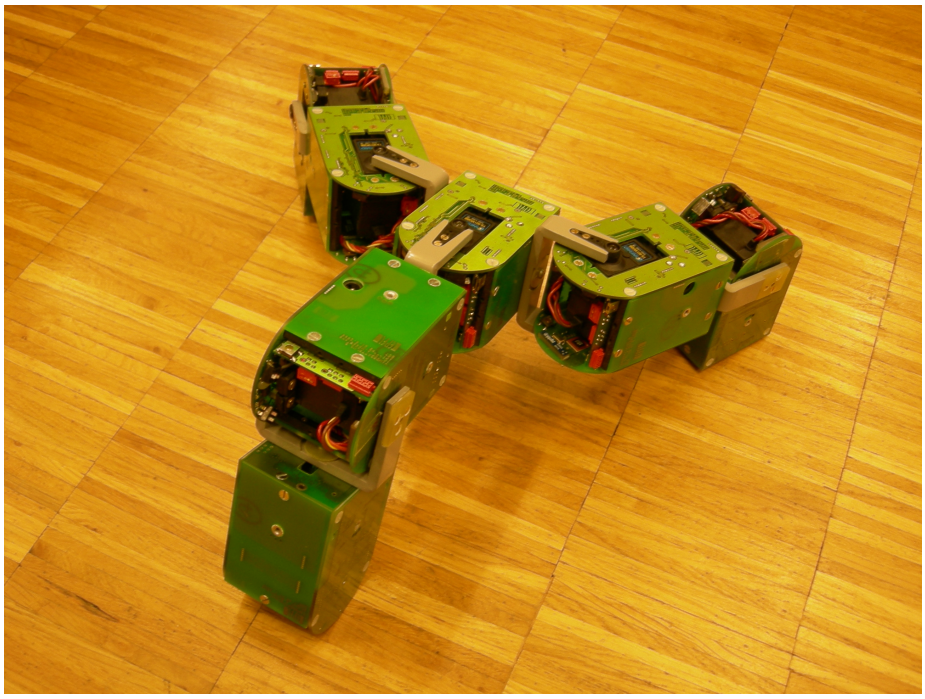


Figure 1.1: Tripod, a YaMoR robot. Picture taken from [15]

In fact a YaMoR robot is made up of several identical modules shown in Figure 1.2. Each one of them has one degree of freedom: the handle can move in a range of 180° around its axis. This movement is controlled by a servomotor inside the module. The modules can communicate with each other via a Bluetooth network using a protocol developed at the BIRG called SNP [18].

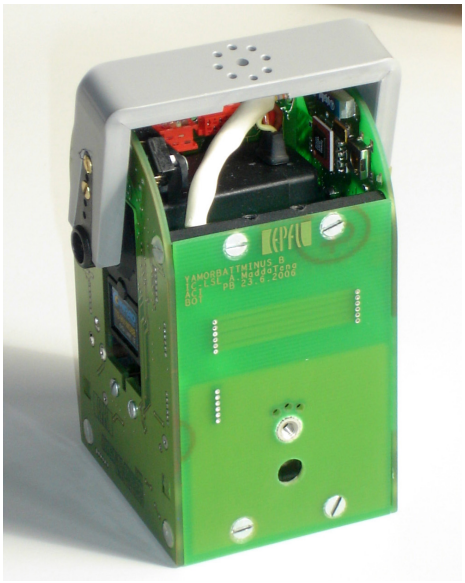


Figure 1.2: A YaMoR module. Picture taken from [15]

An advantage of modular robotics is that a robot can adapt or be adapted to the situation or the environment. If it has to pass for example through a tiny hole, it can transform itself into a snake to pass through and once it is on the other side, it turns itself back to a spider in order to walk faster. At a larger scale a modular robot could transform itself into a bridge for other robots to pass over a gap. Because all the modules are identical, they can be manufactured at a large scale, which considerably reduces the cost of each module. If a module gets damaged, it can easily be replaced by another one, and if the module software is well designed it is almost instantaneously operational with the rest of the robot. YaMoR requires a manual assembly of the modules, whereas its upcoming successor – Roombot – will be able to connect and disconnect modules by itself. In YaMoR they are attached to one to another by a screw on the handle that can fit in one of the five connectors of the other module. The connector can be connected to the module every 15° arround the screw. The module also contains a infrared sensor and accelerometers.

YaMoR robot is driven by a gait system inspired by the human spine called *Central Pattern Generators (CPG)* (see Section 6.1). The CPG model used in YaMoR is made of coupled non-linear oscillators. Each module contains one oscillator, which is coupled with those of other modules. The output of an oscillator controls the servomotor directly. A restricted set of parameters is sufficient to describe a rhythmic gait. When these parameters change, the old gait smoothly and quickly converges to a new one, without imposing strong physical constraints to the servomotors. Sensory inputs or constraints can be added on the oscillator of a module and would influence the behavior of the whole network. For example if the servomotors have feedback on their respective oscillators and a leg is blocked, this would influence the whole gait. When the leg is released, the gait would return smoothly to normality [1].

Because the shape of YaMoR is subject to change at any time, it is convenient that the robot is able to learn walking gaits by itself. The gait self learning also allows it to recover from odd situations, such has module malfunction, ground becoming slippery or muddy, slope variation, etc. but currently the robot is not able to adapt its gait by itself during its operation. Gait learning is made offline. YaMoR does not know anything about its geometrical structure, whereas many other modular robots know at least their tree or graph structure, such as the upcoming Roombot. It has thus no way for computing internally an efficient gait. It has to try out gaits and draw conclusions about them. Thanks to the CPG system plus the fact that symmetries can be introduced, the set of parameters to optimize is very reduced. Finding a good gait for a 10-modules robot usually takes less than five hours.

The technique used to find the values of the free parameters is a function optimization. This function returns the velocity of the robot in function of the inputed free parameters. Unfortunately, the function is not known and each time one wants to know the velocity for a given set of parameters, a simulation has to be performed, which takes more than $20s$. In order to find the maximum of this function, Powell's algorithm is used (Section 6.2.2). Currently the velocity measurement is made by a robot tracking system using a camera fixed on the ceiling of the experimentation area.

As previously mentioned, finding a gait can take several hours. This is the reason why many of the experiments are done in Webots. Webots is a robot simulator developed at the Swiss Federal Institute of Technologies of Lausanne and is widely used in the academic domain all over the world. It offers a 3D interface and a physics engine. The modeled robot are controlled by independent software. Webots brings several advantages over YaMoR experimentations: the simulation can be run faster than realtime, no need to recharge the batteries, no geometrical bounds for the experimenting area, no waiting for the Bluetooth network to be established. Furthermore an arbitrary number of modules can be used in Webots. Unfortunately, current interfaces to Webots and to the real robot are not complete (some features are missing), not made uniform and inconvenient to use.

1.1 Goals

The goals of this project are the following:

- Accelerometers characterization. Determine if the accelerometers can be used or not for positioning. The goal is to replace the current robot tracking system by calculating the position and velocity internally.
- Develop a framework to work with YaMoR easily either in Webots simulator or in the real world.
- Study and if possible improve the learning algorithm.
- Create a lifelong learning strategy for YaMoR in order to overcome possible module addition and removal or malfunction.

1.2 Achieved Work

The accelerometers were characterized in several experiments. It appears that they cannot be used for the positioning due to accuracy issues and the fact that it is impossible to differentiate between an acceleration caused by velocity variation or by the earth gravity.

The framework has been successfully developed and provides now a good platform to work with YaMoR either in Webots or on the real hardware. The new YaMoR Host 3 is a kind of driver that can be used either for controlling the real robot or to work with a Webots simulation. YaMoR Optimizer is a plugin-based optimization platform that is able to distribute the work on many computers. YaMoR Webots Bridge enables Webots simulation to be controlled via an external program.

The optimization algorithm was examined in detail to determine if it was possible or not to improve the learning phase either by shortening the time required or by improving the gait quality. An adaptation that improves both most of the time has been added.

A lifelong learning system was created, which allows YaMoR to detect an anomaly and relearn a new gait with this defect. A system allowing to remove or add modules dynamically in Webots was also developed.

1.3 Report Organization

This document is made of nine chapters and five appendices. You are currently reading Chapter 1, the introduction. Chapter 2 aims at giving a brief overview of what is done or what has been done in this research field. Chapter 3 provides the task list and the planning for this project. In Chapter 4 YaMoR module's accelerometer is characterized. Appendix B provides some measurement results for this chapter. Chapter 5 explains the framework software conception. Appendices C and D provide detailed information about the developed software. In Chapter 6 the CPG network and the gait optimization are explored. Chapter 7 talks about the long life learning process. Chapter 8 is the conclusion and finally Chapter 9 contains the acknowledgement. Appendix A lists some experimental configurations done with YaMoR. Appendix E contains a CD-ROM with all the files of the project.

Chapter 2

State of the Art

2.1 History

The story of self-reconfigurable modular robots began in the seventies with industrial robotic arms that could automatically change the tool at end of the arm to accomplish various tasks. A common connection mechanism concept that was applied to the whole robot has been developed for the first time at the end of the eighties by Toshio Fukuda for CEBOT (acronym stands for cellular robot) [3].

Greg Chirikjian, Joseph Michael, and Satoshi Murata developed lattice reconfiguration systems, whereas Mark Yim developed a chain based system at the beginning of the nineties. "There is a growing number of research groups actively involved in modular robotics research. To date, more than 30 systems have been designed and constructed." [2] The next section presents some of the current systems.

2.2 Current Modular Robots

2.2.1 Roombots

The project for the upcoming successor of YaMoR, Roombots [4], is developed at the BIRG and is funded by Microsoft Research Cambridge and the Swiss Federal Institute of Technology of Lausanne. It aims at creating modules that can assemble together to create furniture such as tables, chairs, stools, etc. that can move around by themselves. Moreover Roombots is capable of reconfiguration i.e. detach some modules and reattach them at another place to change its shape. If an user requires more chairs for example, a table can reconfigure itself into four chairs. Or vice-versa.

The objectives are the design and construction of Roombots modules, the control of locomotion of multi-module Roombots, the control of self reconfiguration and the design and realization of the robot user interface.

2.2.2 M-TRAN III (2005)

From the author's point of view, M-TRAN III [5], developed by AIST and Tokyo-Tech since 1998, is currently the most interesting modular robot. It is an hybrid (lattice and chain) self-reconfigurable robot. Its low energy consuming mechanical connectors allow it to dynamically reconfigure.

Currently M-TRAN group is working on self-reconfiguration, locomotion and adaptation. Their most active field of research is the self-reconfiguration. They study how to reconfigure from the current structure to another given one, the parallel local reconfiguration forming a global flow of module clusters, generation of small structures from a block of modules. They also plan to work on the self-repair: removal of a damaged module and the reconnection of separate robots [5].

For the locomotion they designed motion patterns for various configuration of the robot by offline computation using a genetical algorithm to find out the parameters of a central pattern generator that produces the gait. The locomotion patterns produced by the genetic algorithm were evaluated by a simulator. For the adaptation, they make experiments on realtime CPG control for the gait to be generated adaptively on a surface with variable friction. They also make experiments on shape change in order to walk in conditions where the current shape does not allow to go any further.

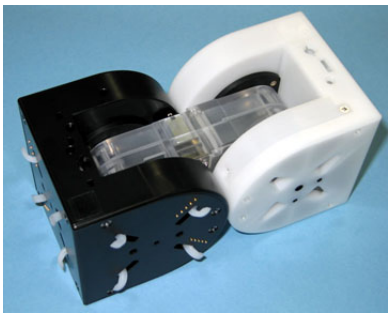


Figure 2.1: MTRAN III module. Picture is taken from [5].

2.2.3 Stochastic 3D (2005)

The Computational Synthesis Laboratory of Cornell University is working on the three dimensional stochastic reconfiguration of modular robots [6]. The particularity in their project is that the modules by themselves have no means of actuating. Furthermore, they can only draw power when attached to the main structure. The unattached modules are moved randomly by the environment. The research group explores factors that govern the rate of the assembly and reconfiguration and shows that self reconfiguration can be exploited to accelerate the assembly of a particular shape, as compared with the static self-assembly. It also explores building such systems at microscale.

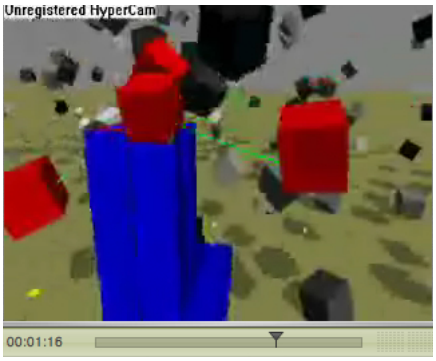


Figure 2.2: Stochastic assembly process. Video snapshot taken from [6].

2.2.4 Molecubes (2005)

Developed by Zykov, Mytilinaios, Adams, and Lipson at the Computational Synthesis Laboratory of Cornell University, Molecubes [7] is a self-reconfigurable robot where modules are connected by electromagnets. The main research field with this robot is the self-replication.

Figure 2.3: Molecube (8 modules here). Picture is taken from [7].

2.2.5 SuperBot (2006)

SuperBot [8] is a reconfigurable modular robot developed by Prof. Wei-Min (Weimin) Shen at University of Southern California. The goal of SuperBot is eventually to accomplish mission is space. SuperBot modules have three degrees of freedom and their casing is made of metal. For the control and coordination of multi-module structures they use a system called *digital hormone control* [10].

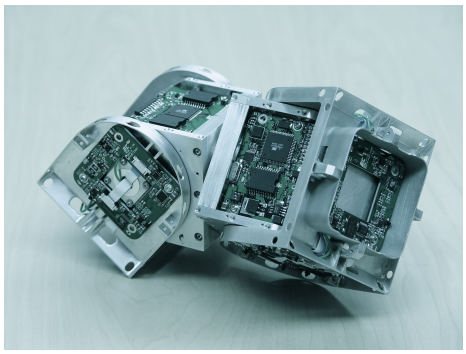


Figure 2.4: A SuperBot module. Picture is taken from [9].

2.3 Lifelong Learning

The life of a robot starts at the first time it is turned on and lasts until the robot is reset or destroyed. Turning it off does not necessarily terminate its task but can also simply suspend it. Very often robots have their behavior set before their task starts. However it is possible to give them the ability to learn an exhibit new behavior by themselves during their operation. This is called lifelong learning.

Whereas lifelong learning has already been implemented in regular robots for long, it remains a big challenge to give a modular robot the possibility to completely and dynamically change its gait to adapt to a new situations not known a priori, because it is continuously changing its topology.

Chapter 3

Task List

3.1 Tasks

- **Getting Started**: this include studying the existing code, learn about central pattern generators, learn how YaMoR works in its whole, how the gait optimization works, how Webots works.
- **Study Webots API**: get the necessary knowledge to work and interact with Webots by program.
- **Robot Movies**: Make some robot movies for the website. The goal here is also to get familiar with the whole robot setup.
- **Accelerometers Characterization**: make measurements on the Sensorboard's accelerometers. The purpose is eventually to determine if the accelerometers can be used in gait optimization and lifelong learning and how.
- **YaMoR Host 3 Coding**: create a kind of common driver software for controlling YaMoR either in a Webots simulation or in the real world. The program should also be controlled by another program. Create the necessary Webots interface for this.
- **YaMoR Optimizer Coding**: create a framework for YaMoR gait optimization that uses YaMoR Host 3 to interact with the robot. Create a plugin architecture.
- **Various Optimizers Coding** Create some plugins for YaMoR Optimizer. Powell, PSO, brute force optimizer, ...
- **Powell Improvement**: Study Powell's algorithm in detail and see if improvements are possible.
- **Change Detection**: Create a lifelong learning plugin for YaMoR Optimizer that detects when an anomaly occurs in YaMoR. Adapt the Webots interface to be able to add or remove modules dynamically to YaMoR.
- **Change Adaptation**: Add the ability to recover from anomaly in the lifelong learning plugin.

- **Report Writing**: the redaction of this document.

3.2 Schedule

Figure 3.1 lists all the tasks with the estimated time investment for each task. The whole work is planned on 20 weeks but one more week was added to compensate official swiss non-working days that occurred during the project. The two gaps correspond to the author's holidays.

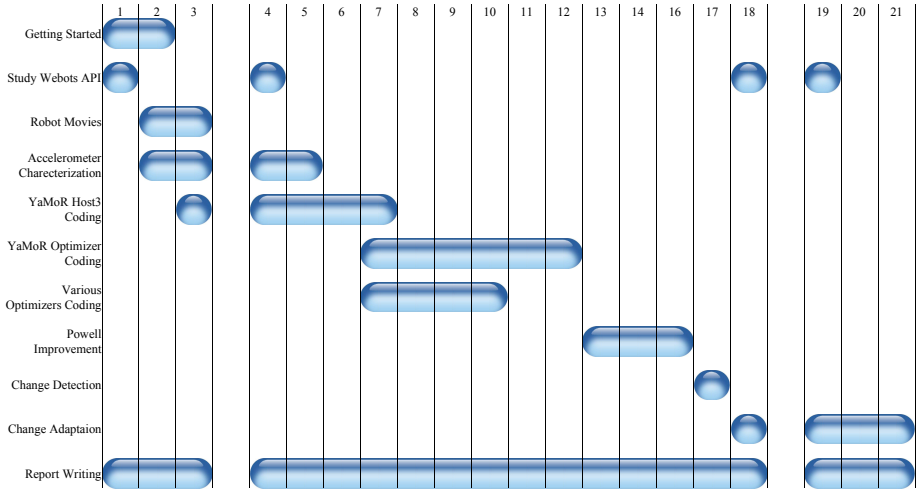


Figure 3.1: Planning. Numbers correspond to the weeks of of work.

Chapter 4

Accelerometer Measurement

In order to determine in which way the YaMoR modules accelerometers could help in the control of locomotion some measurement have been made. The point here is mainly to determine if the accelerometers can or cannot be used to estimate the position of a module and eventually the average velocity of the robot. Beside this main goal, they could also be used to detect impacts, check if a leg is moving or not or even measure the orientation of a module.

4.1 Sensor Board

4.1.1 Overview

The Sensor Board [11] shown in Figure 4.1, a part of the YaMoR module, is located at the bottom the box. It contains accelerometers and an infrared sensor, all of them accessible via a programmable PIC microcontroller. The accelerometers can be used either statically to measure the orientation of the module or dynamically to get the acceleration, assuming the orientation is known. The infrared sensor is able to measure a distance between an object and the module or the ambiant light intensity. The Sensor Board communicates with other parts of the module or a personnal computer via a serial line (UART) controlled by the PIC. Additionally there are two LEDs — a green one and a red one — that can be controlled via the PIC.

4.1.2 Accelerometers MMA7260Q

The accelerometers, which are here the main part of interest are wrapped in a MMA7260Q manufactured by Freescale Semiconductor. It has a selectable sensitivity from 1.5g to 6g, low current consumption ($500\mu A$) and low voltage operation ($2.2V - 3.6V$). Typical applications are freefall detection, image stability in cellphone cameras, motion sensing in pedometers, tilt and motion sensing in video games and robotics.

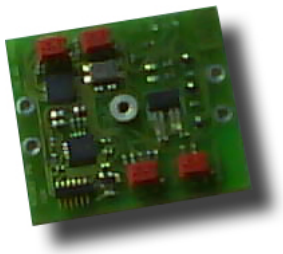


Figure 4.1: Sensor Board

4.1.3 PIC 16F876A

The PIC microcontroller allows to read sensors values, make computations and communicate the result to the outside of the board. On the Sensor Board it is running at 10MHz. The following list shows up its main features.

- 22 inputs and outputs
- Self programmable, with low voltage (here 3.3V)
- 14kB flash for the program memory, 368 bytes RAM, 256 bytes data EEPROM for persistent storage
- 2 analog comparators, 5 analog to digital converters (10 bits of precision each), 2 capture / compare / PWM pins
- I^2C bus and USART

4.2 Measurement Setup

4.2.1 Setup

The measurement setup shown in Figure 4.2 consists in a camera $2.7m$ above the ground, two interconnected computers and the SensorBoard with a LED attached. PC1 collects the data sent by the SensorBoard through a RS-232 serial line. It also collects the data from the LED tracking system. PC2 is responsible for the LED tracking. It runs CamApp (see Section 5.1.2), which finds the position of the LED on the camera image.

The acceleration measured on each of the axis is translated into a directly proportional voltage. This voltage is then quantized by a 10-bits analog to digital converter in the PIC thus generating an integer value between 0 and 1023. The sensitivity is set to $1.5g$, which is the most sensitive one. According to the manufacturer, this is the most suitable for position estimation. The output signal will change by $800mV$ for an acceleration of $9.81m/s^2$, which means that an unit of the integer value corresponds to an acceleration of $0.0395m/s^2$ according to the following equation.

$$\delta = \frac{g * V_{DD}}{0.8V * 1024} = \frac{9.81m/s^2 * 3.3V}{0.8V * 1024} = 0.0395m/s^2$$

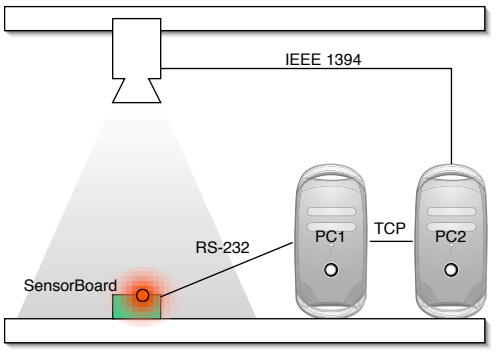


Figure 4.2: Measurement setup

The data is then processed by the PIC and sent out via its UART serial interface to a computer that logs everything that comes in.

4.2.2 PIC Software

As described in Section 4.3 two ways of getting data are explored. First read out raw data for noise measurement and then read out filtered data. Therefore two PIC programs are required. You can find the developed assembly source code of these on the enclosed CD-ROM.

4.2.3 RS232 Software

A program has been coded to read the incoming data computer side and write them to a comma separated values file. The developed C++ source code can be found on the enclosed CD-ROM.

4.3 Measurements

In the first part of this section the noise of the accelerometers output signal is measured. The second part explains some filtering techniques applied to the raw data to improve the position estimation and compares unfiltered, filtered and LED tracking data. Measured values and graphs can be found in Appendix B.

4.3.1 Noise Measurement

Noise is added to the measured values at many places. The imprecision of the accelerometer, the electromagnetic disturbances on the wires that bind the accelerometers and the PIC, the imprecision of the analog to digital converter and the quantization of the value are all sources of noise.

Noise measurement have been done to determine how the noise affects the signal. The accelerometers signal is measured during about 10 seconds with the Sensor Board steady, which will provide more than 1000 sample values for each axis. Because the sensor board is not moving, the theoretical acceleration value is 0.

4.3.2 Filtering

Filtering techniques dedicated to the improvement of the data quality are applied. This section shows the filtering method proposed by the manufacturer of the accelerometer MMA7260Q for the position tracking [13]. The raw data crosses three filter stages, namely the noise filter, the window filter and then the movement end detection. Those filters are explained below.

Noise Filter

The noise filter is applied directly on raw data from the accelerometer. Instead of measuring only one instantaneous value, 64 values are averaged for each axis within the same time window. This is basically a low-pass filter, which will evict some of the noise generated by electromagnetic perturbations and other sources. The noise signal should sum up to zero over the time.

Window Filter

If an acceleration is detected it will be translated as a velocity. If the counter-acceleration is not measured the velocity will stay constant leading to a wrong position estimation of the object. This may happen because some noise still passes through the noise filter. To avoid this, very weak accelerations (between -3 and +3) are considered as no acceleration at all.

Movement End Detection

When the moving object returns to steady state the acceleration values should perfectly sum up to zero (from the moment it started moving), which practically almost never happens because of the noise and the quantization of the signal. If they do not then the computed speed does return to zero and the estimation of the position will be worse and worse over the time. This can be tweaked using this so called "movement end detection" which assume that if no acceleration is detected during a given amount of time then the object is very likely steady. In this case the velocity is set to 0.

4.3.3 Acceleration, Velocity, Position

The accelerometers output an acceleration signal but the point of interest here is the position and not the velocity, because from the position the average velocity can be easily computed. The acceleration signal is thus integrated one time to get the velocity signal and a second time to get the position signal. The trapezoidal method [13] was chosen for the integration.

4.3.4 Movements Measurements

Several movements have been tried and measured. Most of them are movements on the ground firstly because YaMoR is designed to walk on a flat surface and secondly because the camera cannot track vertical movement and this would result in a loss of precision on the two other axis. However other kinds of movement that might happen in YaMoR, such as impact and tilting have also been experimented. The following list enumerates the movements tried.

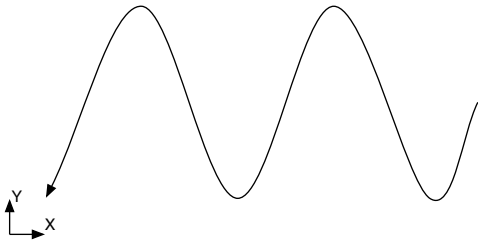


Figure 4.3: Pattern for the sine move.

- Sine movement, about $10cm$ amplitude and $25cm$ displacement on the X axis, shown in Figure 4.3.
- Circular movement, about $5cm$ radius, clockwise, returns to the starting position, shown in Figure 4.4.
- Go and return movement, about $10cm$, returns to the starting position, shown in Figure 4.5.
- Random movement in the air, max $10cm$ high, with tilting, returns to the starting position.
- Impact, dropped from a height of $7cm$.

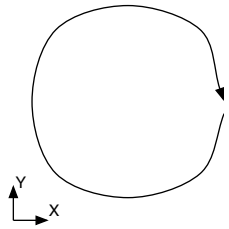


Figure 4.4: Pattern for the circular move.

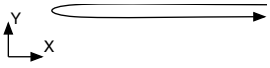


Figure 4.5: Pattern for the go and return move.

4.4 Results

This section provides a summary of the results obtained. For the details please refer to Appendix B.

4.4.1 Noise

Figure B.1 shows the noise signal that is present after quantization, thus after having passed through the PIC's analog to digital converter. The probability distribution of the measured values for each axis is shown in Figures B.2, B.3 and B.4.

The noise ratio is computed in function of the full scale, where N_{ampl} is the noise amplitude, δ is the unit factor ($0.0395 m/s^2$) to convert the integer value that comes out from the analog to digital converter to an acceleration in meters per square seconds.

$$N_{ampl} = N_{max} - N_{min}$$

$$N_{ratio} = \frac{N_{ampl}}{1024 * \delta} = \frac{N_{ampl}}{40.44 m/s^2}$$

From those measurements the minimal value N_{min} and the maximal value N_{max} of the noise are extracted and shown in Table 4.1 as well as the computation of the amplitude N_{ampl} the noise ratio N_{ratio} for each axis.

Axis	N_{min} $[m/s^2]$	N_{max} $[m/s^2]$	N_{ampl} $[m/s^2]$	N_{ratio}
X	-0.269	0.363	0.632	1.56%
Y	-0.267	0.246	0.514	1.27%
Z	-0.249	0.304	0.553	1.37%

Table 4.1: Noise Amplitude

4.4.2 Movements

Table 4.2 indicates for each movement the references of the figures that show the accelerations, the velocities and the positions for each axis. Please note that LED data is missing for the sine move from $1.5s$ to $2s$.

Table 4.2: Figures references

Movement	Accelerations	Velocities	Positions
Sine	Figure B.5	Figure B.6	Figure B.7
Circular	Figure B.8	Figure B.9	Figure B.10
Go and return	Figure B.11	Figure B.12	Figure B.13
Random	Figure B.14	Figure B.15	Figure B.16
Impact	Figure B.17	Figure B.18	Figure B.19

4.5 Discussion

4.5.1 Noise

The noise measurement shows that there is a noise with a maximal amplitude of $0.632 m/s^2$. Altough this represents only 1.56% of the full scale, this is quite huge in comparison to the low acceleration variations expected in the YaMoR application (about 5% of the full scale for typical accelerations). Moreover the measurements were done without the whole YaMoR module, that means with a stable power supply and no servomotor spinning in the neighborhood, which could also add noise.

The probability distribution graphs were expected to show gaussian curves, which is obviously not the case. The worst one is shown in Figure B.2 where some values close to the center of the curve never occurred. Imperfections in the PIC's analog to digital converter would explain this, but no further investigation will be made in this project to determine the exact origin of this curious behavior because of lack of time.

4.5.2 Movements

As one can see in positions graphs where the LED data is available (blue curve), the filtered data (red curve) is always much better, i.e. closer to the LED curve than the unfiltered data (black curve).

At the end of each move, the object is steady. In the velocities graphs of all movements one can see the effect of the movement end detection that translates into a brutal return to zero of the velocity (red curve). On the positions graphs this causes the position to stabilize. When the object has stopped moving then its velocity gets set to zero, thus the position is not changing and getting wronger and wronger anymore. The movement end detection improves the positioning in all the measurements done.

The filtered and unfiltered data seem to be very close in the random move, but pay attention to the scale: they are actually both very far from the LED tracked position (blue curve). This big error is due to the tilting of the Sensor-Board. In fact the accelerometer cannot see the difference between tilting and accelerating because of the earth gravity that always enters into account when measuring an acceleration.

In the impact measurement the accelerometers offsets changes after the shock, which result in a constant acceleration. This one is not visible in Figure B.17 because the shift is too small in comparison with the amplitude of the impact signal, but in Figure B.18 one can see the effect of this slight change: the speed continues to increase except for the Z axis where the shift was fortunately small enough to be cancelled by the window filter. This is also the reason why the movement end detection worked for this axis.

4.5.3 Conclusion

The measurements performed show that the accelerometer is not suitable for positioning. The previously mentionned imprecision problem, even after filtering are summed up during the computation of the speed (integration) and again during the computation of the position (integration too). Small acceleration

imprecisions lead to big position imprecisions. Moreover, after an impact, offset voltages change making the PIC think that an acceleration is measured, thus speed increases more and more (Figure B.18).

Also if the orientation of the module changes, it becomes very difficult to estimate the position. Since the measured acceleration vector is composed of the earth gravity and the actual acceleration, either the orientation is known and the actual acceleration can be computed or the actual acceleration is known and the orientation can be determined.

However the accelerometer can still be used to detect impacts. Impacts are valuable information to detect for example that a limb hit the ground. The fact that offset voltages change and the orientation of a module does not affect the detection of an impact. They can also be used to determine if a module is moving or not.

One could also try to determine precisely where do the strange noise come from. As stated in Section 4.5.1, there is very likely a problem with the analog to digital conversion within the PIC. Measuring which range of voltage corresponds to which digital value may provide relevant clues, for example differently sized ranges.

Chapter 5

Framework Software

5.1 Existing Software

5.1.1 YaMoR Host

YaMoR Host 2.0 developed by Jerome Maye at the Swiss Federal Institute of Technology of Lausanne (EPFL) was used to control the robot and perform the gait optimization. This comprises the establishment and destruction of a bluetooth network between the modules, the setup of the CPG structure and parameters as well as the activation and deactivation of various part of each module. The gait learning is done using one of the three implemented optimization methods, namely PSO, Brent and Powell. YaMoR Host 2.0 is also able to get the position of the robot using a robot tracking system by camera running on another computer. An altered version of YaMoR Host 2.0 exists for running a Webot simulation instead or at the same time than the real robot.

5.1.2 CamApp

This program is a LED tracking application that determines the position (x,y) of a lit LED on an realtime image captured by a camera. It accepts TCP connections and sends continuously the tracked position of the LED to the remote computer.

The position given by CamApp is a pair of integer number representing the coordinates X and Y of the tracked led. These values are expressed in [pixels]. The ranges of the X value is an integer between 0 and 639 and the coordinate of the Y value is an integer between 0 and 479. A function has to be applied to convert these coordinates in meters taking in account the distance of the LED to the camera. Even if the LED is moved on a flat surface, the distance to the camera is changing.

5.2 Motivations

5.2.1 Time Saving

To create a robot configuration in YaMoR Host 2.0 the user has to create five text files by hand, describing the bluetooth network, the limit angles, the CPG

structure, the CPG parameters and the optimization parameters, which is not so convenient.

Once these files are created the user runs YaMoR Host 2.0. She is asked to enter the serial communication port number, the name of the computer that runs CamApp, the port number for CamApp and the identifier of the master module. Then she has to type "B" to build the bluetooth network, "L" to send the limit angles, "U255" to enable the UART of the modules, "C255" to enable the CPG oscillators, "N" to send the CPG structure, "O" to set the optimization parameters and CPG parameters, "M255" to enable all the servomotors.

This work aims at improving all these time-consuming steps, optimize and or automate them in a more user-friendly interface.

5.2.2 Flexibility

YaMoR Host 2.0 is not very flexible. If the user wants to add features she has to understand most of the code and program her modification inside of it. If another user wants to develop other features at the same time, the two modified version have to be merged at the end. If the next user does not like the new features, she has to remove it by first understanding all of the code. A modular architecture and an object oriented programming language would increase the flexibility drastically.

5.2.3 Maintenance

As stated in Section 5.1.1, there are two version of YaMoR Host 2.0. One for the real robot, and one for the simulation in Webot, implemented as a Webot controller. The problem is that when fixing a bug or adding a feature in one of them, the same modification has to be done to the other one. A unique program for both the simulation and the real world would be much more suitable.

5.2.4 Programming Language

YaMoR Host 2.0 was written in C, which is an old language. Its main advantages are its speed of execution, the low memory consumption and the low level control over the underlying hardware. However it is much more difficult to write programs than with nowadays languages because the programmer has to worry about things she should not have to, such as memory management, function prototyping, array overflows, etc. Finally it is not an object oriented language, which would provide a great level of flexibility and reusability.

As speed of execution is not a critical point in this application, the C programming language is not adapted. Cross-platform compatibility is not required, therefore Microsoft's C# is a good choice. It allows first to easily reuse the code of YaMoR Host 2.0 with few adaptations since the syntax is pretty much the same. Secondly the very good integrated development environment Microsoft Visual Studio 2005 can be used, which will save a lot of programming time. Thirdly it is an object oriented language.

5.3 Architecture

The whole software is split up in several programs as shown in Figure 5.1. A brief overview of all these parts is given in this section although each of them is described in more detail in the following sections.

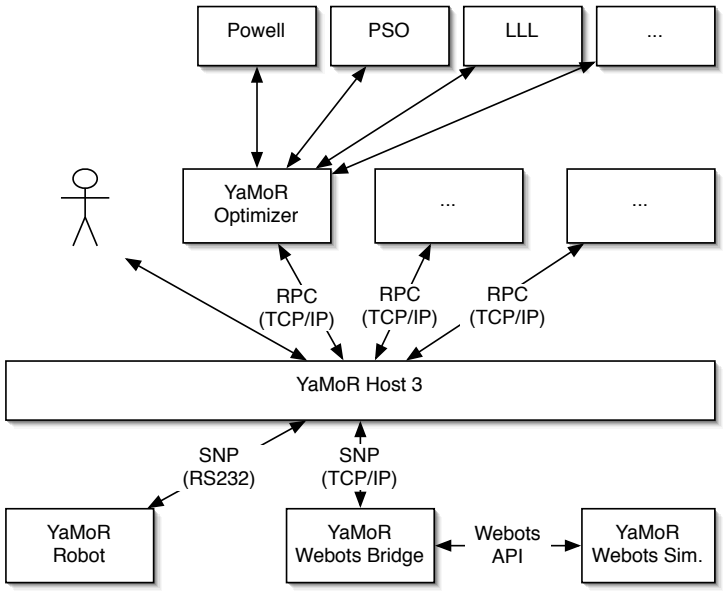


Figure 5.1: Software global architecture

The purposes of YaMoR Host 3.0 is to provide a common interface either to control the real robot or a Webots simulation, and to hide the underlying SNP protocol [18] used by the robot from the user or driving programs. YaMoR Host 3.0 also provides a friendly user interface to configure the robot.

YaMoR Webots Bridge deals with all the stuff that is not exactly the same in the real world than in the Webot simulation. It also dispatches SNP messages to all the YaMoR modules.

YaMoR Optimizer is a program that connects to YaMoR Host 3 via RPC (remote procedure call) and performs gaits optimizations using end-user made libraries like *Powell*, *PSO*, *Lifelong learning*, ...

5.4 YaMoR Webots Bridge

5.4.1 Analysis

Needs

A Webots simulation is used to perform experiments in a more convenient way than on the real robot. In the real world one has to charge accumulators, wait long delays to connect all the modules via Bluetooth (up to seven seconds per

module) and one second between each SNP[1] message that is sent to the robot.

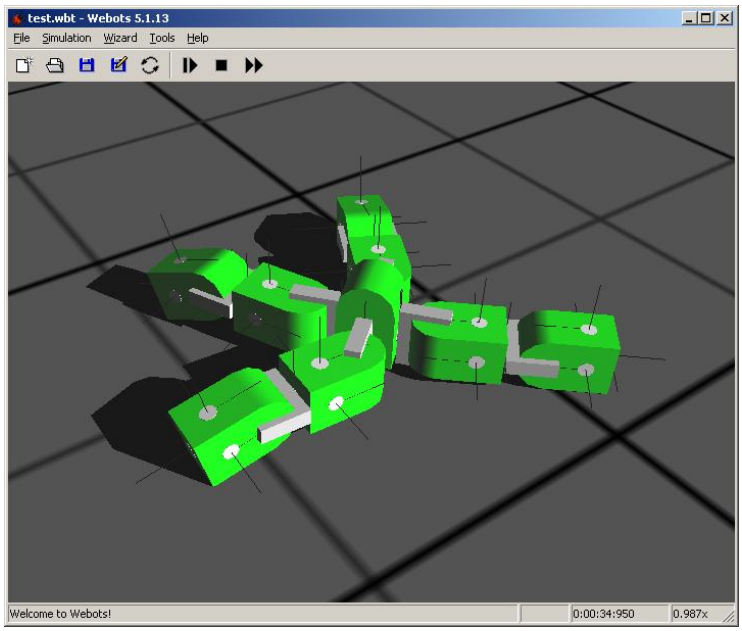


Figure 5.2: Webots snapshot: simulation of YaMoR quadruped.

The purpose of YaMoR Webots Bridge is to allow any software to control a YaMoR Webots simulation by sending it the same data as it would have done to the real robot. As the YaMoR modules use the same program in the simulation as in the real world, they read in SNP messages. This way one can assume that it is possible to send the same bytes either to the real robot or to the Webots simulation. Actually some small changes need to be done and are described in the following sections.

Messages other than SNP message should also be read by YaMoR Webots Bridge so that the client can interract with Webots too, for example to get the position of the robot or to tell Webots which module is the master node.

Although Webots offers direct interfaces to interact with it, a TCP/IP communication is preferred because it allows to run Webots either on a Windows, Mac or Linux machine. Often computer clusters are made of Linux machines, thus not being Windows dependent in this part is a big advantage for the parallelization of simulations (see Appendix D.2.4).

Input

YaMoR Webots Bridge should be started with as command line argument the name of the central module i.e. the one that returns its position when the robot position is requested. Then it should start listening for an incoming TCP connection, which will provide all the commands to be executed by the program. Those are either SNP messages or special requests to be passed to Webots.

[1]Protocol used for the the communication of the modules [18]

Output

The same TCP connection that is used to read in commands is also used to provide SNP feedbacks and other special Webots requests replies.

Webots Special Commands

The following list is a non exhaustive list of Webots special commands that YaMoR Webots bridge should be able to satisfy.

- Get the robot position
- Set a timer
- Get the simulation time
- Set the master node ID
- Connect a module to another one
- Disconnect two modules

5.4.2 Design

Architecture

YaMoR Webots Bridge is the so called *Supervisor Controller* in Webots. It is responsible of the whole simulation whereas standard controllers take each care of one of the YaMoR module. Every controller is compiled to a binary file and every instance of a controller results in a process running one of these binary file. For example a four modules robot requires two binary files – one for the supervisor and one for the module controller – and will lead in the execution of five processes – one for the supervisor and one for each module. The standard controller *YaMoR 2007* used in this project has been developed by Jerome Maye [15] and will not be described here.

Figure 5.3 shows YaMoR Webots Bridge architecture, which is run as the supervisor controller. It waits for an incoming TCP connection. The incoming messages are then split in two groups namely the SNP messages and the special Webots commands. The SNP messages are directly forwarded to every modules (i.e. to the standard controllers). The special Webots command are decoded and executed by the supervisor controller.

Threading

There are two tasks to perform that may have to wait for external events. Thus they are each executed in a separate thread. One of them is reading and filtering the incoming TCP traffic, the other one is reading the data coming from the modules, and possibly generate outgoing TCP packets.

As the two thread access the same resource, namely the TCP socket, synchronization has to be performed in order to prevent data corruption. The critical section has been chosen here to solve this problem. The Windows API offers three methods to handle critical sections.

- InitializeCriticalSection()

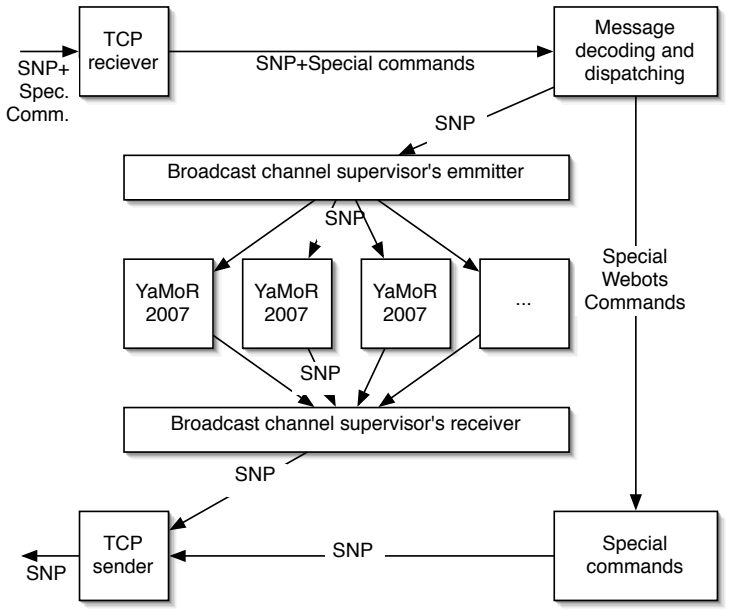


Figure 5.3: YaMoR Webots Bridge architecture

- EnterCriticalSection()
- LeaveCriticalSection()

Each of these functions take as parameter a variable that represents the critical section. The critical section has first to be initialized. Then when a thread calls *EnterCriticalSection* it passes through if no other thread already entered the critical section; otherwise it is blocked until the thread, which is in, calls *LeaveCriticalSection*. This allows to prevent a block of code of being executed by more than one thread at a time.

Module Positioning

In order to connect two modules together, their connectors must be perfectly aligned before being able to activate them. A challenging part of this project was to compute the module position and rotation for connection. When connecting two modules together one of them stays at its place and the other one is moved so that the connection is possible. The former is called *destination module*, whereas the latter is called *source module*.

For each connector the position relative to the module is determined. Also the normal going to outside and the connector north vector are determined. Those are all fixed values except for the front connector, whose values depend on the servomotor angle (front connector computing is described further). Figure 5.4 shows the positions of the connectors on the module and Figure 5.5 shows the normal and north vector of a connector.

Firstly the module rotation is computed. Rotations in Webots are expressed by a rotation axis represented by a vector and an angle. The rotation matrix R

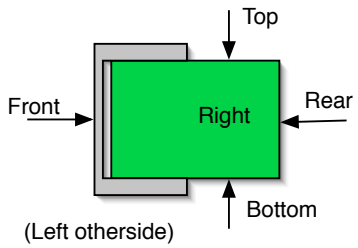


Figure 5.4: Connectors positions

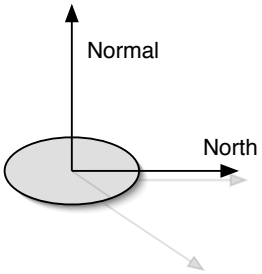


Figure 5.5: Connector normal and north vectors

depending on the rotation angle θ and the unit vector (u, v, w) is

$$R(u,v,w,\theta) = \begin{pmatrix} u^2+(v^2+w^2)\cos\theta & uv(1-\cos\theta)-w\sin\theta & uw(1-\cos\theta)+v\sin\theta \\ uv(1-\cos\theta)+w\sin\theta & v^2+(u^2+w^2)\cos\theta & vw(1-\cos\theta)-u\sin\theta \\ uw(1-\cos\theta)-v\sin\theta & vw(1-\cos\theta)+u\sin\theta & w^2+(u^2+v^2)\cos\theta \end{pmatrix}$$

Let $\overrightarrow{a}$ be the normal vector and $\overrightarrow{b}$ the north vector of the destination connector before rotation. As stated before, these values are known. Let (d_x, d_y, d_z) and θ_d be the rotation vector and angle of the destination module, which can also be easily retrieved through the Webot API. The values after rotation $\overrightarrow{a'}$ and $\overrightarrow{b'}$ are

$$\begin{aligned} R_d &= R(d_x, d_y, d_z, \theta_d) \\ \overrightarrow{a'} &= R_d \cdot \overrightarrow{a} \\ \overrightarrow{b'} &= R_d \cdot \overrightarrow{b} \end{aligned}$$

Let $\overrightarrow{j}$ be the normal and $\overrightarrow{k}$ the north vector of the source connector before rotation. As stated before these values are known. Let $\overrightarrow{j'}$ and $\overrightarrow{k'}$ be the normal, respectively the north vector of the source connector after rotation. The normals of the two connectors $\overrightarrow{a'}$ and $\overrightarrow{j'}$ must be parallel and in opposite direction. The north vectors $\overrightarrow{b'}$ and $\overrightarrow{k'}$ must have an angle of ϕ in-between, which is the angle of connection (a multiple of $\pi/12$, according to the real module's possible angles). Therefore the north vector $\overrightarrow{b'}$ is rotated around the normal vector $\overrightarrow{a'}$.

$$
\begin{aligned}
&R_{a'} = R(a'_x, a'_y, a'_z, \phi) \\
&\overrightarrow{j'} = -\overrightarrow{a'} \\
&\overrightarrow{k'} = R_{a'} \cdot \overrightarrow{b'}
\end{aligned}
$$

The point now is to determine how $\overrightarrow{j}$ and $\overrightarrow{k}$ can become $\overrightarrow{j'}$ and $\overrightarrow{k'}$ after a rotation around an axis. In other words what is this axis and how many degrees to rotate. There are many ways to rotate a vector around an axis so that is comes to another given vector. For every axis lying in a given plane and passing through the origin, there is an angle that fulfills the equation. This planes is defined as if it were a mirror reflecting the vector as the rotated vector. One can compute the planes normals $\overrightarrow{p_a}$ and $\overrightarrow{p_b}$ as follow.

$$
\begin{aligned}
&\overrightarrow{n_a} = \overrightarrow{j} \times \overrightarrow{j'} \qquad && \overrightarrow{n_b} = \overrightarrow{k} \times \overrightarrow{k'} \\
&\overrightarrow{m_a} = \frac{\overrightarrow{j} + \overrightarrow{j'}}{2} && \overrightarrow{m_b} = \frac{\overrightarrow{k} + \overrightarrow{k'}}{2} \\
&\overrightarrow{p_a} = \overrightarrow{n_a} \times \overrightarrow{m_a} && \overrightarrow{p_b} = \overrightarrow{n_b} \times \overrightarrow{m_b}
\end{aligned}
$$

The searched rotation axis directed by $\overrightarrow{vec}$ lies in both planes $\overrightarrow{p_a}$ and $\overrightarrow{p_b}$. Thus it is defined by the intersection of the planes.

$$
\begin{aligned}
&\overrightarrow{v} = \overrightarrow{p_b} \times \overrightarrow{p_b} \\
&\overrightarrow{vec} = \frac{\overrightarrow{v}}{||\overrightarrow{v}||}
\end{aligned}
$$

The angle of rotation ϕ is measured perpendicularly to the rotation axis thus the vectors $\overrightarrow{j}$ and $\overrightarrow{j'}$ need to be projected on the rotation axis to find out $\overrightarrow{v_1}$ and $\overrightarrow{v_2}$ in-between which the angle can be computed.

$$
\begin{aligned}
&\overrightarrow{a_{1proj}} = (\overrightarrow{j} \cdot \overrightarrow{vec}) \cdot \overrightarrow{vec} \\
&\overrightarrow{a_{2proj}} = (\overrightarrow{j'} \cdot \overrightarrow{vec}) \cdot \overrightarrow{vec} \\
&\overrightarrow{v_1} = \overrightarrow{j} - \overrightarrow{a_{1proj}} \\
&\overrightarrow{v_2} = \overrightarrow{j'} - \overrightarrow{a_{2proj}} \\
&\phi' = \cos^{-1} \frac{\overrightarrow{v_1} \cdot \overrightarrow{v_2}}{||v_1|| \cdot ||v_2||}
\end{aligned}
$$

Two rotation matrices are built because the rotation can be done using the shortest angle ϕ' or the longest one $2\pi - \phi'$. The scalar product technique always returns the shortest one. The two original vectors $\overrightarrow{j}$ and $\overrightarrow{k}$ are then rotated once using the first matrix, and once using the second one, and compared with the targets $\overrightarrow{j'}$ and $\overrightarrow{k'}$. ϕ is set to the one that produce the right result and R_s to the matrix that was used to produce it.

The rotation is known i.e. the source and the destination connectors are oriented properly. It remains to face them to each other with a small space in-between to avoid physical engine discordance due to machine floating point imprecision. Let $\overrightarrow{p_s}$ and $\overrightarrow{p_d}$ be the positions of the source and destination connectors relative to the module. The absolute positions of the source connector $\overrightarrow{c_s}$ and of the destination connector $\overrightarrow{c_d}$ are computed. The translation $\overrightarrow{x}$ to apply to the source module is simply the difference between these two positions. Note that a factor k is introduced to avoid the precision problem. The value used is 0.0001, which corresponds to an offset of $100\mu m$.

$$
\begin{aligned}
&\overrightarrow{c_s} = R_s \cdot \overrightarrow{p_s} \\
&\overrightarrow{c_d} = R_d \cdot (\overrightarrow{p_d} + \overrightarrow{a} \cdot k) \\
&\overrightarrow{x} = \overrightarrow{c_d} - \overrightarrow{c_s}
\end{aligned}
$$

In order to determine the position of the front connector the position of the servomotor must be known by the supervisor controller but is available only in the concerned module's controller. The master controller should require this position using the broadcast channel, which would require an adaptation of the protocol, and including some more time lags in the whole computation, which causes – as explained in Section 5.4.3 – concurrency problems. In order to simplify the resolution of this problem, all servomotors are deactivated and set to position 0 during the dynamic connection phase. This allows to treat the front connectors the same way as any other.

5.4.3 Implementation

YaMoR Webots Bridge is based on Jerome Maye's code for *YaMoR Host 2 Webots Edition*, with optimization and LED tracking parts removed. The command line interface was replaced by a TCP/SNP interface with no user interaction at all for controlling the application.

Bypassing Webots

A problem that occured in the module positioning is that each module controller, the supervisor controller and Webots run in different processes. This is a problem for the physical dynamic connection of the modules. The module positioning must be done be the supervisor controller, whereas the connector activation must be done by one of the two modules. The connector activation must be done after the positioning. Therefore the supervisor sends a special message to the module that has to connect. But in this time interval, the physics engine still runs, which causes the module to connect to start falling down, and when the connector activates it is maybe already to far away.

A workaround that allows to improve the probability of good connection (but not to reach 100% of success) consists in freezing all the motors, position repetitively the module during half a second, tell the modules to activate their connector, keep the repetitive positioning during another half second and then unfreeze the robot.

5.4.4 Issues and Known Bugs

Dynamic Physical Connetion

As stated in Section 5.4.3, because of lacks of functions in Webots' API the physical connection of two module is not guaranteed to work. Furthermore if the user tries to connect a module where there is no place or where the module will be partially below the ground, the connection will not take place and the robot could even break up. There is no possibility of lifting up the robot with the current API, because each module has to be lifted separately, which causes the physics engine to panic. Ideally the API should offer a way of deactivating temporarily the physics engine.

Repeated Connect

A non relevant issue present in YaMoR Webots Bridge is that once the TCP connection with the connected program is interrupted the only way to reconnect

is to restart the supervisor controller.

5.4.5 Future

Further development can be made to improve the flexibility of YaMoR Webots Bridge. For instance there is no way of resetting a simulation. This can be achieved either by a real Webots *revert* or by repositioning the robot at its original position and resetting Webots' physical engine. The former requires the connected program to reconnect; the latter does not.

5.5 YaMoR Host 3

5.5.1 Analysis

Needs

YaMoR Host 2.0 was used to control the real robot. A special version of it was also able to control a Webot Simulation. YaMoR Host 3.0 must at least cover the features offered by this previous program (except for the optimization part, which is taken apart).

YaMoR Host 3.0 can be seen as a driver for the robot. It will either control the real robot or a Webots simulation without requiring the user to take care of whether she is using the real robot or not.

The application should provide a more convenient user interface than in the previous version, which was using a command line like interface. The goal here is to save time when experiencing robot gaits or configuration.

YaMoR Host 3.0 should also provide a way for other programs to drive the robot (for example, an optimizer will use this feature). A remote procedure call has been chosen so that the program that use this interface is not forced to run on the same machine. This allows optimization clustering (on several computers).

Input

The program takes as input an XML file describing the robot configuration. This includes the configuration of each module (amplitude, offset, frequency, minimum angle and maximum angle), the CPG structure (network structure, phase lag and weight for each edge) and the bluetooth network structure. This file sets up the default configuration at startup. Alternatively, another file can be loaded by the user.

The user or another program can then change any of these parameters during the execution.

Output

The program outputs the correct SNP messages either through a serial line to the real robot or by TCP/IP to the Webot simulation (YaMoR Webot Bridge, see Section 5.4).

5.5.2 Design

Architecture

Figure 5.6 provides a simplified overview of YaMoR Host 3 architecture. The graphical user interface (GUI) and the remote procedure call interface (RPC) are the upper layer of this application that allows to control the robot with a certain abstraction for example by clicking buttons or calling high level procedures. In counterpart, RS232, TCP and the LED tracking are the lower level and deal directly with the real robot or with YaMoR Webots Bridge to control a simulation. The remaining elements are part of the middle layers, which interconnect the upper and the lower layer.

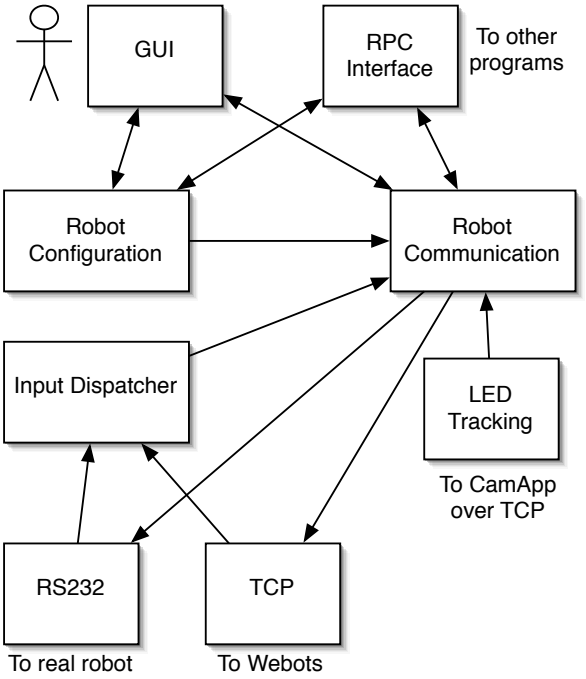


Figure 5.6: YaMoR Host 3 architecture

The robot configuration stores information about the CPG network (described in Section 6.1), which implies how the oscillators are connected together with what phase and what weight. It contains also the bluetooth network structure and the modules properties, namely the amplitude, the offset, the frequency and the limit angles for each of them.

The robot communication translates high level commands from the GUI or the RPC into SNP messages and sends them either to the simulation via TCP or to the real robot via RS232. It also updates the GUI when needed.

The input dispatcher reads in the SNP messages received from the robot and notifies the robot communication with a higher abstraction.

Threading

The following list show the tasks that have to be performed at the same time by YaMoR Host 3.

- Read and decode data sent by the real robot
- Read and decode data sent by the Webots simulation
- Read data from the LED tracking software
- Send data either to the real robot or to the Webots simulation

Each of these tasks is performed by a different thread. In order to simplify the threads interaction, sending data is done by the GUI thread which causes the graphical user interface to freeze when the program is sending data to the robot. The GUI thread is the thread that executes most of the code. It terminates automatically after the main window has been closed.

The RS232 thread is responsible for data that comes from the real robot. Currently it only signalizes to the GUI thread the income of a *connection complete* SNP message. This notification is done using a semaphore. The RS232 thread terminates when the communication port is closed.

The purpose of the TCP thread is the same as the RS232 except that instead of reading on a serial line, it reads on a TCP channel. It terminates when the TCP port is closed or if the connection is interrupted.

The LED thread continuously reads in data that comes from the LED tracking software and expose this data to any other thread that is interested. Access to the shared data is made by the C#'s *lock()* statement, which implements the concept of critical sections.

5.5.3 Implementation

Class Diagrams

Figures 5.7 and 5.8 show the class diagram of YaMoR Host 3. Figure 5.9 show the class diagram for the interfaces library and Figure 5.10 show the class diagram for the common library.

Interfaces Description

The interface *IYaMoRHost3* defines the following methods and properties:

- **ElapsedTime** Get the time elapsed since the simulation mode started. This gets the virtual time.
- **RobotPosition** Get the robot position either in Webots or using the LED tracking system.
- **SimulationModeEnabled** Tells wether the simulation mode is enabled or not.
- **AddCoupling** Add a coupling between two modules.
- **CloseSimulation** Exit Webots.

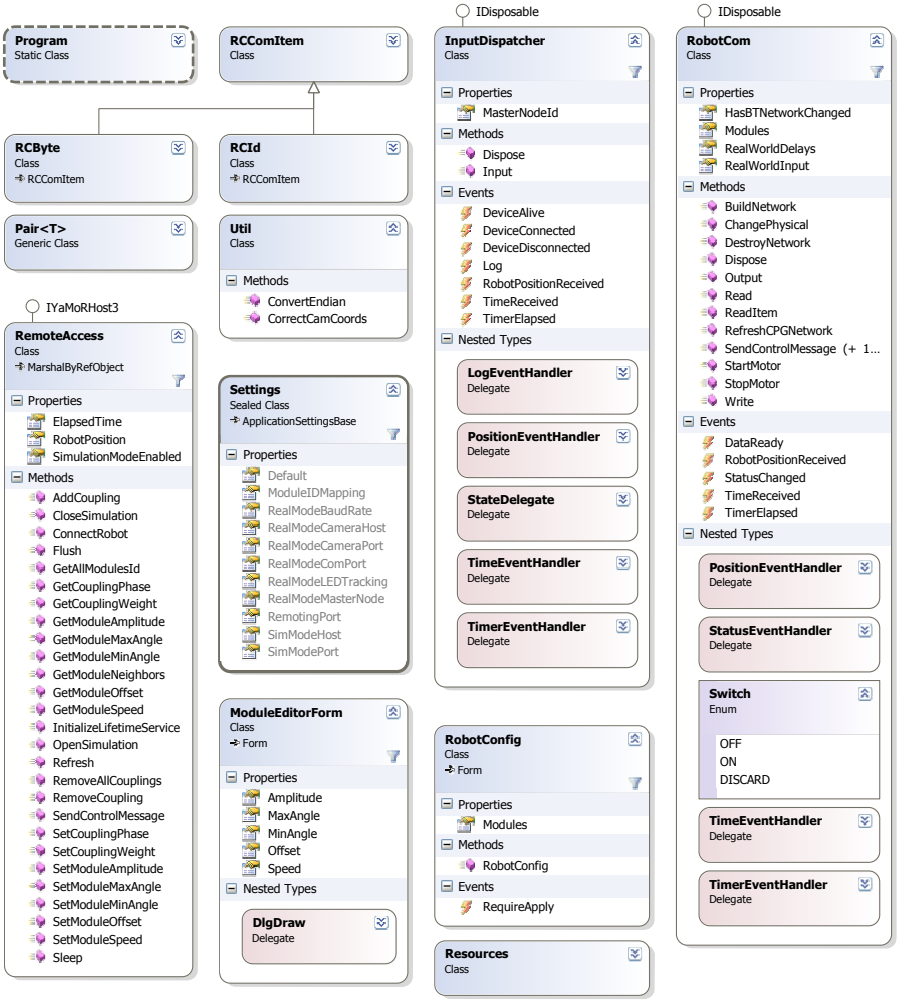


Figure 5.7: YaMoR Host 3 class diagram, part 1 of 2

Figure 5.8: YaMoR Host 3 class diagram, part 2 of 2

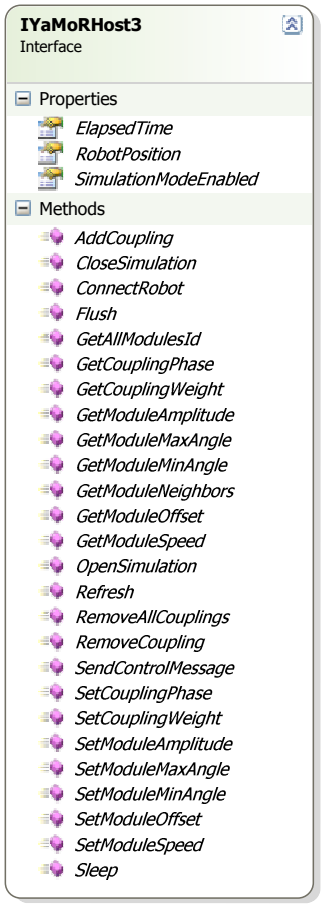


Figure 5.9: YaMoR Host 3 Interfaces Library class diagram

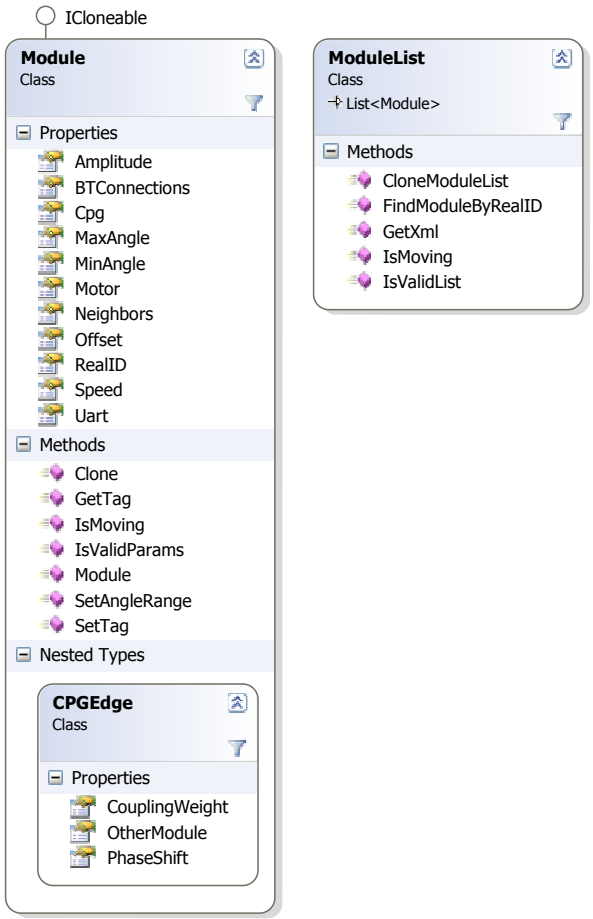


Figure 5.10: YaMoR Host 3 Common Library class diagram

- **ConnectRobot** Set up the robot's bluetooth network.
- **Flush** Send all the pending commands to the robot.
- **GetAllModulesId** Get the ids of all the modules.
- **GetCouplingPhase** Get the coupling phase between two modules in radians.
- **GetCouplingWeight** Get the coupling weight between two modules.
- **GetModuleAmplitude** Get the movement amplitude of a module in radians.
- **GetModuleMaxAngle** Get the servomotor maximal angle of a module in radians.
- **GetModuleMinAngle** Get the servomotor minimal angle of a module in radians.
- **GetModuleOffset** Get the movement offset of a module in radians.
- **GetModuleSpeed** Get the movement period of a module in Hz.
- **OpenSimulation** Open a Webots file (*.wbt).
- **Refresh** Resend all the parameters to the robot.
- **RemoveAllCoupling** Remove all the existing coupling between all the modules.
- **SendControlMessage** Send a control message to the robot.
- **SetCouplingPhase** Set the phase shift in a coupling between two modules in radians.
- **SetCouplingWeight** Set the coupling weight between two modules in radians.
- **SetModuleAmplitude** Set the movement amplitude of a module in radians.
- **SetModuleMaxAngle** Set the servomotor maximal angle in radians.
- **SetModuleMinAngle** Set the servomotor minimal angle in radians.
- **SetModuleOffset** Set the movement offset in radians.
- **SetModuleSpeed** Set the movement period in radians per Hz.
- **Sleep** Pause the calling thread a given amount of either virtual or real time depending on the activated mode.

Classes Description

The following classes were created for YaMoR Host 3:

- **Program** Contains the entry point for the application.
- **RCComItem** Represents a information that makes SNP message.
- **RCByte** Represents a byte to put in a SNP message.
- **RCId** Represents an id to put in a SNP message. Ids are translated before being actually sent, depending on wether they go to the simulation or to the real robot.
- **Pair**$< T >$ Represents a pair of two values of type T.
- **RemoteAccess** An implementation of the interface IYaMoRHost3.
- **Util** Contains utility methods.
- **Settings** Allows to access the data that must persist after the program shuts down.
- **ModuleEditorForm** The form that allows to edit graphically a module.
- **InputDispatcher** Receives and dispatches the data that comes from the robot.
- **RobotConfig** The robot configuration window.
- **RobotCom** Contains all the required methods to communicate with the robot.
- **ShapeModule** A graphical representation of the module.
- **ShapeAngle** A graphical representation of an angle.
- **MainForm** The main window.
- **AboutBox** The about window.
- **LedTracker** The LED tracking system.
- **LedTracking** A graphical user interface for the LED tracking system.
- **PhyLinkForm** The window that allow to edit physical links.
- **TristateSwitch** A button that can be leaved in three distinct states.
- **ConfigPanel** The settings window.

The following classes are part of YaMoR Host 3 common library:

- **Module** Represents a YaMoR module with its bluetooth connections, CPG links and own oscillator properties.
- **ModuleList** Represents a list of modules. It is used to represent the whole robot.

5.5.4 Issues and Known Bugs

There is a bug in the real YaMoR modules that causes the bluetooth connection setup not to work properly. When a module connects it should send an acknowledgment message back, which it for some reasons not always does. The workaround implemented in YaMoR Host 3 is waiting ten seconds instead of the acknowledgment message. This slows down drastically the connection process.

5.6 MathEval

5.6.1 Analysis

Needs

The upcoming software for gait optimization (Section 5.7) requires strings to be evaluated as mathematical expression with the possibility for the user of adding custom functions. After some unsuccessful searches for an existing and suitable library, the decision of creating a self-made library was taken.

The MathEval library should be compiled in a DLL so that it can be used in several programs. It has to be able to evaluate a mathematical expression with an integer numbers or a real numbers engine. A mathematical expression is made of values, brackets, custom function and the operators *plus*, *minus*, *times* and *divide* with their respective precedence. The custom functions should be addable without having to modify the library.

Input

The evaluator takes as input a string that represents the mathematical expression to evaluate, the desired number format (integer of real), and optionally a set of user functions.

Output

The output is the result of the evaluation in the specified format.

5.6.2 Design

Architecture

Figure 5.11 shows an overview of the MathEval library architecture.

User Functions Evaluation

User functions appear in the form *FuncName(params)* in a mathematical expression, where *FuncName* is the name of the user function and *params* is the input parameters list, values separated by commas. Note that the brackets are always required even if there is no parameters.

When the evaluator needs to evaluate a user function it first looks if the user has registered the function using *AddFunction*. If the function is not registered and the user specified an auto-reflect object, the evaluators tries to find the required function inside the object by reflection (even private methods). If it is still unsuccessful, it raises the event *RequireFunctionEvaluation*. And if this

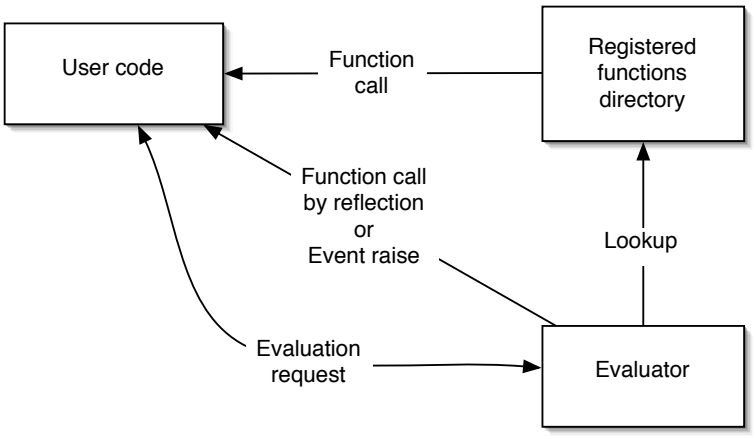


Figure 5.11: MathEval architecture

also fails then it throws an exception. For the three possible calls, parameters are passed to the function as an array of values.

Genericity

MathEval is a generic type; however only *int*, *double* and *float* are supported. The type of MathEval defines the type values in the expression are converted to, the return type of the evaluation and the type of the parameters and the return type for custom functions.

5.6.3 Implementation

Class Diagram

Figure 5.12 shows the class diagram for the MathEval library.

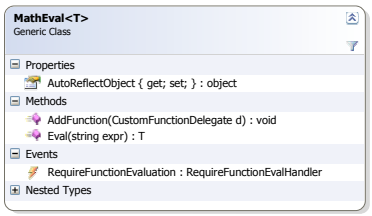


Figure 5.12: MathEval class diagram.

Classes Description

- **MathEval** is the main class of the library. It contains the evaluator and stores the registered user functions.

5.6.4 Issues and Known Bugs

There are no known issue or bug in MathEval.

5.7 YaMoR Optimizer

5.7.1 Analysis

Needs

One of the key features of YaMoR robot is its ability to learn walking by itself. Therefore an optimization program is used. As shown in the past [15] it might be relevant to be able to experiment different optimization techniques.

The role of YaMoR Optimizer is – despite its name – not to do the optimization by itself, but to provide to the actual optimizers a common an convenient way of performing optimizations. It should allow the optimizers to submit simulations. The simulations are then run by YaMoR Optimizer whenever and wherever possible.

As one does not want to recompile the whole program each time an optimizer is added or modified, the optimizers are compiled as separated libraries (DLL) that are dynamically loaded by YaMoR Optimizer, as a kind of a plugin.

The optimizers should also be able to start without user interaction, which allows to make a batch start i.e. start automatically several instances of the same optimizer with different parameters for each.

Last but not least, the program should of course use YaMoR Host 3 to interact with the Webots simulation or the real robot. Its RPC interface over TCP/IP can be used for this and allows to run simulation on any machine that has YaMoR Host 3 running.

Input

The programs require a list of host names (computer names or IP addresses) and TCP ports, which correspond to computers that are running YaMoR Host 3. This way optimizers can benefit from the parallelization of their simulations (if possible).

It also needs a list of DLL that contain an optimizer. The user can then choose within this list which optimizer she wants to load.

Output

YaMoR Optimizer should issue the correct RPC calls to run the submitted simulations on the available machines.

5.7.2 Design

Architecture

Figure 5.13 shows an overview of the YaMoR Optimizer architecture. The main component in this program is the simulation organizer. Its task is to wait for a host to be available and for a simulation to be submitted. Then it creates a simulation manager object with both. The simulation manager

has thus a simulation and a host to execute this simulation. It is responsible of executing this simulation on that host. Once it is done, it notifies the simulation organizer back and the host becomes available again for another simulation. The simulation manager is destroyed.

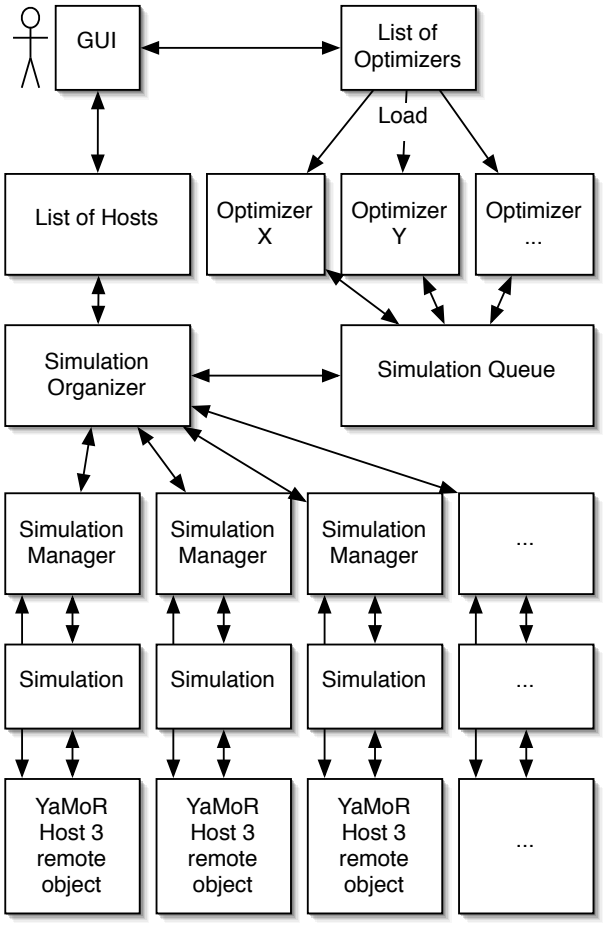


Figure 5.13: YaMoR Optimizer architecture

Threading

The base of YaMoR Optimizer runs in its own thread. When the user starts an optimizer, it is run in a new thread. Then an optimizer usually creates and runs simulations which are each executed in a separated thread. The simulation-to-execute queue and the free-hosts queue is also managed in another thread to avoid the user interface to freeze when there is no awaiting simulation or no free host. The control over the execution of a simulation is done via a returned ISyncObject. This one allows to know when the simulation execution has started and when it is finished. The synchronization for the simulation manager is made by a semaphore. Each free host adds a resource to the semaphore. Each

simulation consumes a resource and releases it when it is over.

General Optimization Configuration

YaMoR Optimizer offer a general optimization configuration file format support. There are two main parts. The first one is the robot structure with the starting values for all the parameters exactly as it was before (described in Appendix C.4). The second part describes which parameter have to be optimized and the parameters whose value are constrained depending on other parameters values. These rules are listed as direct child of the *yamorconfig* element.

To tell the optimizer that a parameter has to be optimized, an *optimize* rule has to be placed as follow:

```
<optimize param="..." min="..." max="..." />
```

The attribute *param* is the name of the parameter to be optimized followed by an underscore and the module id, for instance `"amplitude_3"`. For the parameter involving two modules the id of the second module is simply added at the end, like `"phase_2_5"`. The attributes *min* and *max* specify the bounds of this parameter.

To create constraints i.e. parameters with value depending on other ones a *contrain* rule has to be placed as follow:

```
<optimize param="..." expr="..." />
```

The attribute *param* is the same as for the *optimize* rule. The attribute expression is a MathEval expression that will be evaluated each time the optimized parameters are changed. The user can use in this expression all the parameters name as functions (so do not forget brackets) plus some predefined functions such as M2Pi which computes the angle to add to its argument to get a multiple of 2π. An example for *expr*: `"M2Pi(phase_0_1()+phase_1_2()+phase_2_3())"`.

5.7.3 Implementation

Class Diagram

Figure 5.14 shows the class diagram for YaMoR Optimizer and Figure 5.15 shows the class diagram for YaMoR Optimizer Interfaces library.

Interfaces Description

The interface *IOptimizer* is the interface that must be implemented to create a new optimizer. It defines the following method, properties and events:

- **Author** Get the author's name.
- **Name** Get the name of the optimizer.
- **UserConfig** Get the UserConfig object, which describes the optimizer's parameters.
- **Version** Get the version of the optimizer.
- **BatchStart** Start many instances of the same optimizer.

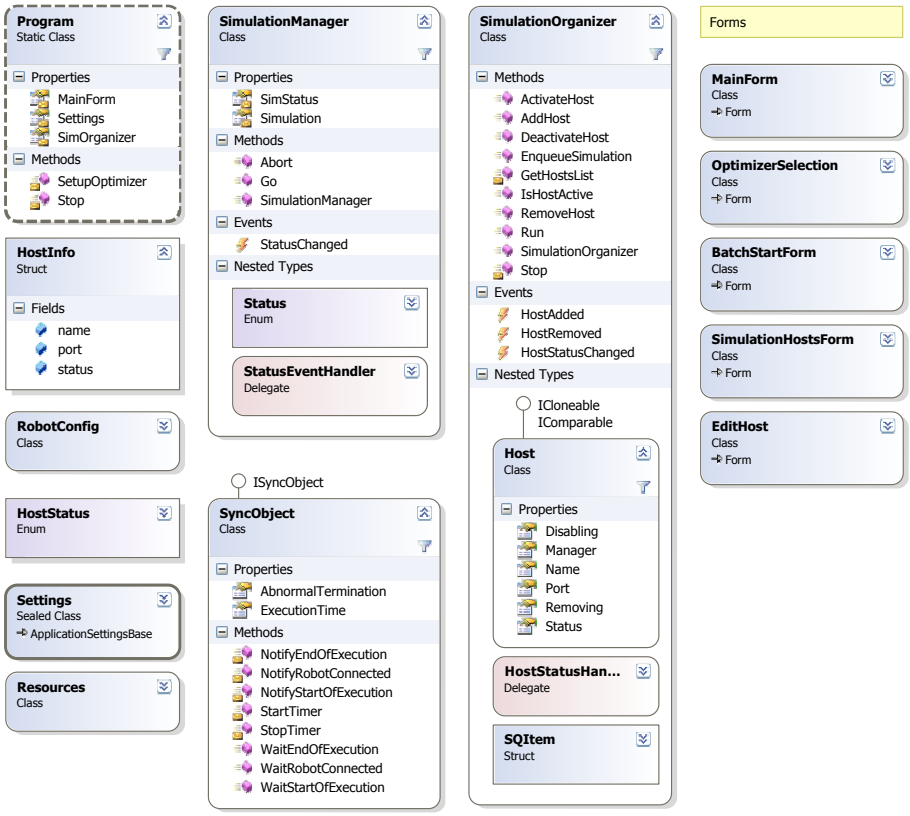


Figure 5.14: YaMoR Optimizer class diagram

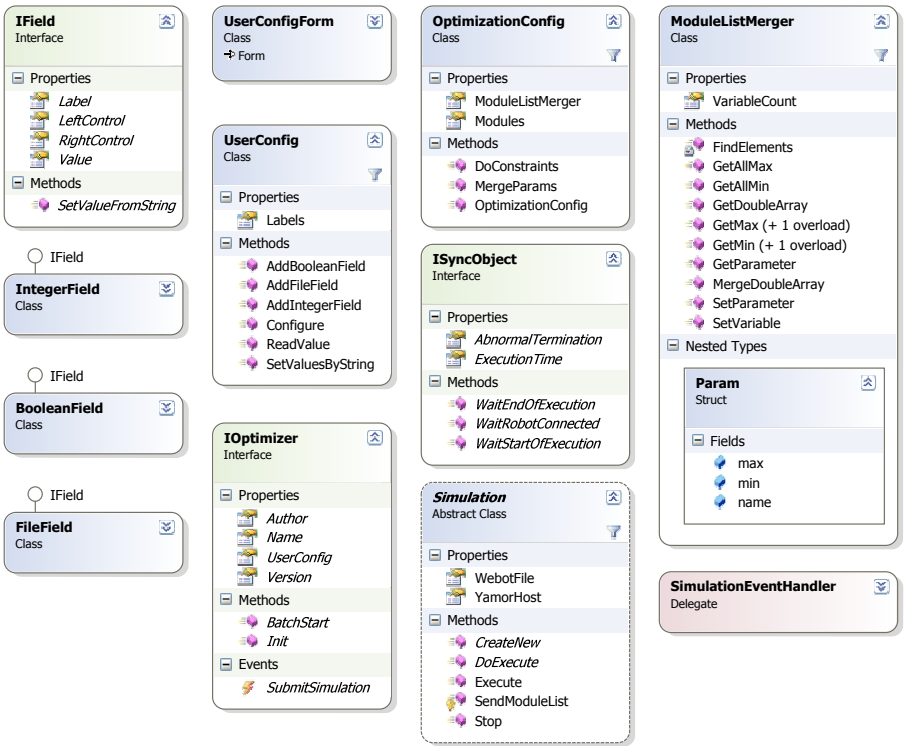


Figure 5.15: YaMoR Optimizer Interfaces library class diagram

- **Init** Initialize the optimizer.
- **SubmitSimulation** Event. The optimizer raises it when it want to submit a simulation for execution.

Classes Description

The following classes were created for YaMoR Optimizer:

- **Program** Contains the entry point of the application. Spawns the simulation organizer thread.
- **RobotConfig** *Unused.*
- **Settings** Allows to access the data that must persist after the program shuts down.
- **SimulationManager**
- **SimulationOrganizer** Manages the execution of the pending simulations on the available hosts.
- **SyncObject** Contains methods to get information about the simulation execution status.
- **MainForm** The main window (multi-document interface).
- **OptimizerSelection** The window that lists the available optimizers.
- **BatchStartFrom** A dynamic form whose fields depend on the selected optimizer. Allows to start many instances of the same optimizer at a time.
- **SimulationHostForm** The window that lists the available hosts and their status.
- **EditHost** The window that allow to edit host information.

The following classes are part of YaMoR Optimizer Interfaces library:

- **IntegerField** A field that accepts bounded integer for the UserConfigForm.
- **BooleanField** A checkbox for the UserConfigForm.
- **FileField** A field that accepts a file name and allows the user to browse the file system.
- **UserConfigForm** A dynamically generated form that allow to configure an optimizer.
- **UserConfig** Describes and store the optimizer configuration.
- **OptimizationConfig** The optimization configuration. Can read the configuration file.
- **ModuleListMerger** A class that merges the module list (the robot configuration) with an array of double describing the values of the parameters. It also does the constraints.

- **Simulation** Abstract. Represents a simulation. Optimizers that want to submit simulations must create a concrete class based on this one. Method to implement are *DoExecute*, which actually runs the simulation, and *CreateNew* to clone the simulation.

5.7.4 Issues and Known Bugs

The program does not shut down the started optimizers when it shuts down. This causes their thread to continue their execution so the process is not terminated. When the user exits YaMoR optimizer, she has to close first all the optimizers. Otherwise she can terminate the process using the Windows' Task Manager.

Chapter 6

Gait Optimization

6.1 Central Pattern Generators

This section provides a quick overview on the central pattern generators (CPG) and their application in YaMoR robot. An exhaustive description is available in Jerome Maye's master thesis "Control of Locomotion in Modular Robotics" [15].

6.1.1 Overview

"A CPG is a network of neurons, capableof producing oscillatory signals without oscillatory inputs" [1]. CPGs are present in spines and allow the brain not to care about the detail of the gait but only to send simple signals like *walk*, *run*, etc. Since all the actuator must be synchronized in a gait there are connections in-between the neurons. The CPG is robust to perturbations: if it has motor feedback and an actuator is blocked for example the whole gait is modified. When the actuator is released the gait smoothly returns to normality. The oscillators run in a so-called limit cycle and if for some constraints they leave it, they smoothly reconverge to it when the constraints are released.

A model of CPG has been developed in the robot locomotion control application field, using coupled non-linear oscillators [1]. The parameters that characterize the network are for each connection the phase shift and the weight of the connection; for each oscillator, the frequency, the amplitude and the offset. If θ_i [rad] is the oscillator set-point, ϕ_i [rad], r_i [rad] and x_i [rad] are the state variables corresponding to the phase, the amplitude, resp. the offset, ω_i [rad], R_i [rad] and X_i [rad] are the control parameters for the desired frequency, amplitude, resp. offset, ω_{ij} and φ_{ij} are the coupling weights and phases biases with oscillator j, r_j and ϕ_j are state variables from oscillator j, and a_r and a_x are constant positive gains that control the speed of convergence both set to $4rad/s$, then the oscillator i is ruled by the following equations:

$$
\begin{aligned}
\dot{\phi}_i &= \omega_i + \sum_j \omega_{ij} r_j \sin(\phi_j - \phi_i - \varphi_{ij}) \\
\ddot{r}_i &= a_r(\frac{a_r}{4}(R_i - r_i) - \dot{r}_i) \\
\ddot{x}_i &= a_x(\frac{a_x}{4}(X_i - x_i) - \dot{x}_i) \\
\theta_i &= x_i + r_i \cos(\phi_i)
\end{aligned}
$$

6.1.2 YaMoR Application

The CPGs are a very suitable way for the modular robot locomotion because each module can contain a part of the network on which his motors are directly plugged. Thus all the module can have the same code for the motor command. It is only necessary for them to know which part of the network they are, i.e. what are their direct neighbors (in the CPG network, which are not necessarily the same as the physical neighbors).

For the YaMoR robot, each active module contains one oscillator. The coupling is up to the user, but usually she wants to put CPG links at the same place as physical connections, with a coupling weight of 1. As an example, Figure 6.2 shows the CPG network for the quadruped robot.

In the YaMoR modules limit angles are also implemented, which causes the output to the servomotor to be bounded to a specified range. The CPG however does not know nor see anything about these bounds. Those were introduced to avoid module collisions, but can also be used to get output signal shapes other than sines.

6.2 Optimization

The gait of YaMoR is obtained by the optimization of a function which is the velocity of the robot (see Section 6.2.1) depending on the robot's free CPG parameters. Several optimization techniques have been tried, namely Powell, particle swarm optimization (PSO), simplex and other ones. Powell produces the best result in terms of quality of gait and time required. PSO converges more slowly but is highly parallelizable because all the particles can be computed in parallel since they do not depend on each other within one iteration. Powell is not parallelizable at all: a new simulation is generated according to the result provided by the last one. Simplex and other methods have not shown results as good as those [15] [21].

In this chapter, all the measurements are done on a quadruped modular robot [17]. Its physical structure is shown in Figure 6.1. Figure 6.2 shows its CPG network structure. The optimization configuration is detailed in Table 6.1.

6.2.1 Velocity Evaluation

The velocity of the robot is computed by dividing the distance travelled in a given time span. The distance is taken to be the shortest path between the robot position at the beginning of the time span and its position at the end of

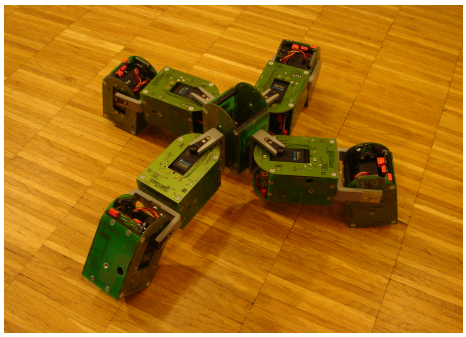


Figure 6.1: Quadruped Robot.

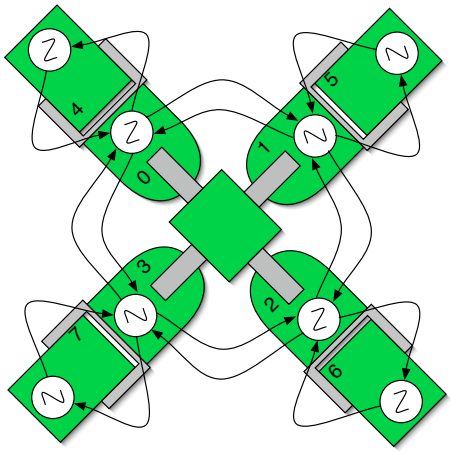


Figure 6.2: Quadruped Robot CPG Structure.

Factor	range	
Amplitude of modules 0, 1, 2, 3	$[0 : \pi/4]$	rad
Amplitude of modules 5, 6, 7, 8	$[0 : \pi/2]$	rad
Offset of modules 5, 6, 7, 8	$[0 : -\pi/2]$	rad
Phase lag from inside to outside	$[0 : 2\pi]$	rad
Phase lag from 0 to 1	$[0 : 2\pi]$	rad
Phase lag from 1 to 2	$[0 : 2\pi]$	rad
Phase lag from 2 to 3	$[0 : 2\pi]$	rad

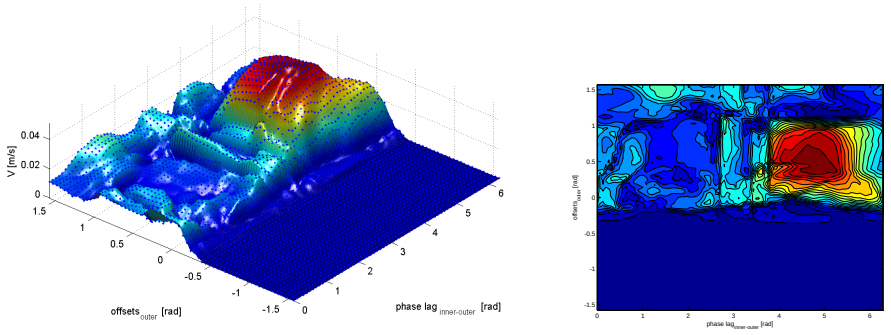
Table 6.1: Free parameters for the quadruped.

the time span. This favors straight gaits because the straight line is the shortest path between two points. If the robot would not have followed a straight path it would never have gone that far with the same velocity. The robot position in the real world is obtained by a LED tracking system with a camera. The module that carries the LED is called the *central module*.

The beginning of the time span is fixed at the eighth second. It gives the time for all the control messages (networking) to arrive at destination and for the central pattern generator to converge to some limit cycle. The second boundary is up to the user. The longer she waits, the more precise the velocity measurement will be. This is due to the noise introduced by the fact that gaits are not smooth as if the robot were rolling on wheels. For the quadruped it showed out that 12s is long enough to produce good and usable results. In the simulation the noise is however reduced by the fact that the simulation starts always with the same initial conditions.

Usual resulting gaits from the first optimizations caused the robot to walk in circle with a smaller or bigger radius. It is very unlikely that it goes perfectly in a straight gait. So it tends to go either to the left or to the right. Consider two gaits A and B. The gait A makes the robot go straight forward, whereas the gait B makes it walk hafl of a circle, but at the end it is further than with gait A. The optimization would select gait B as the best one. However, if the evaluation time is doubled, gait B will make the robot to return at its starting position (complete circle), and thus gait A will be selected. Optimizations with a longer evaluation time span produced straighter gaits.

Figure 6.3 shows an example of the velocity in function of the two out of the seven free parameters: the phase lags between inner and outer modules and the offset of the outer modules. The five other parameters listed in Table 6.1 are set to fixed value taken from a previous gait optimization. The figures are obtained by measuring 3969 (64x64)points all over the valid domain at regular intervals. A cubic interpolation is then applied to draw the surface going through these points. The walk duration is 12 seconds.



(a) (b)

Figure 6.3: Robot velocity in function of two parameters: the outer offsets and the inner-to-outer phase lags. The red zones correspond to the higher velocities whereas the dark blue ones show the lowers.

6.2.2 Powell Explained

As stated in [15], the Powell optimization is the most suitable algorithm for this application. It converges generally with few function evaluations to a maximum, which is not necessarily the global maximum. Tests showed that a good solution is found quickly in comparison to other algorithms. The Powell's algorithm works as follow:

```
Build a direction set D made of the unit vectors aligned
    to the axis of the space.
Pick a starting point ps
p = ps
Repeat
  Foreach direction d of D
    Find the function's maximum starting from p and
        following the direction d.
    Set p to be the position of the maximum found so far.
  End For
  Create a direction d1 from ps to p
  If value at ps+2*(p-ps) > value at p
    Find the function's maximum starting from ps and
        following d1
    Set p to be the maximum found so far.
    If d1 did a big current maximum improvement
      Add it to D
      Evict the direction that produced the best
          improvement so far from D.
    End If
  End If
  ps = p
Until the maximum improvement reaches a given threshold.
```

In order to find the maximum of the function following a direction one can transpose it to a one-parameter function and find the maximum using the Brent method [20] [19]. The Brent method requires the minimum to be firstly bracketed. This is done by a golden section search algorithm [19].

Unfortunately this does not take into account that the search space is bounded i.e. all the parameters have a range of possible values. At first the invalid domain was set to provide bad values as shown in Figure 6.4 so that the optimization algorithm would quickly focus on the region of interest. Knowing the bounds of the problem can lead in this case to a more clever way of doing it. Also one of the reasons that make Powell's algorithm powerful is its ability to replace some of its directions by more relevant one. Experiences on the quadruped showed that it is rare that new directions are introduced. It appeared that in most of the cases the evaluation of a new direction requested points from the invalid domain because of a to big jump, causing the direction to be obviously not taken in the set. Some modifications have been brought and are explained in Section 6.2.3, 6.2.4 and 6.2.5.

Powell's algorithm has been ported from the numerical recipes C source code [19] to C#. In order to study its good operation and behavior some optimization were done with the seven free parameters set and with the two free parameters set. The former provides a real simulation framework, whereas the latter allows to draw the powell path on the previously measured contour plot (Figure 6.3).

Figures 6.5, 6.6 and 6.7 show some of the results obtained doing the same simulation but with different starting points. In the first one the maximum is

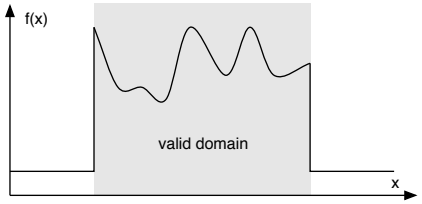


Figure 6.4: One dimensional function with constant value for the invalid domain.

quickly found, thanks to the introduction of a new direction. In the second one, the algorithm converges to a local optimum. It is very improbable to get out of such a maximum because no other better maximum exists in the directions the algorithm can move (except if some new directions have already been added). In the third one, for some reason related to the slope at the starting point plus a random factor that is part of the bracketing algorithm, the path starts in the wrong direction and gets a little lost before converging to a local optimum.

Figure 6.8 shows the optimization timeline of the seven parameters set. The upper graph shows the evolution of the seven parameters. The lower graph shows the evolution of the robot velocity. The evaluation time was $20s$ here. In this optimization no new direction was introduced. Figure 6.9 shows the same optimization but this time a new direction was added at simulation 280. It is used again at simulations 340 and 415. When all the parameters change very briefly (red arrow) it means that a new direction is being evaluated to decide wether insert it in the directions set or not.

Because of the random factor used in the bracketing in Powell's algorithm and because of the noise of the robot velocity measurement, it is very improbable that two simulations with the same initial parameters and starting point produce the same path in the solution space. Often they do not even produce the same result at all as shown in Figure 6.10.

6.2.3 Stadium Bounding

The first idea to speed up the convergence of the one-dimensional function optimization is to replace the bad values of the invalid domain by a stadium-shaped function as shown in Figure 6.11. The further the required point is from the valid domain, the worse the returned value is, instead of returning a constant value. The constant value causes the function to be discontinuous, which is suspected to slow down the convergence of the algorithm. The name "stadium" comes from the football stadium shape of such a function with two parameters (vertically reversed).

Function computation inside the valid space requires a simulation whereas an outside evaluation requires no simulation in the case of the constant bad value approach; the bad value can be returned directly. With the stadium approach, the value at the border has to be determined once so that the outside value can be computed by adding to the border value a malus corresponding to the distance to the valid domain.

The bracketing algorithm uses the same number of iterations with both ap-

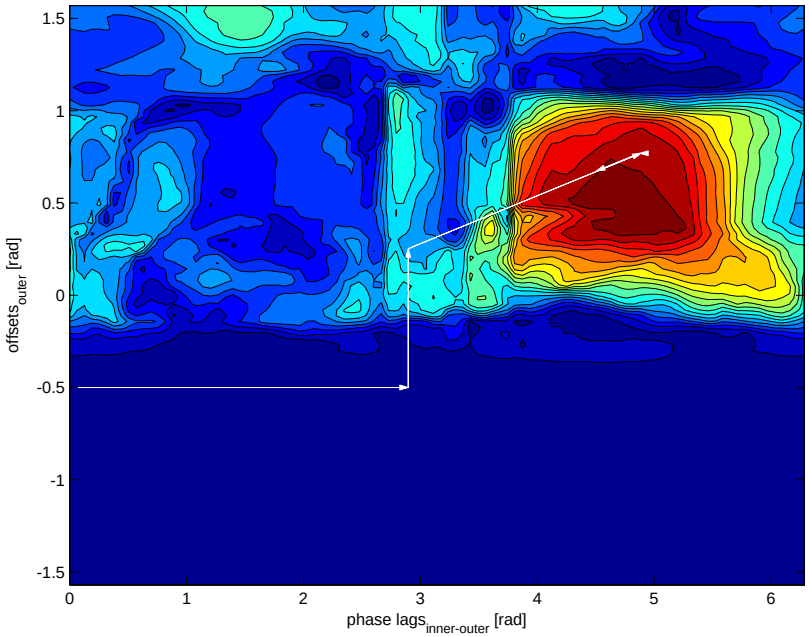


Figure 6.5: Powell path starting from $(0; -0.5)$. Direction set change after the first iteration.

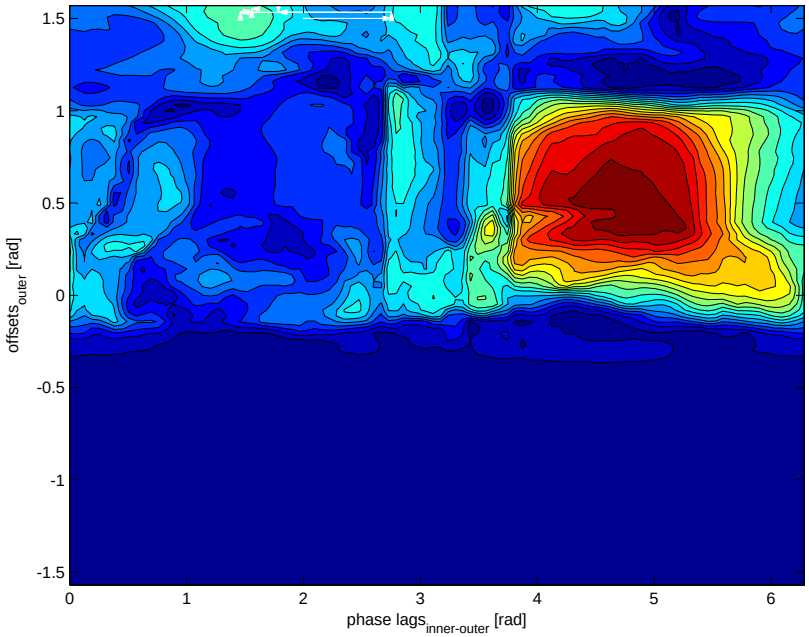


Figure 6.6: Powell path starting from $(2; -1.5)$. Converges to a local maximum.

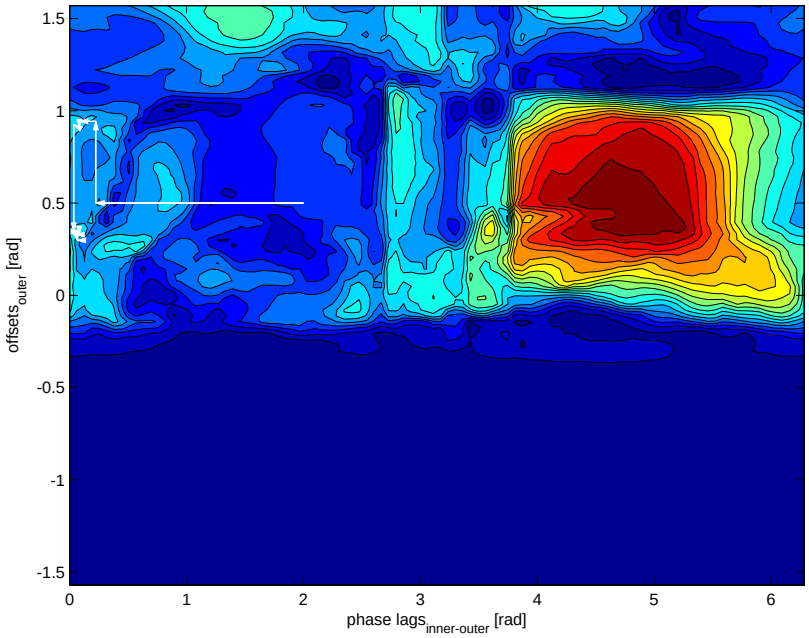


Figure 6.7: Powell path starting from $(2; 0.5)$. Starts in the opposite direction for some reason depending on the slope at the starting point. Does not recover from this "choice".

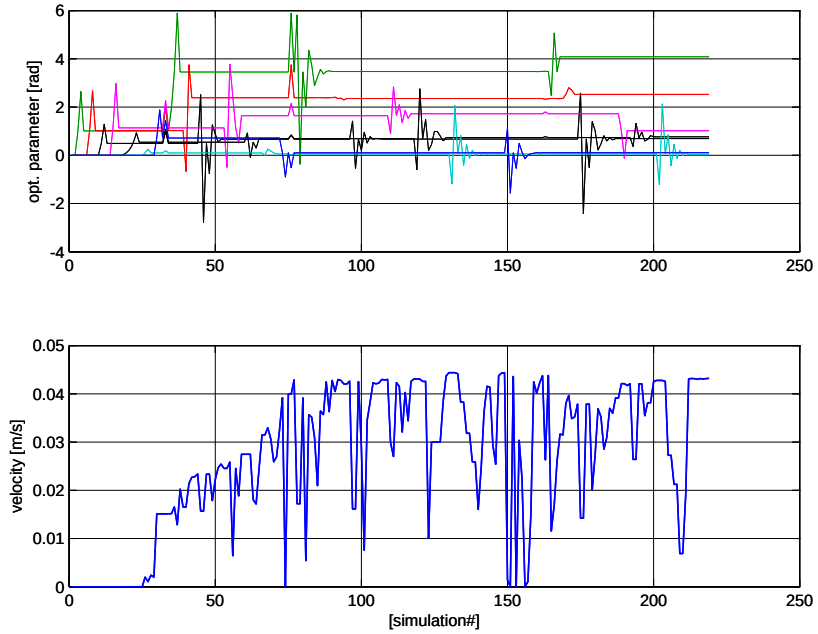


Figure 6.8: Powell optimization of the seven parameters set. No new direction was introduced.

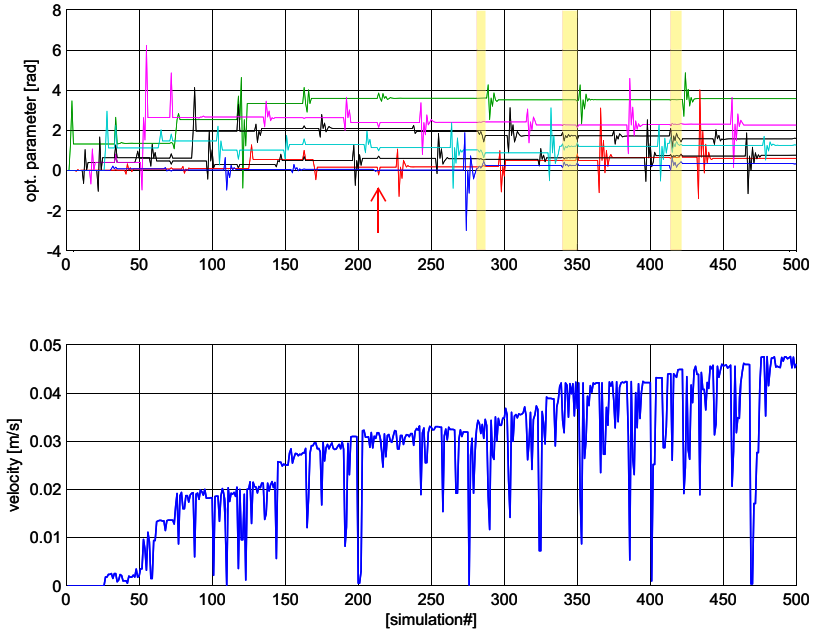


Figure 6.9: Powell optimization of the seven parameters set. A new direction was introduced at simulation 280 and reused afterwards.

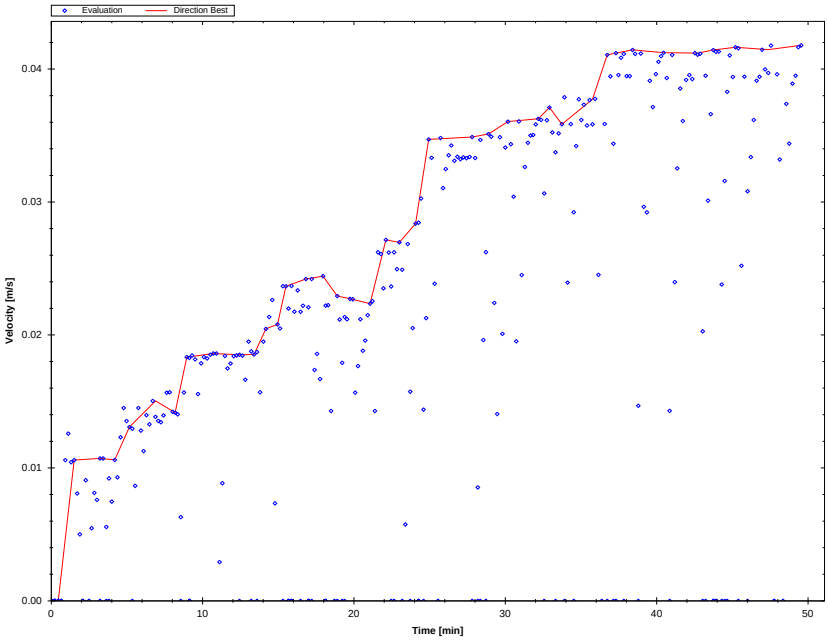
proaches. However in some cases the bracketed interval is smaller with the stadium approach. A smaller interval causes the brent method to converge slightly quicker, but it is in average compensated by the additional simulation needed to compute the stadium function. Experimentations showed no differences neither in time of execution, nor in result quality.

6.2.4 Easy Direction Change

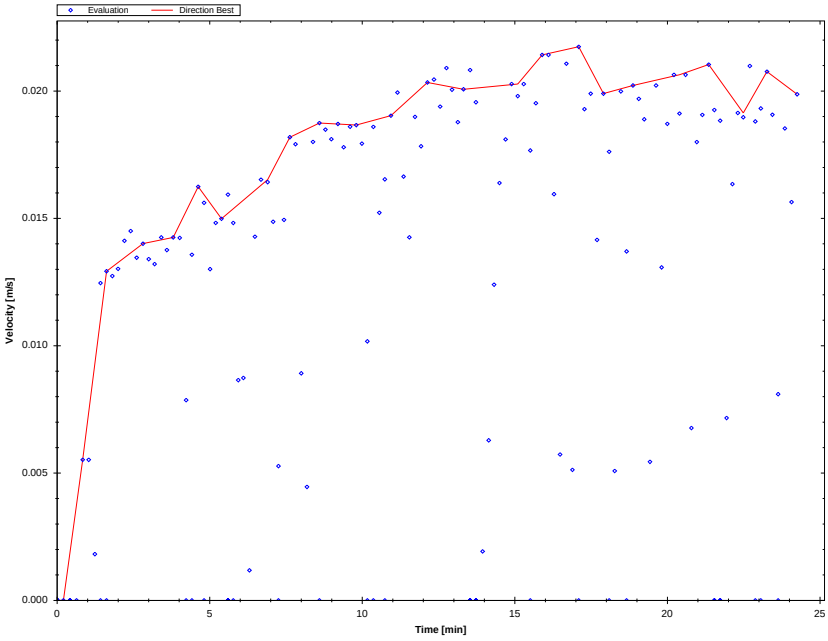
As stated before, new directions are rarely introduced by Powell's algorithm because when new directions are tried, points from the invalid domain are often requested. New directions, if relevant, allow the algorithm to converge much faster.

When an invalid point is requested, a bad value is returned making Powell's algorithm decide that the direction is not good. In order to avoid this problem, the invalid point requests are translated to the nearest valid point for the direction evaluation. It causes the new direction to be evaluated on a shorter distance but validly.

The results produced by this method are worse than the ones from the classic one. It might be due to the fact that there is a bug in the program both in PowellOptimizer plugin and in LifelongLearning plugin, which use the same code. The bug causes the result file to contain NaN values (not a number). It has not been fixed yet due to lack of time. However making the direction change easier is not necessarily a good thing. If new directions are added too easily even if they are not really good some other directions are evicted even if they are more relevant.



(a)



(b)

Figure 6.10: The same optimization configuration run two times may produce completely different results.

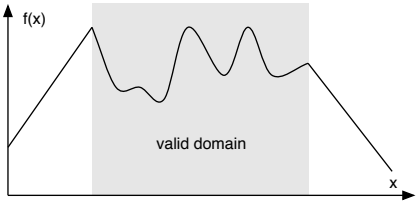


Figure 6.11: One dimensional function with gradient outside value (stadium).

6.2.5 One-Dimensional Function Optimization

The golden section search and the Brent method were replaced by a regular interval search. In order to find the maximum, the new algorithm measures ten points at regular intervals to cover the whole direction space. Then nearby the best measured point five other points are measured at smaller interval to get more accuracy.

This method is less sensitive to the local maxima because the whole direction domain is considered at each step, so it is easy for the algorithm to jump from one maximum to another one. With the classic method one has to be lucky to make such a jump because it depends of the size of the initial step, which is chosen randomly. This new method takes advantage of the fact that the bounds are known and the approximative minimal width of the maxima too. In YaMoR, changing a parameter for a few degrees does usually not cause big velocity change in both directions (adding or substracting degrees). Furthermore the 10 simulations (resp. the 5) are not interdependent and can be parallelized.

This method provides better results in most of the cases. Figure 6.12 compares the two methods running ten times an optimization on the seven parameters set each time with a different starting point. With the new method, optimizations a and b are *much* better, e and f are better, c, h and i are equivalent, d and j are worse.

6.3 From Simulation to Real World

The CPG configuration of the best gait found by Powell's algorithm for the quadruped and the seven parameters set using Webots simulations was transfered on the real robot. It could not walk. This is due to the fact that the gait caused the body to move close to the ground and because of the lag of the servomotors the robot could not lift up from the ground. However simulation and real world still have some other differences. The result of the simulations should provide a good starting point for a real world optimization since the real world and the simulated environment are not that far from each other.

6.4 Future

The technique of easily changing the direction set should be explored. In case of a high number of free parameters moving in diagonal can save a lot of simu-

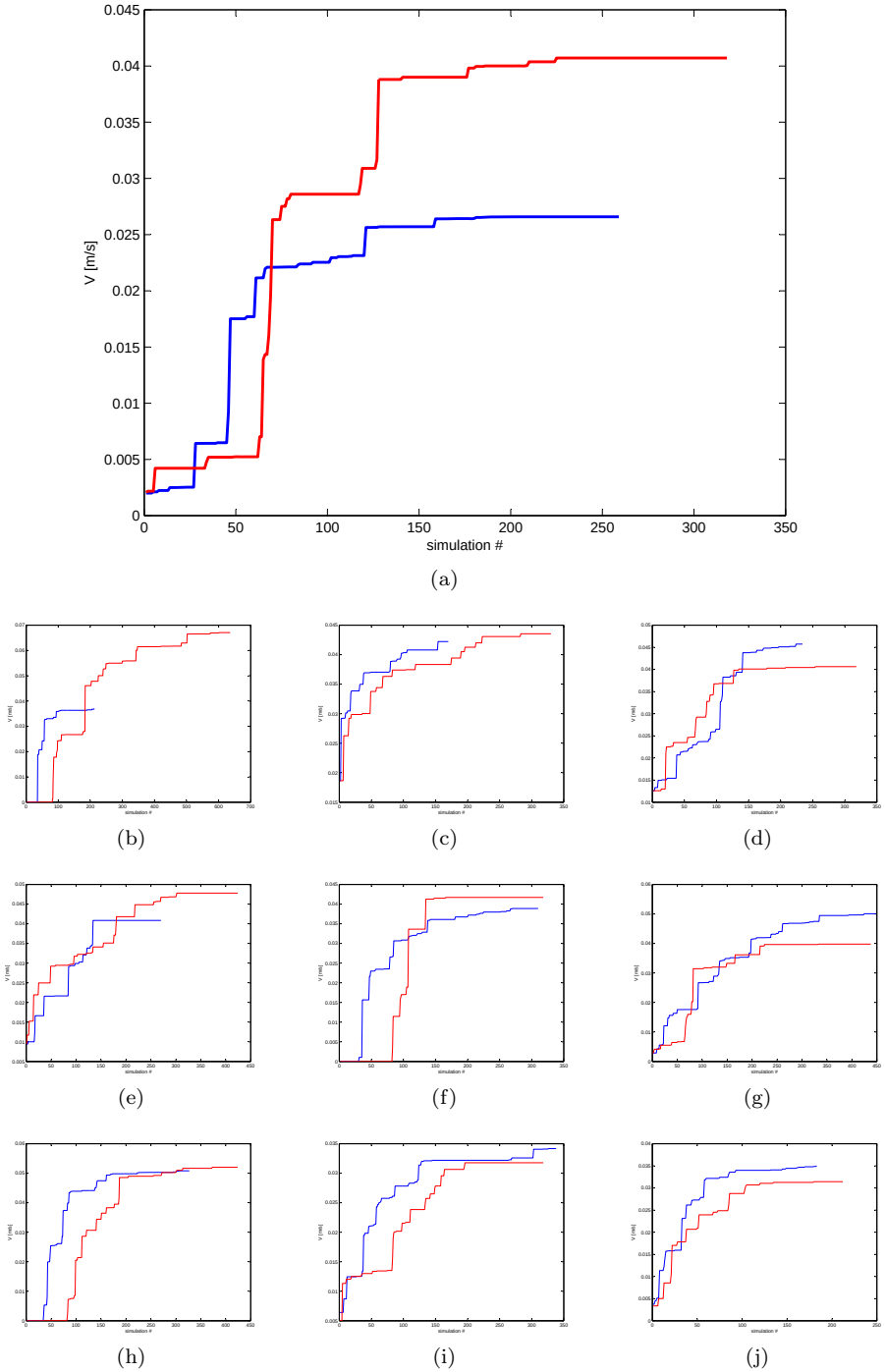


Figure 6.12: The curves show the best velocity achieved so far during an optimization. The blue one is the classic Powell method, the red one is the one with enhanced one-dimensional function minimization.

lations. However as stated before this could also deteriorate performances.

Within the set of directions, using the right direction at the right time could improve a lot the time needed for convergence. One could design a statistics-based approach for choosing the direction instead of looping through them always in the same order. It can be that some specific direction is not relevant at all during many iteration. A statistics oriented system could rearrange directions in the set so that they come up in a probable optimal order.

Chapter 7

Lifelong Learning

7.1 Overview

With lifelong learning abilities, YaMoR can learn or relearn during its life i.e. without being shut down, to adapt itself to new situations it is faced to and prevent it from normally walking. Whereas common situations like a motor getting damaged or the whole robot turning upside down can naturally happen, more specific ones are also considered, such as spontaneous module insertion or removal. Because one of the goals of modular robotics is to give robots the ability to change their shape, YaMoR must be able to recover from or take advantage of such situations.

To simplify the task it is assumed that the Bluetooth network is always connecting all of the modules present and even the malfunction of a module has no influence on the Bluetooth network.

7.2 Base Strategy

In order to give YaMoR lifelong learning abilities, a plugin was developed for YaMoR Optimizer (see Section 5.7). It consists in the following main program:

```
1 Load a YaMoR robot (Webots)
2 Load a walking gait
3 Repeat
4   Make gait statistics (velocity measurement)
5   Wait until anomaly detection
6   Adapt to the anomaly (relearn)
7 Loop
```

The velocity estimation, the anomaly detection and the relearning process are described in the following sections. In order to focus mainly on the gait adaptation, the bluetooth network is assumed to be always fully operational i.e. all involved modules can communicate together.

7.3 Anomaly Detection

There are many types of anomaly that can occur during the life of YaMoR. The following ones are identified:

- **Motor defect.** When a motor is damaged it has either no force anymore or completely stuck, whereas the software, i.e. the CPG node, continues to work normally.
- **Module defect.** The software or the battery is down. The CPG node does not work anymore.
- **Turning upside down.** The robot cannot move anymore because it is not in its normal orientation.
- **Ground material change.** The ground becomes more slippery or more sticky.
- **Module insertion.** Some modules were added to the robot.
- **Module removal.** Some modules were removed from the robot.

Any of these anomalies result in a velocity change of the robot, either better or worse. If the velocity gets increased a relearning is maybe not worth because it takes time and the velocity has already been improved. However it could lead to an even better gait. Anyway in the case of a velocity decrease, a relearning should be performed. When the physical structure is changed (module addition and removal), it is assumed that YaMoR is aware of it. Even if it is actually not, the upcoming Roombots is designed to know about physical changes. Also in this case a relearning might be thrown.

The accelerometers may also be used to detect an anomaly. If the robot is upside down for example, the accelerometers are well-suited to detect this. They could also observe a change in a periodic signal they usually measure. However they are not used for the moment because velocity measurement and physical changes detection were sufficient for this project and because using the accelerometers would have required of a software adaptation of the YaMoR, which is time costly.

Velocity is unfortunately hard to measure in lifelong learning because of the noise. The robot movement is not a straight movement with constant velocity but some quasi chaotic movement that repeats at each CPG oscillator period. If the distance travelled during one period would be measured and divided by the period time, this would give a good velocity estimation. However, the period time is not exactly the same from one period to the other probably because of Webots bad timings and the operating system's process scheduler. Moreover the distance travelled by the robot in one period is not always the same because of various sources of noise. These two imprecisions cause the velocity measure to vary a lot (tested only in Webots).

The velocity is measured during a time span of three periods to reduce the noise introduced. Be v the current stored velocity, v^+ the last measured one and t the anomaly threshold. If $|v - v^+| < t$ then $v := 0.9v + 0.1v^+$. Otherwise the anomaly is signaled. This equation allows the velocity to be subject to small variations along the robot's life.

7.4 Adaptation

If the CPG structure requires no changes a Powell optimization can be started directly. It is convinient to start the optimization from the current gait's param-

eters set because often the new optimum is not that far away from the old one. However, when a module is added, removed or damaged (CPG node not working anymore) the CPG network needs adaptation. The next section describes a method to auto-generate a CPG network from the modules physical connections graph. The assumption is made, that there is a CPG coupling wherever there is a physical connection. The following questions arise now: how to build the CPG network? What parameters are free and which ones depend on others? Which parameters are to be optimized and which ones are fixed?

7.4.1 CPG Network Construction

The base data that the algorithm has is the physical connections graph. The vertices represent the modules and the edges the physical connections. Figure 7.1 shows on the left a physical graph. Be G the physical graph of the available modules and H the CPG network directed graph, that contains the same vertices as G. The bigger connected subgraph g is extracted. This causes the algorithm to ignore detached modules. Be A and B two vertices in g and A' and B' the two corresponding vertices in H. If there is an edge between A and B then there is an edge from A' to B' and from B' to A', with a phase shift of 0 and a weight of 1. Figure 7.1 shows on the right the resulting graph H of G.

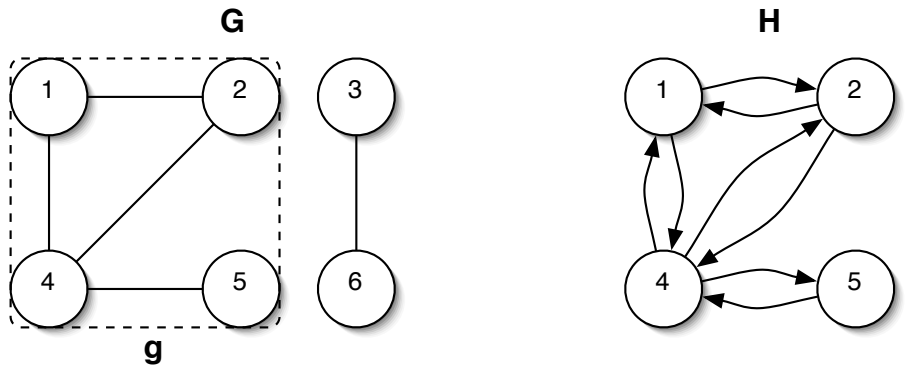


Figure 7.1: Example graphs. Left: physical graph G and bigger connected subgraph g, right: CPG graph H.

7.4.2 Constraining

The next step is to establish constraints for the phase shifts. The phase shifts of each loop in the CPG network must sum up to a multiple of 2π. If each phase shift is variable and each loop is an equation using those variables, one can compute how many phase shifts are constrained and how many are free (note that not each loop provides a valuable equation; some of them are redundant).

Find Paths

A function $Path(G, A, B)$ that finds all the paths from A to B in a graph G is defined. It returns all the paths i.e. possible lists of vertices traveled from A to B, including A at the beginning and B at the end. The algorithm used is a recursive graph walkthrough.

Find Loops

A function $Loop(G, A)$ that finds all the path starting from A and ending at A is defined. It returns all the loops i.e. lists of vertices traveled from A and back to A. It runs $Path(G, A, A)$. The function $Loops(G)$ that finds all the loops in G is defined. For all vertices A in G it runs $Loop(G, A)$. The resulting loops are then put together and filtered to remove redundancy; for example the loop 1-2-3-4-1 is the same as 3-4-1-2-3.

Constraints Algorithm

The constraints algorithm produces a list of all the edges with the sum description required for the phase shift computation. The value of the edge's phase shift must be added to this sum so that total is a multiple of 2π. These constraints can then easily be translated to MathEval expressions (see Section 5.6). It is also possible to provide a list of edges the user would like not to be dependent on other edges, i.e. unconstrained.

```
loops = Loops(CPG)
cn = an empty dictionary

Repeat endless
  fix = False

  // Try to look in the keep-non-constrained-list
  For each edge (A,B) In keepNonConstrained
    If cn not contains key (A,B) then
      cn[(A,B)] = 0;
      fix = True
      Exit For
    End If
  End For

  // Pick any of them
  If fix = False then
    For each loop L In loops
      For i = 0 to length(L) - 2
        A = L[i]
        B = L[i+1]
        If cn not contains key (A,B) then
          cn[(A,B)] = 0;
          fix = True
          Exit For
        End If
      End For
      If fix = True then Exit For
    End For
  End If

  If fix = False then Exit Repeat

```

```
34    // Look for new constraints
35    Repeat
36      changed = False
37      For each loop L In loops
38        nfree = 0;
39        For i = 0 to length(L) - 2
40          A = L[i]
41          B = L[i+1]
42          If cn not contains key (A,B) then
43            nfree = nfree + 1
44            free = (A,B)
45          End If
46        End For
47
48        // Only one unset edge in the loop
49        // Must constrain it.
50        If nfree = 1then
51          newCn = empty list
52          For i = 0 to length(L) - 2
53            If (L[i],L[i+1]) <> free
54              Add (A,B,-1) to newCn
55              // Expand if depending on other
56              // constrained variables
57              Repeat
58                expanded = False;
59                For each tuple (A,B,C) In newCn
60                  expansion = cn[(A,B)];
61                  If length(expansion) > 0
62                    newCn.RemoveAt(i);
63                    For j = 0 to length(expansion) -1
64                      edge = (expansion[j][0],
65                        expansion[j][1],
66                        expansion[j][2] * C );
67                      Insert edge In newCn at position i
68                    End For
69                    expanded = True;
70                    Exit For
71                  End If
72                End For
73              While expanded = True
74              cn[free] = newCn
75              changed = True;
76            End If
77          End For
78        End If
79      End For
80    While changed = True
81  End Repeat
```

Figure 7.2 shows the constraints algorithm run on the CPG sample graph shown in Figure 7.1.

Parameters Fixation

The output of the constraints algorithm contains a lot of free parameters (not depending on some others). Furthermore, the amplitude and offset for each module become also free. The weight for all edges and the frequency and the limit angles for all modules are left unchanged. However the total amount of free parameters might be too high for the optimization i.e. too much time would be required to get a good gait. In order to limit the number of free parameters,

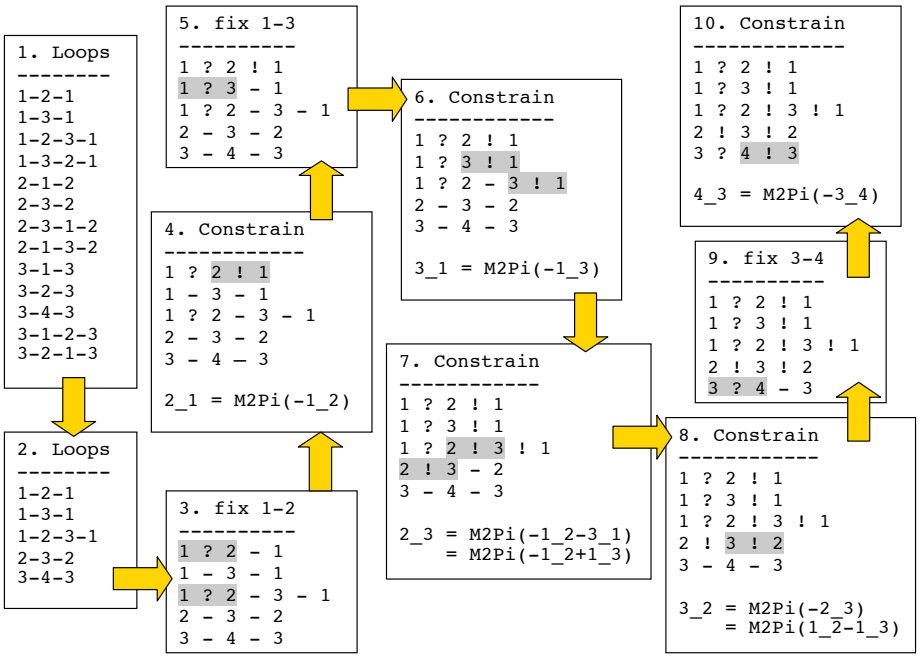


Figure 7.2: Constraints algorithm run example

a given number of them are picked randomly and set to a random value.

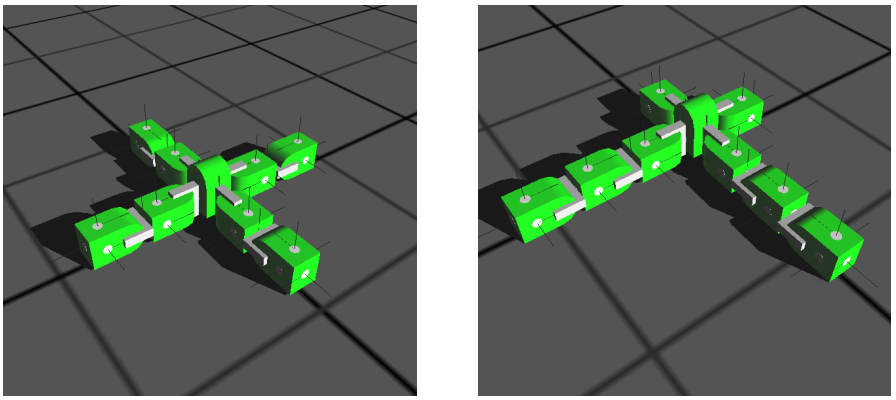
7.5 Tests

7.5.1 Transformation

As a test a transformation script was created. It shortens two legs of the quadruped and put the removed modules on the end of the other two legs (see Figure 7.3).Webots is started with the quadruped reproducing the best gait it learnt with the seven parameters set (see Table 6.1). The robot walks for a few minutes. Then the transformation script is executed. At this moment the velocity is expected to decrease. The robot should notice the structure change and start the optimization process until the gait becomes good again. Before doing the optimization it first generates a new CPG network with 7 free parameters (not the same parameters as before because the network is transformed). Limit angles are kept.

7.5.2 Upside Down

Another test was to make the robot fall from a table from a height such that it lands upside down. Webots is started with an extended legs version of the quadruped so that it is possible for it to walk upside down. The initial gait is the best gait learnt by the regular quadruped using the seven parameters set. The four new modules are neither actuated nor part of the CPG network. The robot



(a) (b)

Figure 7.3: The quadruped robot before and after transformation.

walks on the table and then falls down. It should notice a velocity decrease and trigger an gait optimization. It is expected that it develop the ability of walking upside down. This test does not require a CPG network structure change.

7.6 Results

Figure 7.4 shows the robot velocity in function of the time. The initial velocity is about $5cm/s$. The transformation takes place at the fourth minute and is visible in the graph through a velocity decrease as expected. A new CPG network that corresponds to the new physical structure is generated with 7 free parameters. The relearning process is then started and after $80min$ the robot is able to walk again with a velocity of about $4.5cm/s$. A movie of this adaptation is available on the enclosed CD-ROM.

Figure 7.5 shows the recovery process after having fallen upside down. These are screenshots from the movie that can be found on the enclosed CD-ROM.

7.7 Discussion

The results obtained and presented in Section 7.6 show that the developed lifelong learning strategy works very well for the implemented experimental scenarios. However because of the CPG network free parameters limitation, some of the parameters are randomly picked and set to random values. Because of this not only achieving the optimal gait is probably impossible but also a suitable gait may be impossible to find depending on which parameters remain free. This problem does not occur when the CPG structure is not required to be changed.

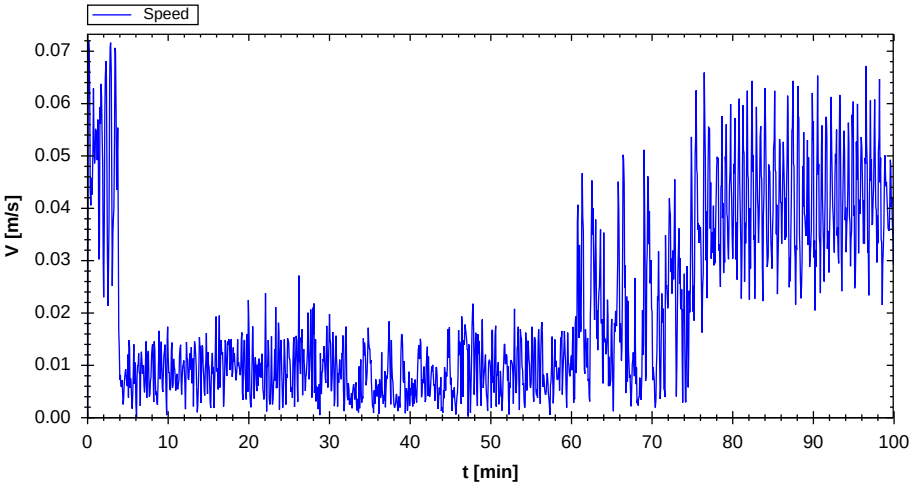
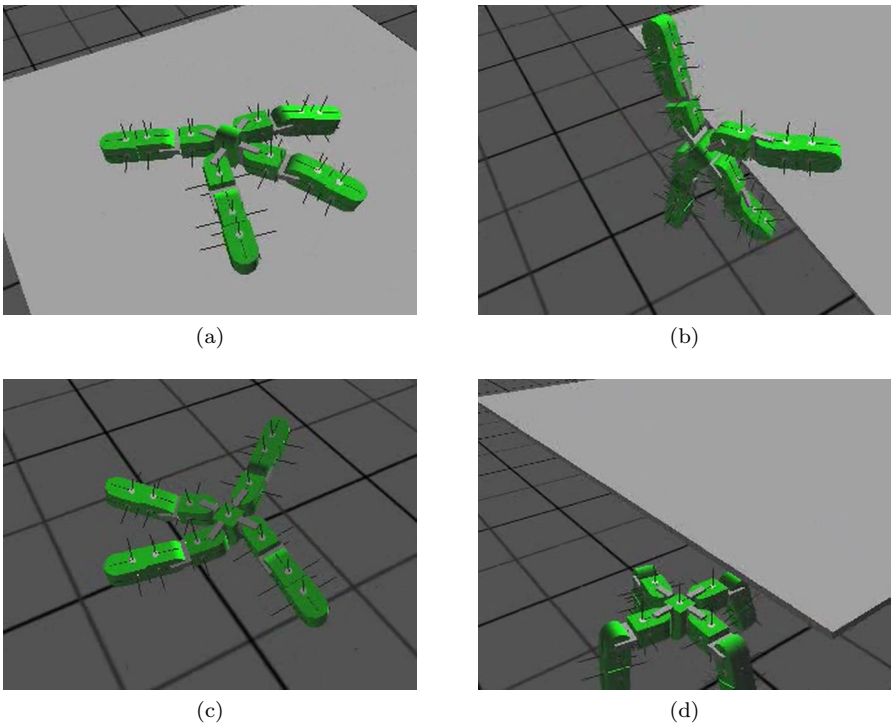


Figure 7.4: Quadruped velocity evolution upon sudden transformation.



(a) (b)

(c) (d)

Figure 7.5: Quadruped recovery after fall.

7.8 Future

7.8.1 CPG Network Construction

As stated in the discussion, the CPG structure change may cause a walking gait to be unreachable. The CPG network construction algorithm should be more investigated, particularly to detect symmetries. Symmetries are a simple way to reduce the number of free parameters. They allow to replace free parameters by constraints that depend on other free parameters. The seven parameters set used for the quadruped had typically a lot of symmetries introduced by hand. The CPG network construction algorithm should be able to automatize this task. Limit angles are however always set by hand.

7.8.2 Gaits Database

For all the abnormal situations that occur which do not require a CPG structure change or involve a defect motor i.e. the situations where the environment or the robot orientation is responsible for a velocity decrease, a gait database can be built for each along the robot's life. For example when it comes up to a slippery ground, it relearns the walking gait but without forgetting the old one that was good on regular ground. When it comes back to normal ground it can retake the old gait without having to run a time-costly optimization. The database would contain records made of the following information

- Parameters values for the CPG.
- When was the gait used for the last time.
- What gaits were used before this one and what were the symptoms of the anomaly (described later) for each of these gaits.
- The robot orientation for the gait.
- The expected velocity with this gait.

This is a non exhaustive list of the information that could be kept in the database. The gaits used before this gait and when the gait was used for the last time can provide statistical information that could be used for the new gait selection. The symptoms that occurred in each of this gait (such as accelerometer signals, velocity decrease, values of the light sensors) when the anomaly was detected allow to attribute a likeliness value to each of the gait of the database so that they can be tried out in the most efficient order. If none of the database gaits allows to reach the expected velocity, then a new optimization is made and the new gait is stored in the database.

Chapter 8

Conclusion

This work showed that the accelerometers signal of the YaMoR modules cannot be used to evaluate the robot position. The only option at the moment is to track the robot using a camera. However, accelerometer data can still be used for other purpose such as determining the orientation of the robot or detecting impacts. The light sensors present in each module could be used too to determine if the module is facing the ground for example.

The software developed during this project allows to work with YaMoR, focusing only on robotic problems with no need to worry about programing issues. It has already been proved to be very helpful in this project, especially during the gait optimization phase and the long life learning phase. The author did not need to worry anymore about the way data is transported to the real robot or the simulation or how the computer cluster is going to parallelize the requested simulations. He just had to use the convenient high-level application programming interface offered by YaMoR Optimizer, which itself uses YaMoR Host 3 – a program that can be seen as a kind of driver for YaMoR, with an additional graphical user interface.

The gait optimization algorithm (Powell) used in Jerome Maye's work [15] was improved by taking into account that the gait parameters are bounded. This had not been considered before. Now it finds better gaits or converges faster in most cases. However some more work should still be done in this field. As a matter of fact, a techique imagined to reduce the time needed to find a good gait could not be experienced properly because of a bug in the software discovered at a too advanced stage of this project (see Section 6.2.4).

The lifelong learning abilities allow robots to overcome unusual situations that occur along their life. In this project, efforts were put in always being able to walk. The robot is subject to standard anomalies such as motor failure, being turned upside down or arriving on a slippery ground. Since it is modular and the goal is eventually that it changes its shape by itself, it is also subject to module addition and removal, which interferes with the walking. That is why the CPG network used for the gait generation can be created automatically.

A system of a gaits database was also imagined, enabling the robot to remember gaits it used and to reuse them when it feels that the situation currently faced has already occurred in the past. The gaits database could spare the robot having to relearn gaits it already found once from scratch and, by this, save several hours of optimization.

The optimized objective function could be changed to make YaMoR learn other types of gait, such as turning left and right or flip itself if upside down. Afterwards, by adding more sensors to the modules, it could have a behavior more complex than simply walking straight forward. One can imagine for example obstacle avoidance, like the Salamandra Robot also developed at the BIRG, which is also a modular robot working with CPGs but not reconfigurable.

The author feels a little frustrated by the fact that he did not have more time to explore this cutting-edge topic, but the framework software took a large part of the time allowed for this work and was really a must-have before going on in lifelong learning.

Future work is described in Sections 5.4.5, 6.4 and 7.8.

Chapter 9

Acknowledgement

The author would really like to thank the following persons:

- **Alexander Sproewitz** who assisted him along this work. He introduced the author to YaMoR hardware. He was always available to answer questions. He pre-read this report a brought valuable comments. He did Matlab programs to display some of the results the author got.
- **Auke Jan Ijspeert**, responsible for the BIRG, who proposed this project in his laboratory.
- **Beat Hirsbrunner**, official supervisor for this work at the University of Fribourg.
- **Jerome Maye**, who did the work on which this one is built and helped the author to get started with his code.
- **Ivan Bourquin**, who provided valuable help about Webots simulator.
- **Alessandro Crespi**, last but not least, for his computer technical support.

Also he appreciated the fact that the University of Fribourg and the Swiss Federal Institute of Technology of Lausanne can collaborate for such projects.

Bibliography

[1] J. Buchli, A.J. Ijspeert, "Distributed central pattern generator model for robotics application based on phase sensitivity analysis." In A.J. Ijspeert, M. Murata, and N. Wakamiya, editors, Biologically Inspired Approaches to Advanced Information Technology: First International Workshop, BioADIT 2004, volume 3141 of Lecture Notes in Computer Science, pages 333-349. Springer Verlag Berlin Heidelberg, 2004.

[2] M. Yim, W.-M. Shen, B. Salemi, D. Rus, M. Moll, H. Lipson, E. Klavins, G. S. Chirikjian, "Modular Self-Reconfigurable Robot Systems", 1070-9932/07 IEEE, 03/2007.

[3] T. Fukuda, S. Nakagawa, Y. Kawauchi, and M. Buss, "Self organizing robots based on cell structuresCEBOT", in Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS), Nov. 1988, pp. 145150.

[4] http://birg.epfl.ch/page65721.html

[5] http://unit.aist.go.jp/is/dsysd/mtran3

[6] P. White, V. Zykov, J. Bongard, and H. Lipson, "Three Dimensional Stochastic Reconfiguration of Modular Robots", Computational Synthesis Laboratory, Cornell University, Ithaca, NY 14853

[7] http://ccsl.mae.cornell.edu/research/selfrep

[8] Dr. Wei-Min Shen, "Modular, Multifunctional and Reconfigurable SuperBot", Information Sciences Institute, University of Southern California

[9] http://www.isi.edu/robots/superbot.htm

[10] Behnam Salemi, Mark Moll, and Wei-Min Shen, "SUPERBOT: A Deployable, Multi-Functional, and Modular Self-Reconfigurable Robotic System", Information Sciences Institute, University of Southern California

[11] Adamo Madalena, "Sensors Board, Reference Guide 2.0", Biologically Inspired Robotics Group, Swiss Federal Institute of Technology of Lausanne, 01/2006.

[12] Adamo Madalena, "YaMoR II", Biologically Inspired Robotics Group, Swiss Federal Institute of Technology of Lausanne, 01/2006, master thesis.

[13] Kurt Seifert and Oscar Camacho, "Implementing Positioning Algorithms Using Accelerometers", Freescale Semiconductor Inc, AN3397 Rev 0, 02/2007, technical report.

[14] Josh Bongard, Victor Zykov, Hod Lipson, "Resilient Machines Through Continuous Self-Modeling", Science, 10.1126/science.1133687, 11/2006

[15] Jerome Maye, "Control of Locomotion in Modular Robotics", Biologically Inspired Robotics Group, Swiss Federal Institute of Technology of Lausanne, master thesis.

[16] A. Sproewitz, R. Moeckel, J. Maye, A. J. Ijspeert, "Learning to move in modular robots using central pattern generators and online optimization", Biologically Inspired Robotics Group, Swiss Federal Institute of Technology of Lausanne, under review.

[17] A. Sproewitz, Y. Ye, X. Zhang, "Towards Gait Optimization on a Quadruped Modular Robot: Design of Experiments", Biologically Inspired Robotics Group, Swiss Federal Institute of Technology of Lausanne, 05/2007, report.

[18] Rico Mockel, "Bluetooth Scatternet Protocol (SNP), User Guide", Biologically Inspired Robotics Group, Swiss Federal Institude of Technology of Lausanne, Ver 1.0, 03/2007, technical report.

[19] W. H. Press, B. P. Flannery, S. A. Teukolsky, W. T. Vetterling, "Numerical Recipes in C", Second Edition, 1992.

[20] R. Brent, "Algorithms for Minimization without Derivatives", NJ: Prentice-Hall, 1973.

[21] Willson Sudarsandhari Shibani and Klaske Van Heusden, "Comparing different optimization algorithms for optimizing locomotion of a modular robot system", 2007, Report.

Appendix A

Experimental Configurations

A.1 Snake

The snake robot is made of eight active modules plus one inactive as shown in Figure A.1. The CPG network as simple as it can be connects bidirectionally the physical neighbors. The weight of each connection is 1.0; the phase shift of a connection going from left to right is $\pi/2$ and $-\pi/2$ for a connection going from right to left.

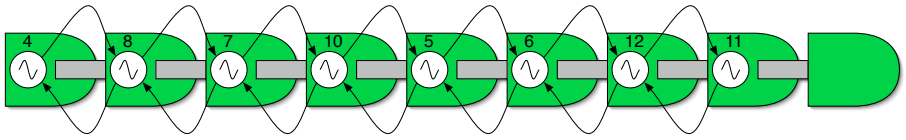


Figure A.1: Snake robot setup

Those values will make the snake undulate with a wavelength equivalent to the length of a chain of four modules. This choice is made because a module is able to lift itself plus two other. Trying to lift more could cause damages to the servomotor.

The parameters of the modules are the same for each module. The offset is 0, the amplitude has been empirically set to $0.6rad$ and the frequency has been arbitrarily set to $0.6rad/s$.

A.2 Trebuchet

The trebuchet robot has three active modules and one inactive one. The CPG network shown in Figure A.2 connects bidirectionally the physical neighbors. If the phase shift from a module a to one of its neighbors b is ϕ then the phase shift from b to a is $-\phi$. All the connection weights are set to 1.0.

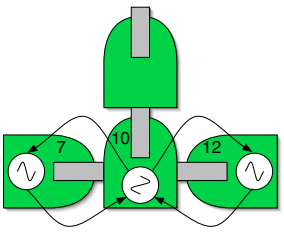
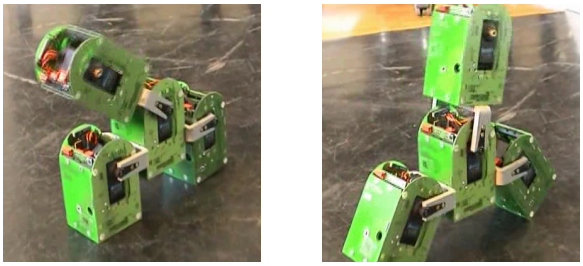


Figure A.2: Trebuchet robot setup



(a) (b)

Figure A.3: Snapshots from the Trebuchet video.

The phase shifts are set such that the two "legs" (modules 7 and 12) move in anti-phase so that they open and close. The top module makes a brutal velocity change when the legs open, which causes the robot to move forward. All the non sinusoidal signals are created using the limit angles of the modules, which cause the servomotor to block brutally if trying to go over the limits. All the values have been determined empirically and are detailed in Table A.1. Snapshots from the trebuchet video are shown in Figure A.3.

Module	Ampl $[rad]$	Offset $[rad]$	Freq $[rad/s]$	Limits $[rad]$	CPG links (to, ϕ $[rad]$)
10	1.0	0.0	1.0	$-1.0; 0.0$	$(7, 3.0)(12, 0.0)$
7	1.0	0.8	1.0	$1.0; 1.57$	$(10, -3.0)$
12	1.0	-0.8	1.0	$-1.57; -1.0$	$(12, 0.0)$

Table A.1: Trebuchet configuration

A.3 Dog

The dog robot shown in Figure A.4 is made of four active modules plus one inactive one (the head). The imagined gait consists in carrying itself on the rear leg, jump forwards as far as it can, then fold, and repeat the operation. All the parameters have been determined empirically and are listed in Table A.2. Snapshots from the dog video are shown in Figure A.5.

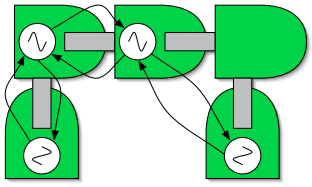


Figure A.4: Dog robot setup

Module	Ampl $[rad]$	Offset $[rad]$	Freq $[rad/s]$	Limits $[rad]$	CPG links (to, ϕ $[rad]$)
5	1.0	-1.0	0.6	$-1.55; 0.7$	$(12, 3.2)$
10	0.9	0.0	0.6	$-0.7; 0.7$	$(11, -3.2)$
11	0.6	0.0	0.6	$-0.7; 0.7$	$(10, -3.0)(12, 0.0)$
12	0.9	0.2	0.6	$-0.9; 1.0$	$(5, -3.2)(11, 0.0)$

Table A.2: Dog configuration

(a) (b) (c)

Figure A.5: Snapshots from the Dog video.

A.4 Wheel

The wheel robot is made of three active modules and three inactive ones as shown in Figure A.6. The gait wanted is the robot rolling to go forwards. The wheel robot is the trickiest experimental configuration tried. The point is to avoid the servomotor to destroy themselves by opposing their forces.

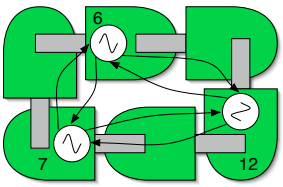


Figure A.6: Wheel robot setup

The chosen CPG configuration prevents such conflicts. If one considers a pair of an active and an inactive module as a stretchable segment, the robot can be seen as a triangle (see Figure A.7). As the longer possible segment is less than twice the shorter possible segment, any physically possible servomotor angles combination builds up a valid triangle. To avoid collisions modules pairs must always bend to the outside.

The position signals that have to be fed to the servomotors is shown in Figure A.8 as a black curve. They have a trapezoidal-like shape. They have to be approximated to a sine, which can possibly be bounded. The key points are that the signal goes from 0 to $\pi/2$ during one third of the period time, stays at $\pi/2$ during the next third and goes from $\pi/2$ to 0 on the next third, then repeats. It must never go below 0 or over $\pi/2$. A signal that fulfills these constraints is a sine passing through $(0, 0)$, $(2\pi/3, \pi/2)$ and $(4\pi/3, \pi/2)$. This gives us the following equation system if a is the amplitude of the sine, x the offset and ϕ the phase shift.

$$\begin{cases} \sin(\phi) & a & +x & = & 0 \\ \sin(\phi + \frac{2\pi}{3}) & a & +x & = & \frac{\pi}{2} \\ \sin(\phi + \frac{4\pi}{3}) & a & +x & = & \frac{\pi}{2} \end{cases}$$

The resolution of this equation system gives us $\pi/3$ for the amplitude, $\pi/3$ for the offset and $3\pi/4 + 2\pi k, k \in \mathbb{N}$ for the phase shift. Actually there is another solution with different values for the amplitude and the phase shift that produces exactly the same signal. Only the first solution is considered. Each parameter is used for each of the active modules. For the phase shifts, k is chosen to be 0 thus the phase shifts are $3\pi/4$. The phase shifts sum up to a multiple of 2π, which is required in a CPG loop. Swapping the active modules will only affect the direction the wheel will roll in. Snapshots from the wheel video are shown in Figure A.9.

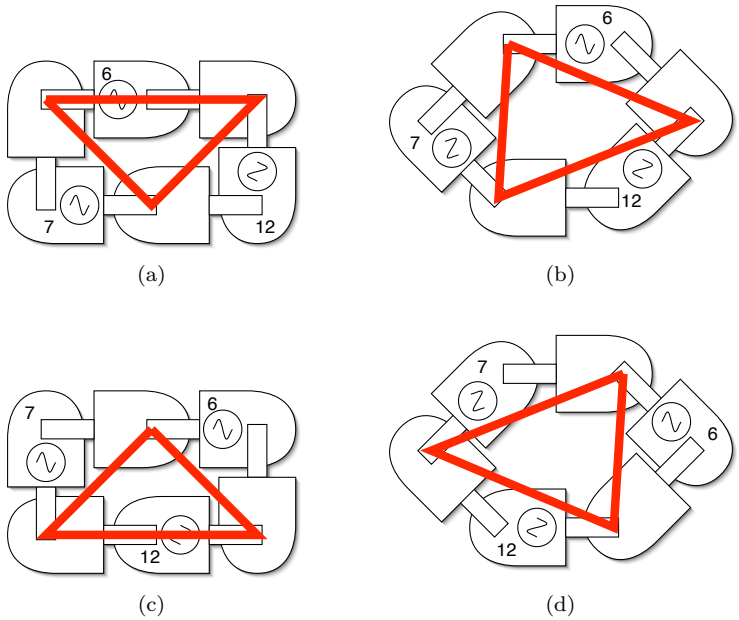


Figure A.7: Wheel gait steps.

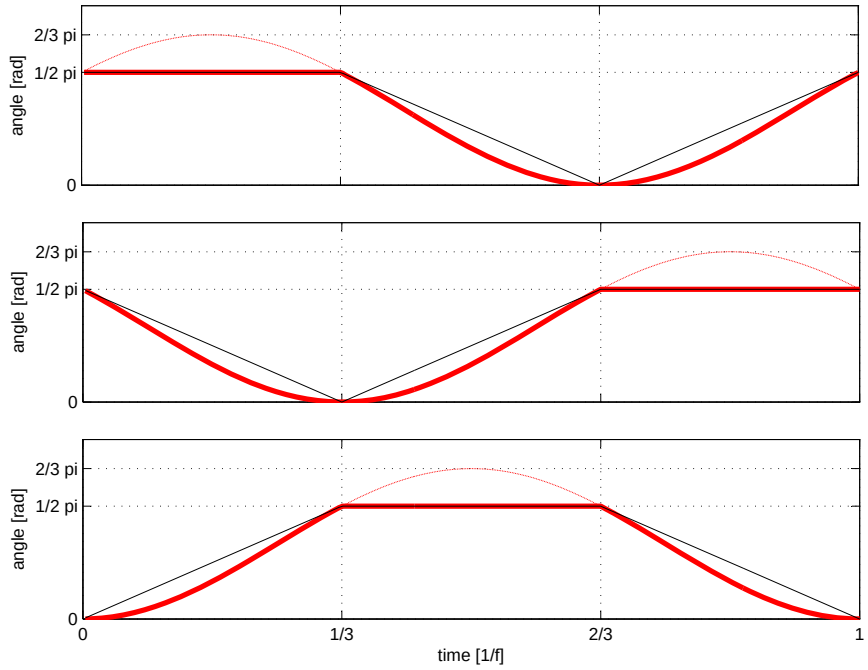
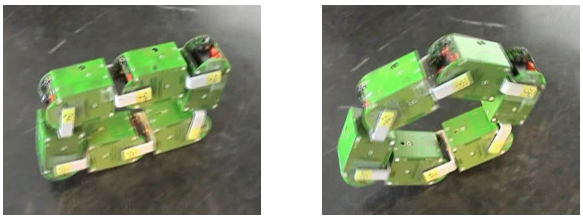


Figure A.8: The 3 servomotors positions of the wheel robot. The black line is the desired position function whereas the red line is the actual position function.



(a) (b)

Figure A.9: Snapshots from the Wheel video.

Appendix B

Accelerometer Measurements

This appendix provides detailed information about the measurements done for SensorBoard's accelerometer characterization described in Chapter 4.

For all of the figures showing either the accelerations, the velocities or the positions on the three axis, the blue curve shows the information gathered by the LED tracking system (see Section blabla). The position is directly given by the system. The velocity and the acceleration are differentiated from the position. The black curve shows the data from the accelerometers with the noise filter, whereas the red curve represents the same data with after the window filter and movement end filter have been applied (see Section blabla). The acceleration is given by the accelerometers for these two curves; the velocity and the position are integrated from the acceleration. For easy comparisons, each of the axis X, Y and Z has the same scale in a plot.

The blue curve (LED) is the most trusted one for the positioning because it is not the result of a double integration. Furthermore the LED tracking system has a very low noise and a good resolution (about $3.3mm$).

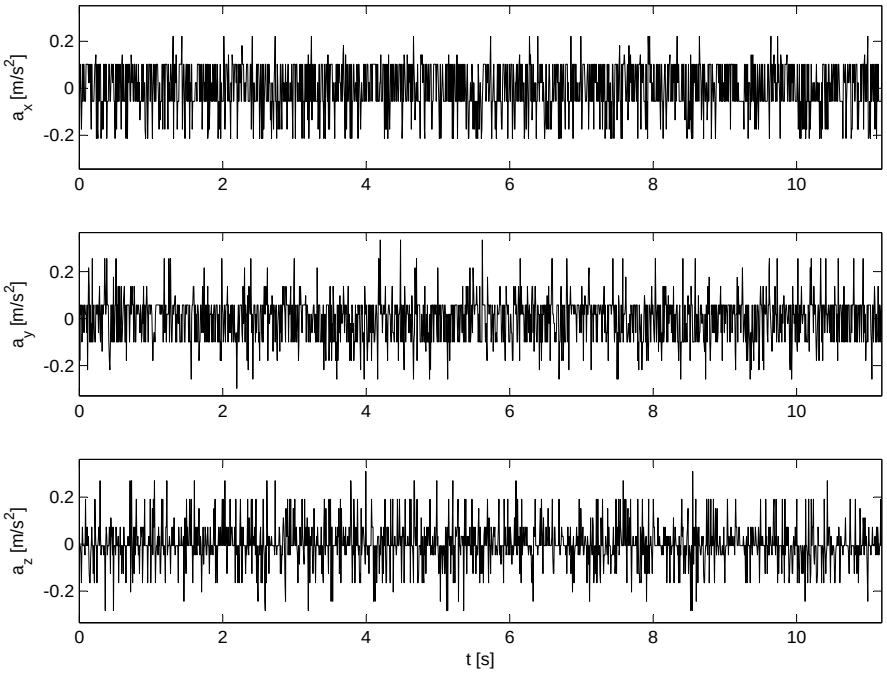


Figure B.1: Noise of the accelerometers.

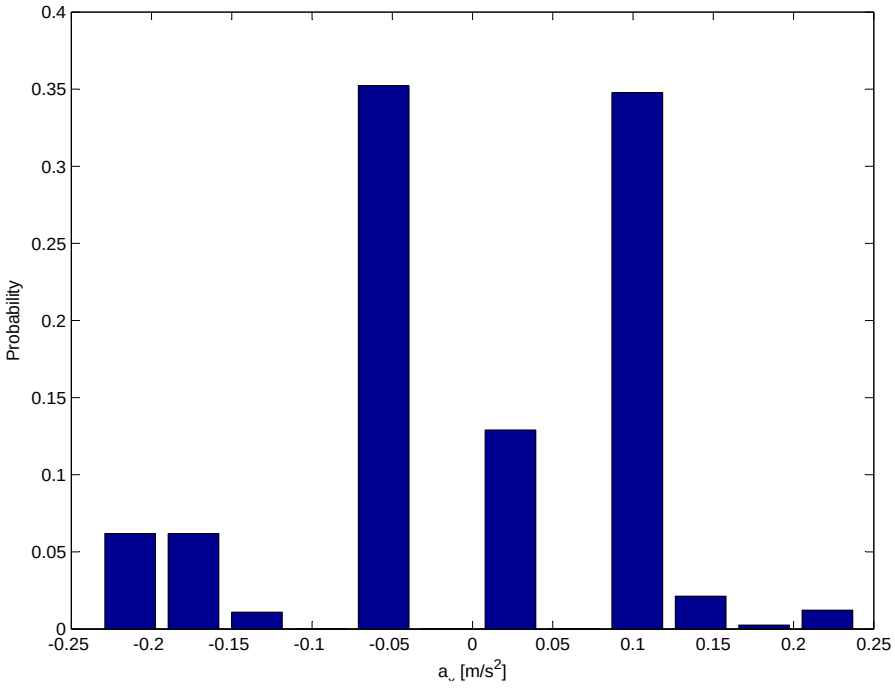


Figure B.2: Noise probability distribution for the X axis.

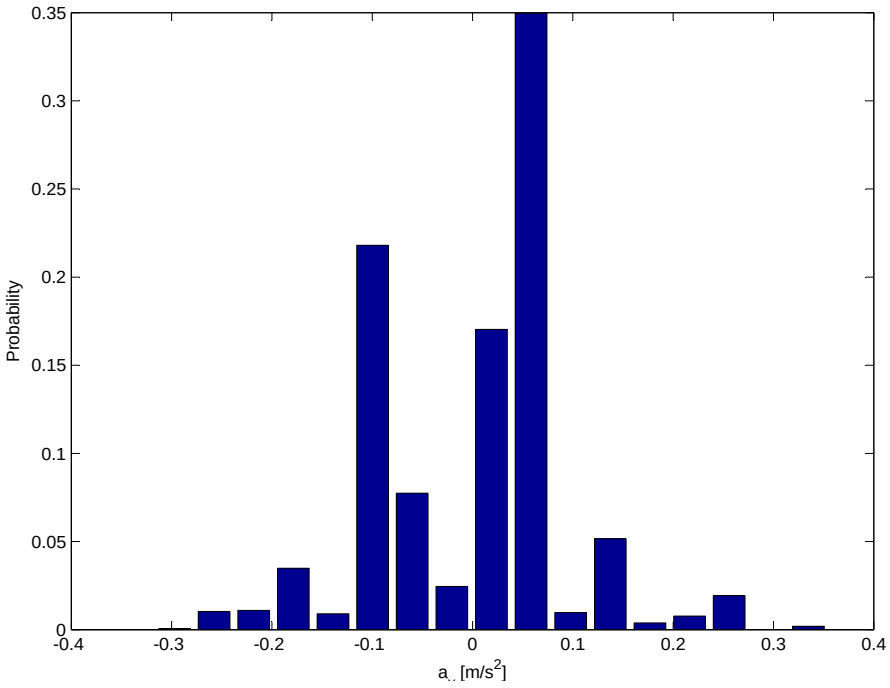


Figure B.3: Noise probability distribution for the Y axis.

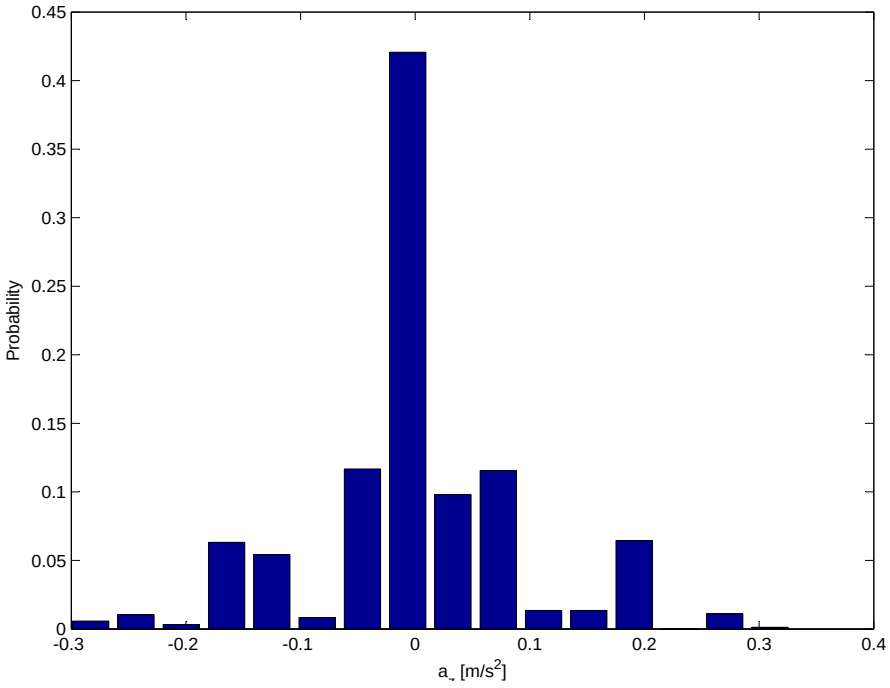


Figure B.4: Noise probability distribution for the Z axis.

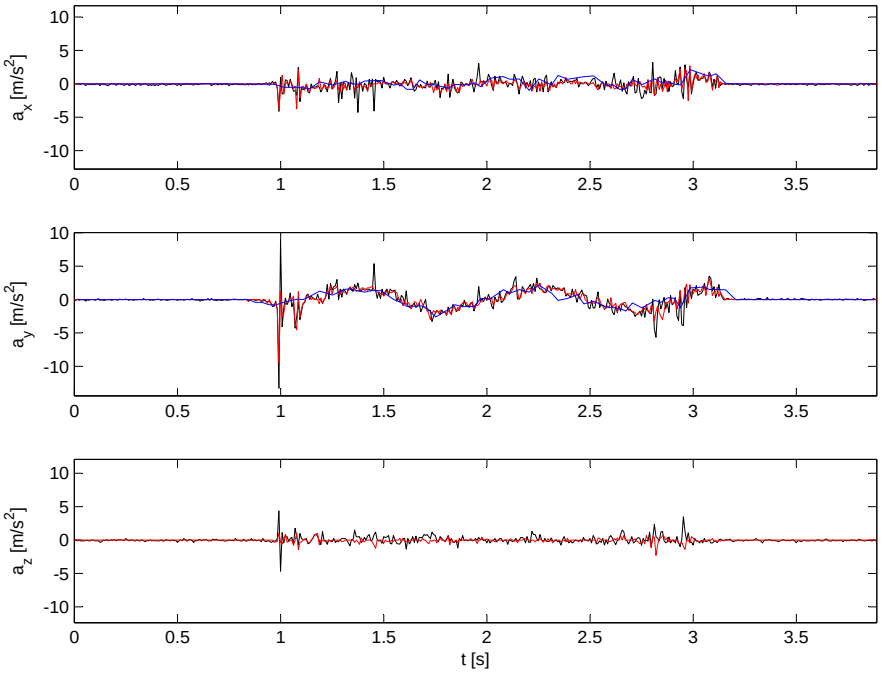


Figure B.5: Accelerations for the sine move.

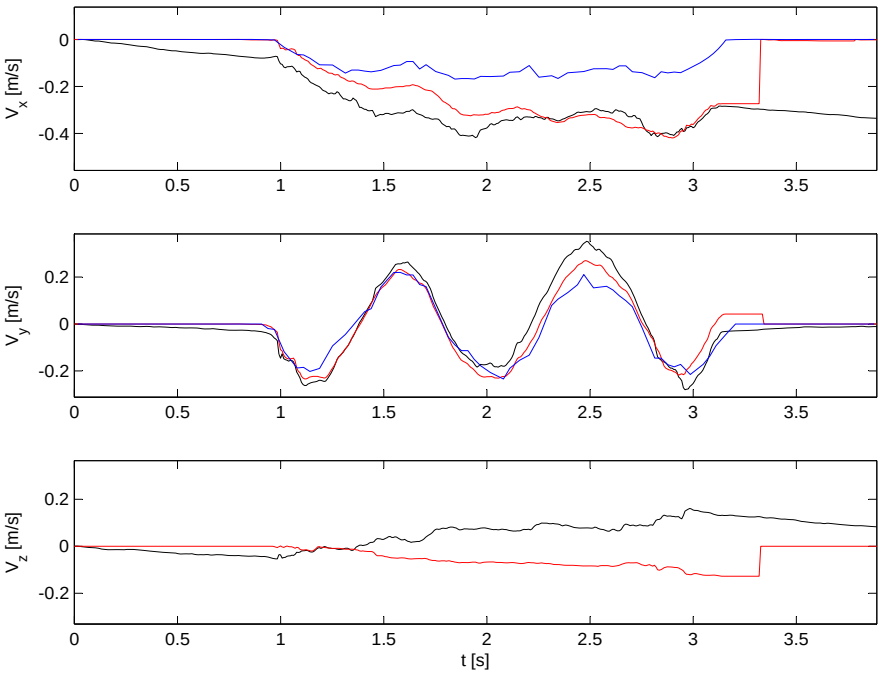


Figure B.6: Velocities for the sine move.

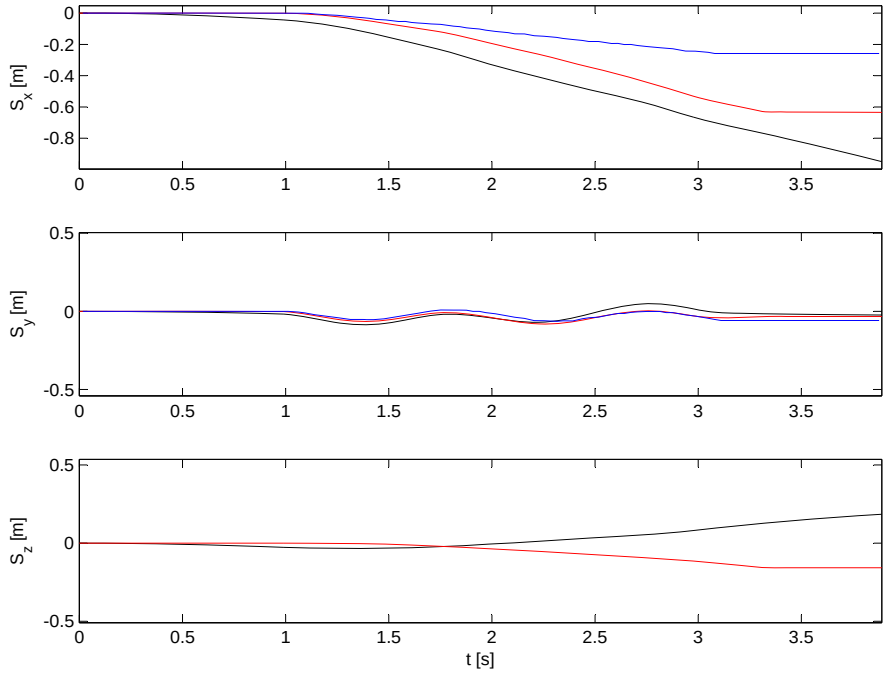


Figure B.7: Positions for the sine move.

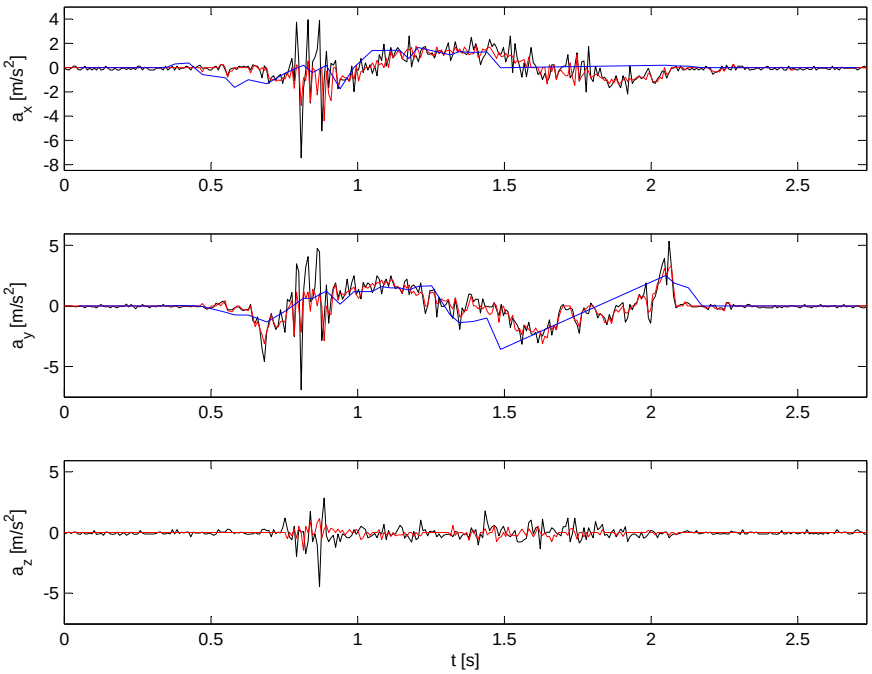


Figure B.8: Accelerations for the circular move.

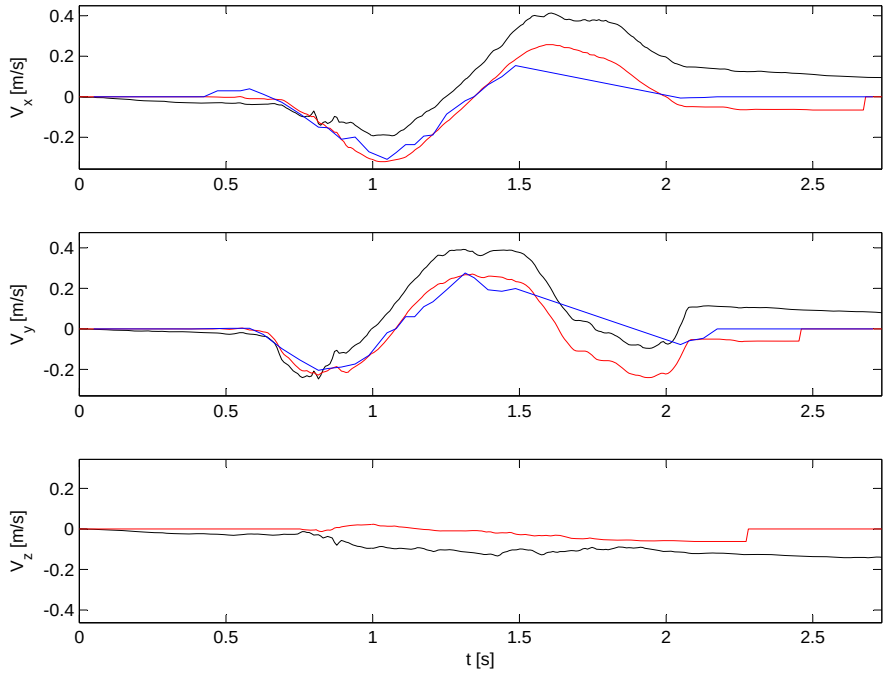


Figure B.9: Velocities for the circular move.

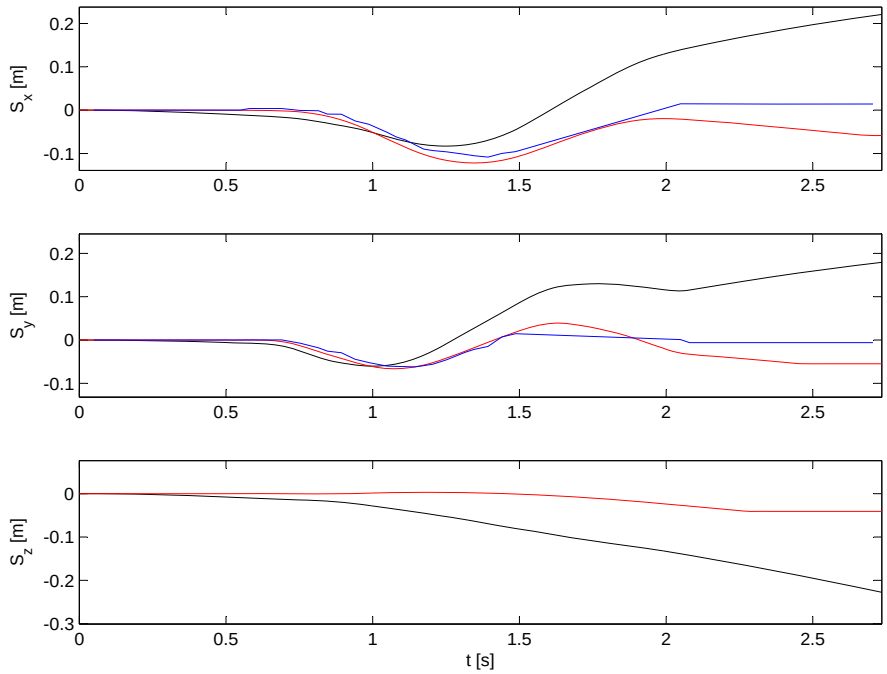


Figure B.10: Positions for the circular move.

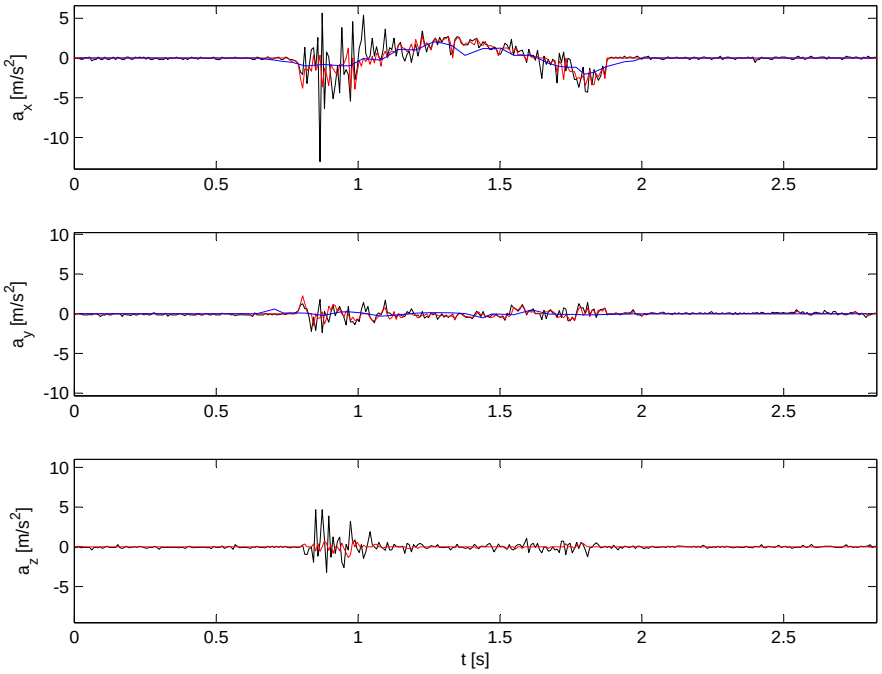


Figure B.11: Accelerations for the go and return move.

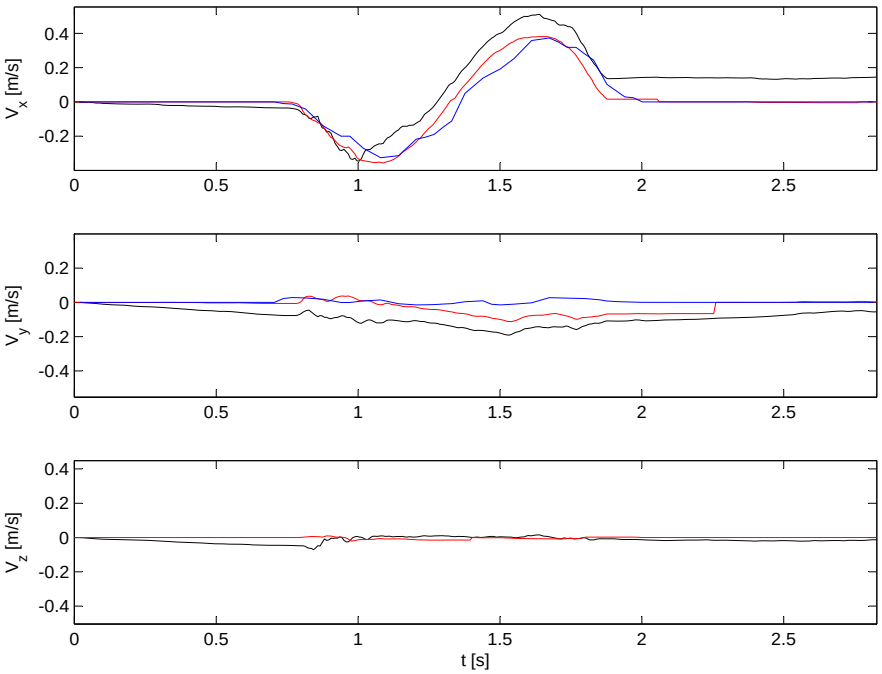


Figure B.12: Velocities for the go and return move.

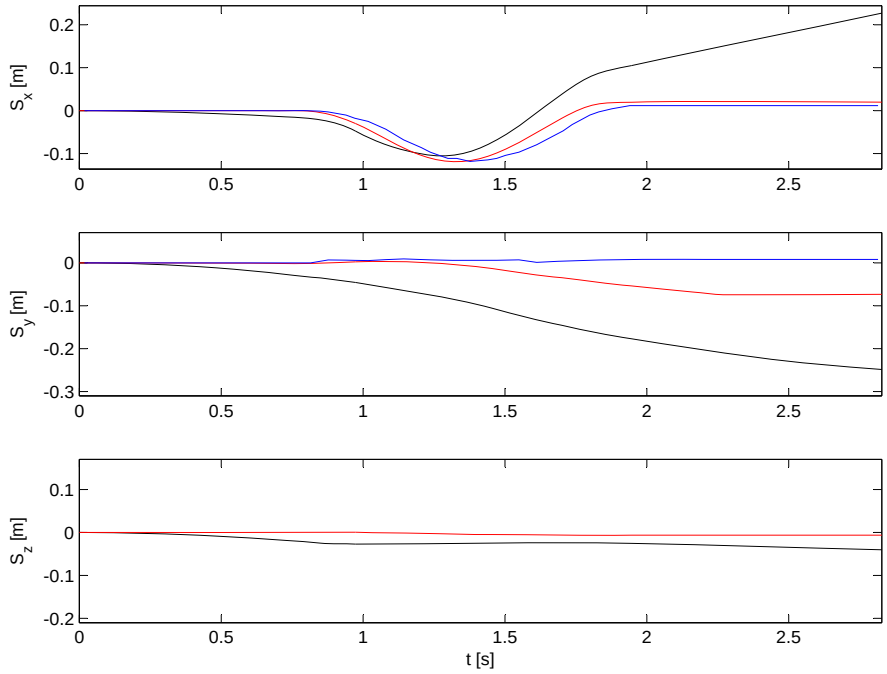


Figure B.13: Positions for the go and return move.

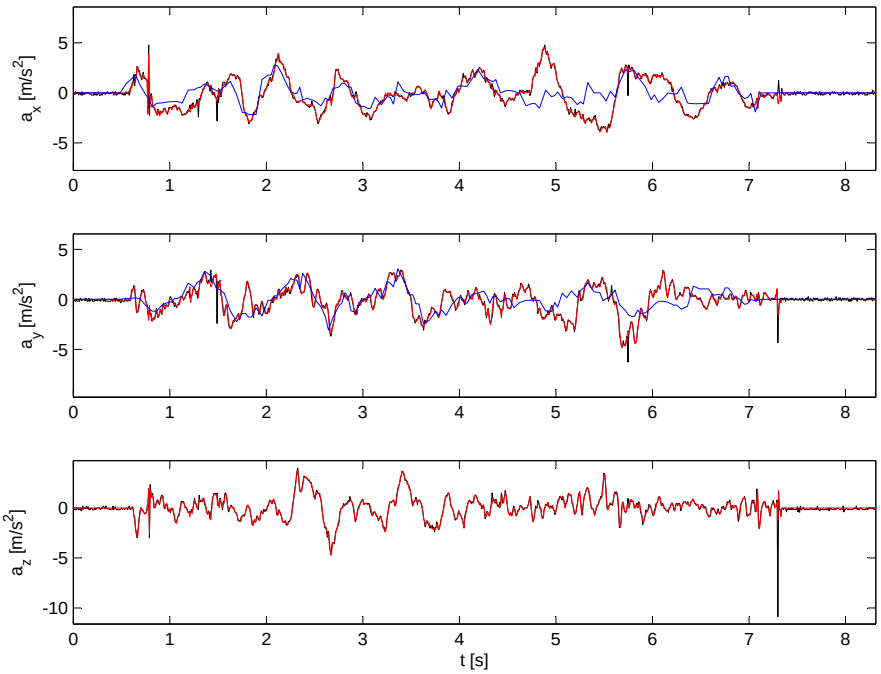


Figure B.14: Accelerations for the random move.

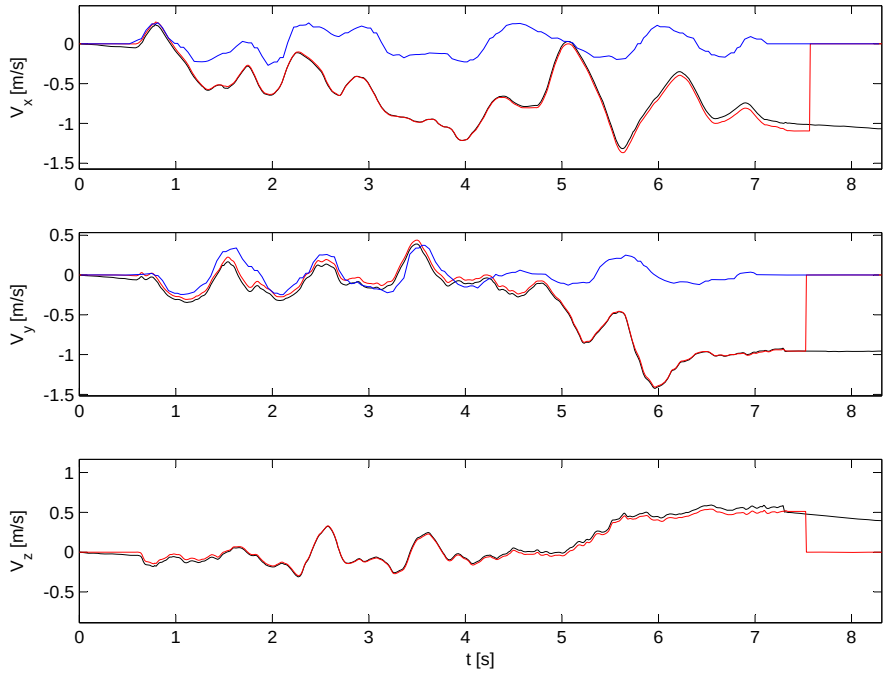


Figure B.15: Velocities for the random move.

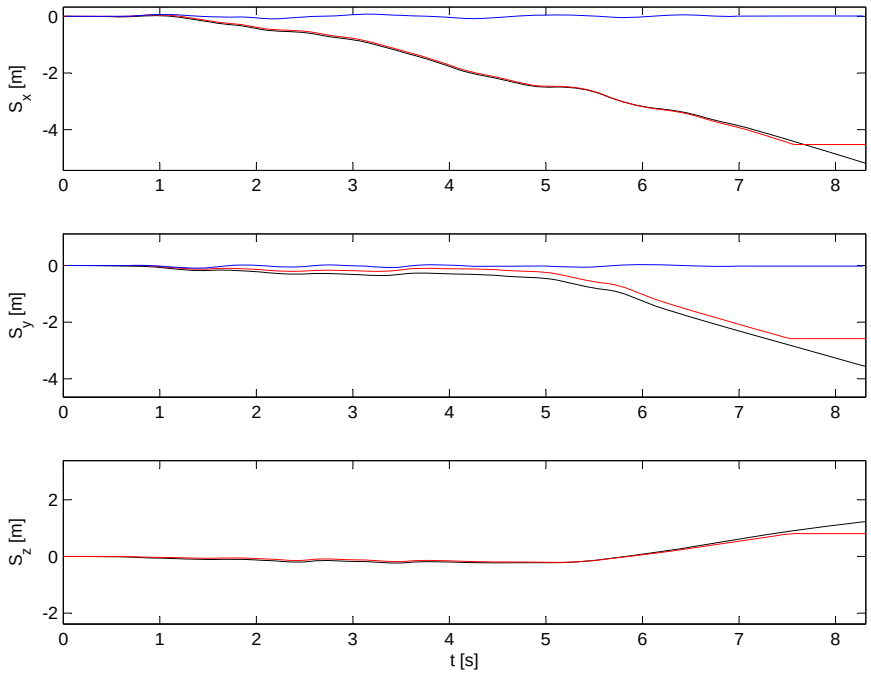


Figure B.16: Positions for the random move.

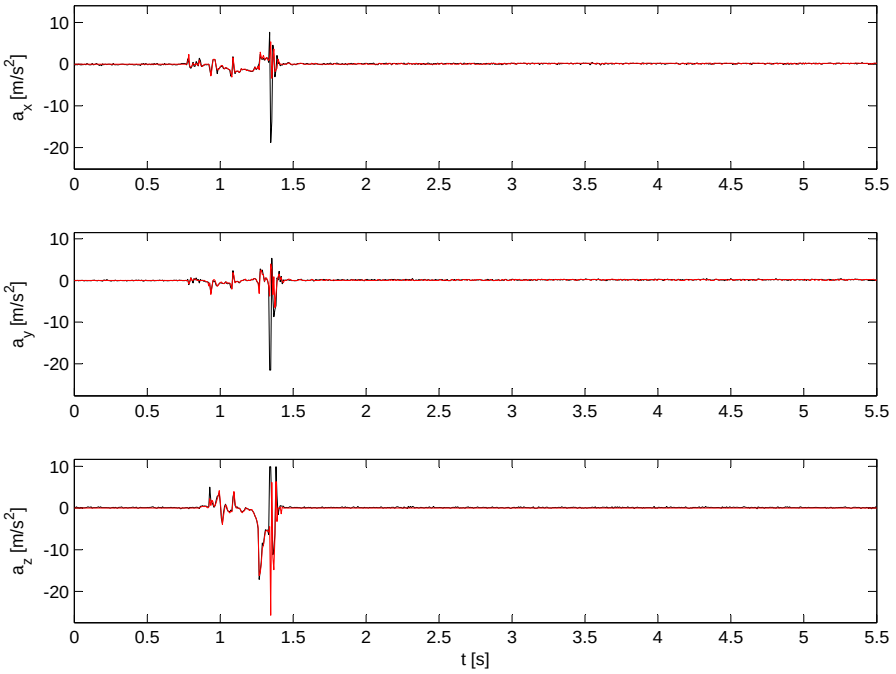


Figure B.17: Accelerations for the impact move.

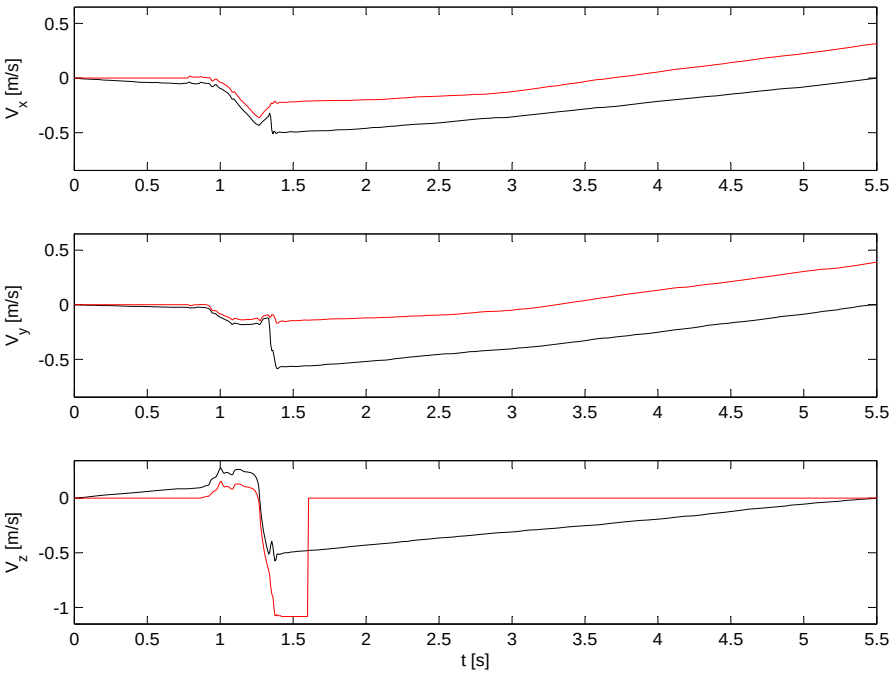


Figure B.18: Velocities for the impact move.

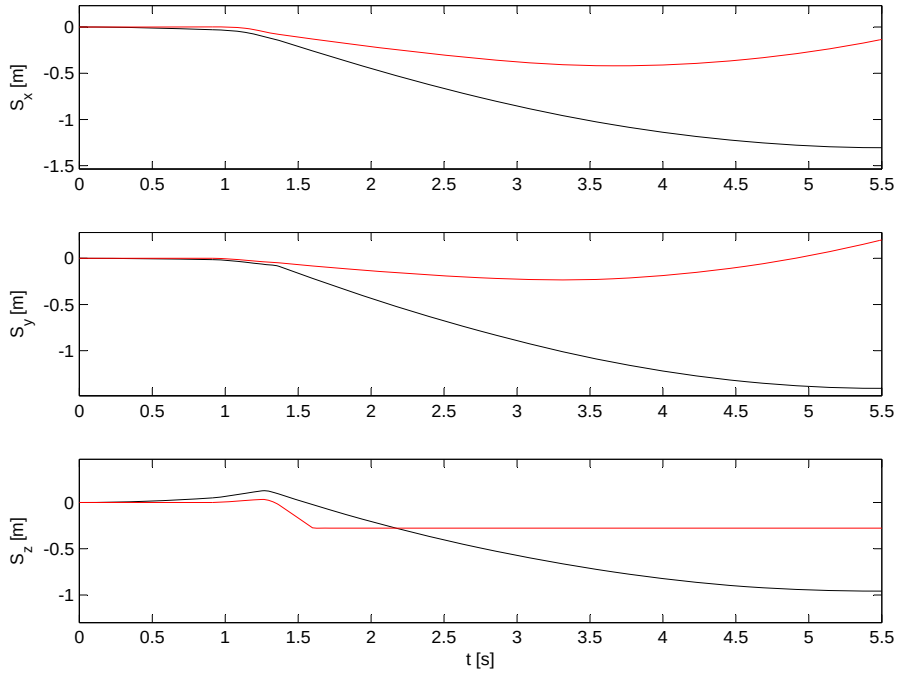


Figure B.19: Positions for the impact move.

Appendix C

YaMoR Host 3 Documentation

C.1 Settings

Before using YaMoR Host 3 you have to configure the settings properly. Click *Tools, Settings* to display the settings window (Figure C.1). Define the COM port settings if you want to use the real mode. If you want to use the LED tracking system, define its settings as well. Even if you do not use the real mode, you have to set the master node ID, which corresponds to the module that makes the bluetooth communication with the PC. If you want to use the simulation mode, set the host, which Webots is running on (usually *localhost*) and the port it is listening to (usually *2223*).

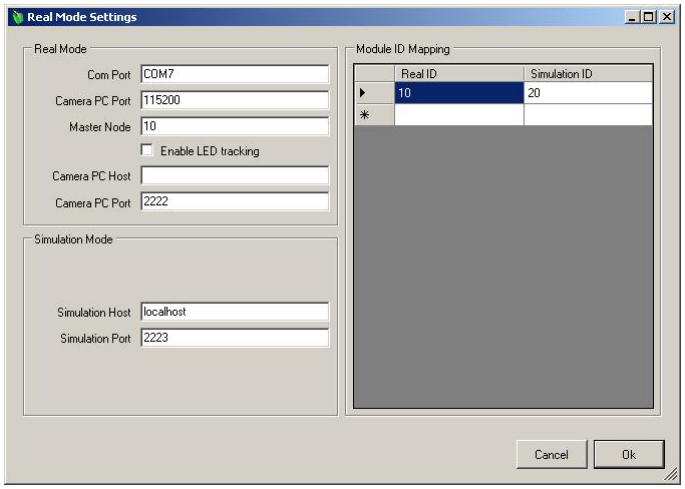


Figure C.1: YaMoR Host 3 Settings window

Because module IDs are not necessarily the same on the real robot than in Webots, you can define an ID mapping from the real IDs to the simulation IDs.

All IDs will be automatically translated by YaMoR Host 3 before being sent to Webots when using the simulation mode.

C.2 Graphical User Interface

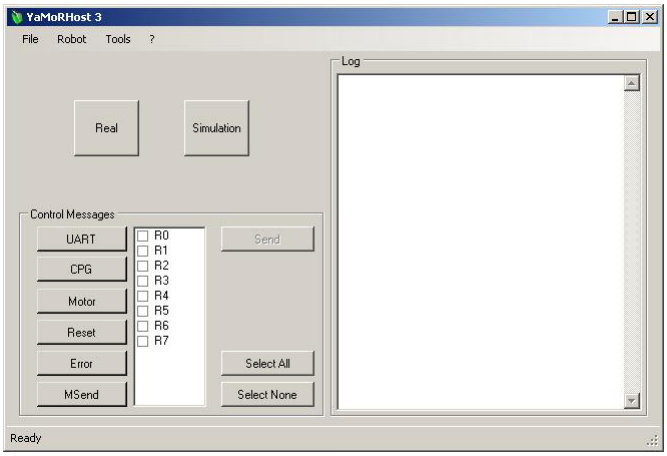


Figure C.2: YaMoR Host 3 Main window

C.2.1 Configure the Robot

The robot configuration describes the bluetooth network, the CPG network and the modules parameters. It can be loaded from or saved to a file using *File, Open* and *File, Save*.

Bluetooth

First set the bluetooth network in the *Bluetooth* tab (Figure C.3). Each row represents a connection from a module to another one. The left part of a row must appear as right part in a previous row. The left part of the very first row should be the master node id. The IDs listed also define which modules are available (except the master node).

CPG Network

In the *CPG* tab (Figure C.4), each cell of the CPG network grid represents a possible connection from the ID of the the row to the ID of the column. The connection values are expressed in the form *phaseshift,weight*. Leaving a cell blank means no connection.

Modules

All the modules can be configured in the *Modules* tab (Figure C.5). It contains the amplitude, the offset, the frequency and the limit angles for each modules.

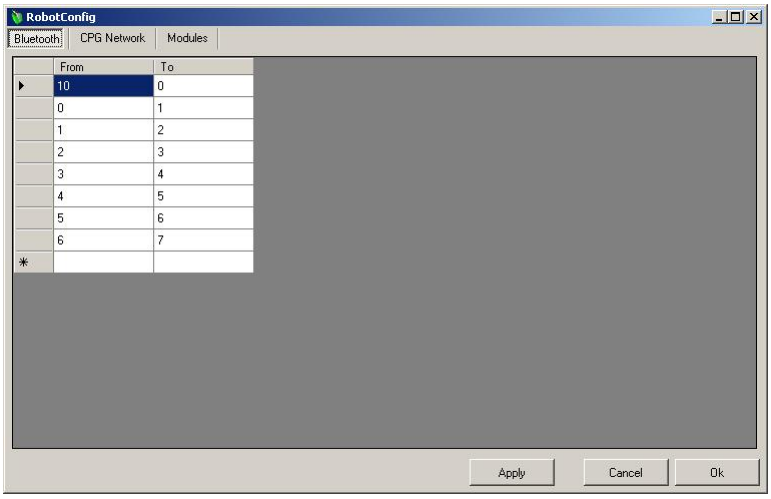


Figure C.3: Robot configuration: Bluetooth network

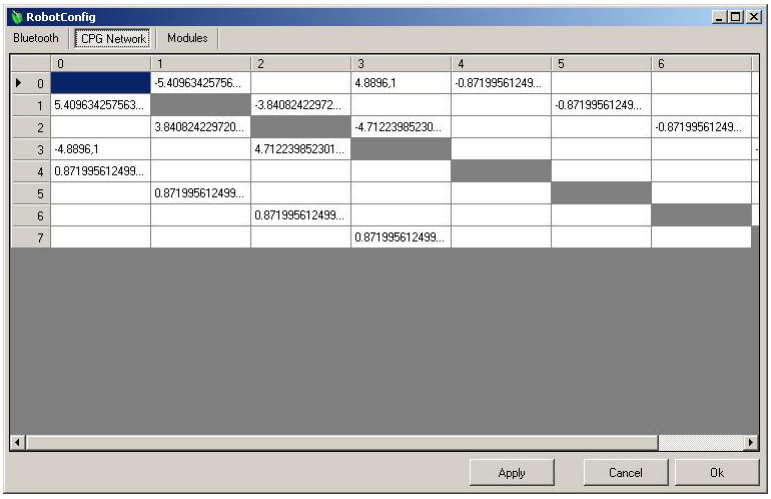


Figure C.4: Robot configuration: CPG network

If you click the rightmost button of a row, you will get a graphical module editor (Figure C.6).

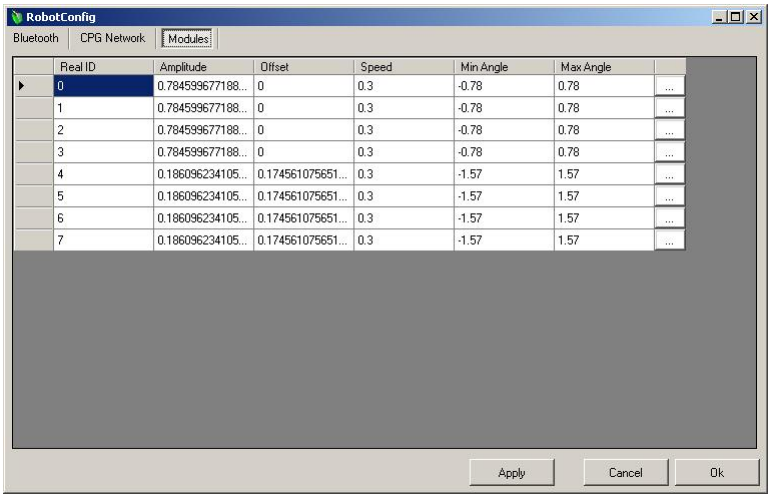


Figure C.5: Robot configuration: Module parameters

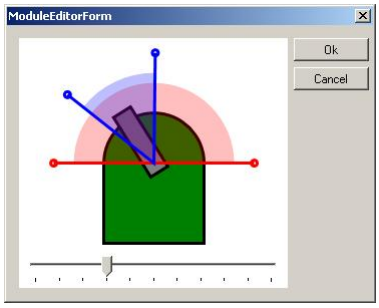


Figure C.6: Robot configuration: Interactive module editor

C.2.2 Connect the Robot

In order to connect the robot either the real mode or the simulation mode must be enabled first. Click on desired button on the main window to enable the mode you want. Once the mode is enabled, click *Robot, Connect*.

C.2.3 Send the Configuration

Once the robot is connected you have to send the parameters to it. Click *Robot, Full CPG Refresh*, which is going to sent all the parameters to the robot. When you modify the robot configuration, this is done automatically, but only for the parameters that changed.

C.2.4 Send Control Messages

With the main window you can send control messages as described in SNP protocol [18]. Click a control message option button to make it green, right-click to make it red. Select the modules you want to send the control message to and click *Send*. Options marked in red will be turned off, option marked in green will be turned on and options marked in gray will not be changed. You typically want to activate the UART, the CPG, and the motors for all modules.

C.2.5 LED Tracking

You can activate the LED tracking by clicking *Tools, LED Tracking*. Note that you must have a CamApp running on the machine you specified in the settings of YaMoR Host 3. The window (Figure C.7) displays the raw coordinates that are sent by CamApp and the corrected coordinates after the barrel deformation has been corrected. You can record the path of the LED to a comma separated values file by clicking *Record*.

Figure C.7: LED tracking system interface

C.2.6 Defect Motors

You can simulate a motor defect by clicking *Robot, Defect Motors*. When a motor is set as defect, a motor off control message is sent to the corresponding modules and motor on control messages will not be forwarded to this module anymore until you remove the check mark in the menu.

C.2.7 Physical Links

You can connect or disconnect modules by clicking *Robot, Physical Links*. To disconnect a module and put it away, select the module you want and click *Disconnect*. To connect a module to another one, select the module you want to take (it must be disconnected from any other) select the connector you want, select then the destination module and its connector and click *Connect*.

You can also run a transformation script. A script is an XML document whose document element is `<script>`. There are three possible commands in a script:

```
<connect idfrom="..." connfrom="..."
  idto="..." connto="..." angle="..." />
```

```
<disconnect id="..." />

<pause delay="..." />
```

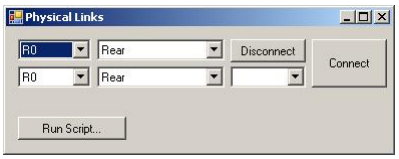


Figure C.8: Physical links editor

The attributes *id*, *idfrom* and *idto* are the real ids of the modules you want to connect or disconnect. The connectors *connfrom* and *connto* have one of the following values: *front*, *rear*, *left*, *right*, *top* or *bottom*. The pause delay is an integer value expressed in milliseconds.

C.3 RPC Interface

Before using the RPC interface you have to get the remote object that implements *IYaMoRHost3* defined in *IYaMoRHost3.dll*. This is achieved by the following C# code.

```
1  IYamorHost3 yamorHost;
2  string host = "localhost";
3  int port = 2224;
4  try
5  {
6      if (ChannelServices.GetChannel("tcp") == null)
7      {
8          ChannelServices.RegisterChannel(
9              new TcpChannel(), false);
10     }
11     yamorHost = (IYaMoRHost3)Activator.GetObject(
12         typeof(IYaMoRHost3),
13         "tcp://"+host+":"+
14         port+"/RemoteAccess");
15     // effectively test the connection
16     PointF p = yamorHost.RobotPosition;
17 }
18 catch (Exception e)
19 {
20     Console.WriteLine("Could not connect to server. " +
21         e.Message);
22 }
```

Listing C.1: Code to connect to the RPC server

Once the remote object is properly acquired you can call the method described in *IYaMoRHost3* from this object. Note that YaMoR Host 3 must be running in order to access its RPC interface.

C.4 Configuration File

This section describes the robot XML configuration file format for YaMoR Host 3. Robots configurations are saved to, and loaded from this format.

C.4.1 XML Schema

```
<?xml version="1.0" encoding="Windows-1252"?>
<xs:schema attributeFormDefault="unqualified"
elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="yamorconfig">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="modulelist" maxOccurs="1" minOccurs="1">
     <xs:complexType>
      <xs:sequence>
       <xs:element maxOccurs="unbounded" name="module">
        <xs:complexType>
         <xs:sequence>
          <xs:element minOccurs="0" name="bluetooth">
           <xs:complexType>
            <xs:attribute name="to" type="xs:unsignedInt" use="required" />
           </xs:complexType>
          </xs:element>
          <xs:element minOccurs="0" maxOccurs="unbounded" name="cpglink">
           <xs:complexType>
            <xs:attribute name="to" type="xs:unsignedInt" use="required" />
            <xs:attribute name="phase" type="xs:decimal" use="required" />
            <xs:attribute name="weight" type="xs:decimal" use="required" />
           </xs:complexType>
          </xs:element>
         </xs:sequence>
         <xs:attribute name="realid" use="required" type="xs:unsignedInt" />
         <xs:attribute name="amplitude" type="xs:decimal" use="required" />
         <xs:attribute name="speed" type="xs:decimal" use="required" />
         <xs:attribute name="offset" type="xs:decimal" use="required" />
         <xs:attribute name="minangle" type="xs:decimal" use="required" />
         <xs:attribute name="maxangle" type="xs:decimal" use="required" />
        </xs:complexType>
       </xs:element>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

C.4.2 Configuration Example

The following configuration example is used for the wheel (see Section A.4).

```
<?xml version="1.0" encoding="Windows-1252"?>
<yamorconfig>
 <modulelist>
  <module id="1" amplitude="0" offset="0" speed="0"
  minangle="-1.57" maxangle="1.57">
   <bluetooth to="6"/>
  </module>
  <module id="6" amplitude="1.04" offset="1.04" speed="0.6"
  minangle="-1.57" maxangle="1.57">
   <bluetooth to="7"/>
   <cpglink to="12" phase="-2.09" weight="1.0" />
   <cpglink to="7" phase="2.09" weight="1.0" />
  </module>
  <module id="7" amplitude="1.04" offset="1.04" speed="0.6"
  minangle="-1.57" maxangle="1.57">
   <bluetooth to="12"/>
   <cpglink to="6" phase="-2.09" weight="1.0" />
   <cpglink to="12" phase="2.09" weight="1.0" />
  </module>
  <module id="12" amplitude="1.04" offset="1.04" speed="0.6"
  minangle="-1.57" maxangle="1.57">
   <cpglink to="7" phase="-2.09" weight="1.0" />
   <cpglink to="6" phase="2.09" weight="1.0" />
  </module>
 </modulelist>
</yamorconfig>
```

Appendix D

YaMoR Optimizer Documentation

D.1 Graphical User Interface

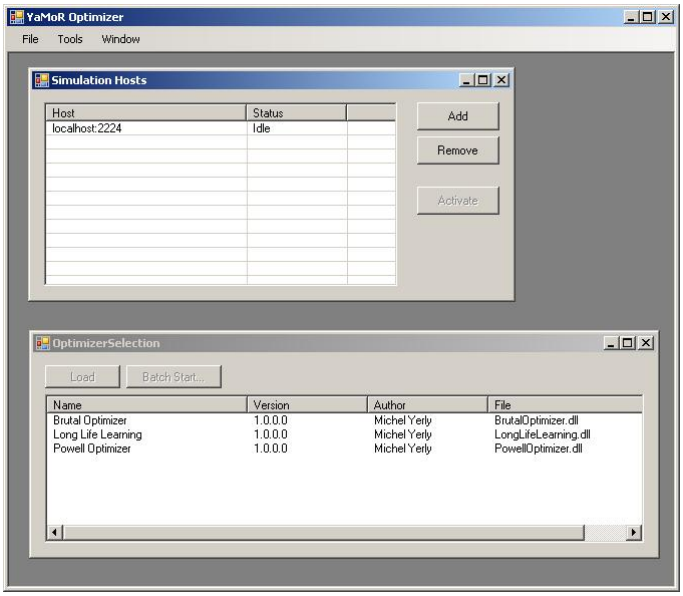


Figure D.1: YaMoR Optimizer Framework

D.1.1 Simulation Hosts Window

To get the simulation host window shown in Figure D.1, click on *Tools, Simulation Hosts*. In this window you can enumerate the available hosts for simulation running. It also displays the status of each of the hosts.

D.1.2 Optimizer Selection Window

To get the optimizer selection window also shown in Figure D.1, click on *Tools, Optimizer Selection*. In this window you see all the available optimizer and you can start them from here or make a batch start.

D.2 Writing an Optimizer

D.2.1 Workflow

An optimizer creates simulations it needs to execute, according to its own algorithm. Then it submits them to YaMoR Optimizer so that it can eventually run them on a idle machine and return the results.

D.2.2 IOptimizer Interface

In order to write an optimizer that is compatible with YaMoR Optimizer you have to implement the interface *IOptimizer* defined in *YaMoROptimizerInterfaces.dll*. There are several self description method to implement such as *Name*, *Version* or *Author*, but focus is accorded to the most important ones.

Init

Called when YaMoR Optimizer initializes your optimizer. The YaMoR Optimizer's main form is passed as parameter so if your optimizer owns a window, you probably want to put in the following code.

```
1 public void Init(Form mainForm)
2 {
3     frmParent = mainForm;
4     frmMain = new MainForm(this);
5     frmMain.MdiParent = frmParent;
6     frmMain.Show();
7 }
```

Listing D.1: Code to initialize a child window

UserConfig

Should return a *UserConfig* object (see Section D.2.3) that represents the window used to set the parameters of the optimizer. It also contains the actual values for each of them. Usually this object is global to your class and is built up in the constructor of your class.

SubmitSimulation

This is an event you must raise when you want YaMoR Optimizer to run a simulation. The simulation you pass as parameter must be a subclass of *Simulation*. As return value when throwing the event, you get an *ISyncObject*, which provides information about the state of execution of your simulation. For instance, you can make your thread wait for the simulation to begin, or end, get the simulation duration, etc.

BatchStart

This method is called when the user wants to start many instances of your optimizer. Usually you want to execute the following code sheme.

```
1 public void BatchStart(string[] parms)
2 {
3     cfg.SetAllValuesByString(parms);
4     myOptimizer.Start();
5 }
```

Listing D.2: Code template for the batch start

D.2.3 UserConfig

The *UserConfig* class allows you to create an optimizer configuration window in a very convenient way. You just have to add fields the the objects with the methods *Add<x>Field*, where <x>represents the kind of field you want. You can then display the window by calling the *Configure* method and then read out the value using *ReadValue*. All the values can be set at a time programmatically using *SetAllValuesByString*.

Currently the implemented fields are *IntegerField*, *FileField* and *BooleanField*. If you want to add new file type you have to create a new class that implements the interface *IField*. Then you have to add the method *Add<x>Field* to the class *UserConfig*.

D.2.4 Parallelizability

In order to take advantage of the many hosts you have set up in YaMoR Host 3 your optimizer should submit as many simulations as possible at the same time. If it needs the result from one previous simulation to create the next one, then parallelizing is not possible.

Appendix E

CD-ROM