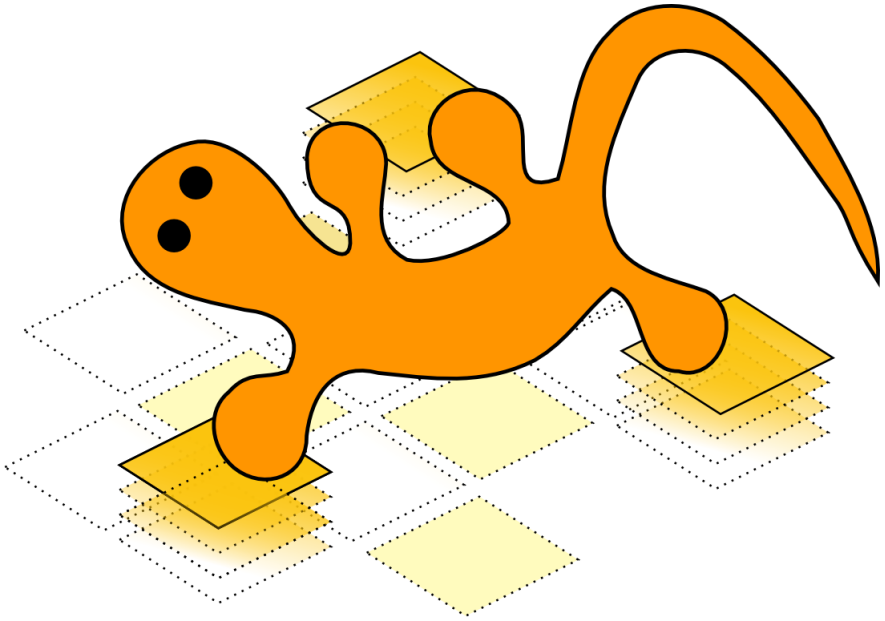
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

BIOLOGICALLY INSPIRED
ROBOTICS GROUP (BIRG)



Towards an improved framework for YaMoR : Simulation and real-time control

---

Semester Project
Summer 2004-2005

Cyril Jaquier - Kévin Drapel

Supervisors: Professor Auke Ijspeert, Alessandro Crespi, Andres Upegui

June 21, 2005

Abstract

The Biologically Inspired Robotics Group (BIRG) of the Swiss Federal Institute of Technology in Lausanne (EPFL) is working on modular robotics. We developed a Java application called Bluemove to remotely control the first version of the YaMoR modules using Bluetooth. The user can directly describe the position sent to the servos in a graphical manner. However, despite the fact that the user can drag the control points in a trajectory, he can only do that with a single curve at once. We tried to find new ways to improve the control via a scripting and plugins system with a graph interface. The user can then explore complex behaviors which were not possible with the first version of the application. Moreover, the commands can be mapped to the keyboard. As testing new robots can be a tedious job, we also worked on a 3D simulator that receives information from our Bluemove application and allows the user to quickly try various configurations in a virtual world.

Contents

List of Figures

Chapter 1

Project description

1.1 Introduction

The aim of the YaMoR (Yet another Modular Robot) project at BIRG is to construct a wireless autonomous modular robot made of several units. During the winter semester 2004/2005, we worked on a Java application, Bluemove, that could remotely control the modules. Using trajectories drawn by hand, it was possible to quickly try new configurations and check how the curves sent to the servos would behave on the real robot.

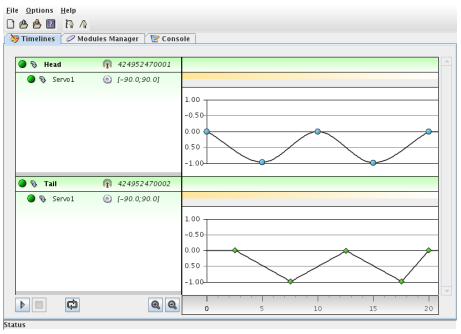


Figure 1.1: Bluemove, timelines editor

However, despite the fact that the user can drag the control points in a trajectory, he can only do that with a single curve at once. We tried to find new ways to improve the control via a scripting and plugins system. The user can now add processing nodes to a graph which will be evaluated at each step. These nodes are either generators, trajectories, filters and outputs, they affect the signals sent to the modules. The user can therefore explore complex behaviors which were not possible with the first version of the application. Moreover, the commands can be mapped to peripherals like the keyboard. As testing new robots can be a tedious job, we also worked on a 3D simulator that receives information from our Bluemove application and allows the user to quickly try various configurations in a virtual world.

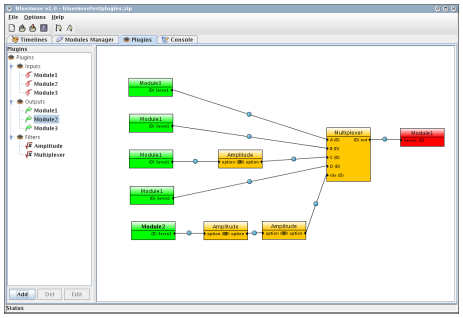


Figure 1.2: Bluemove, real-time control and scripts

We also had the opportunity to present the robot and the concept of modular robotics to the audience during the inauguration of the IC building at EPFL. We were able to clarify the directions that should be taken for the second generation of the modules.

Chapter 2

Inauguration of the IC building at EPFL

2.1 Introduction

On 21 March 2005, the EPFL inaugurated its new IC (computer and communication science) building. Some laboratories including BIRG were invited to present their projects. About 600 people attended the event and we made some demonstrations of the modular robot. As the audience was not strictly restricted to engineers or computer science students, it was the opportunity to have a feedback about this project from an external and neutral point of view.

This event took place a few days after we noticed some problems with the new FPGA boards. We will discuss in details about these electronic issues in chapter 5

Figure 2.1: The BIRG table during the inauguration of the IC building

2.2 Demonstrations

We created a video of about 3 minutes showing the work that had been done on the YaMoR project, including projects by Yvan Bourquin, Daniel Marbach, Elmar Dittrich, Rico Moeckel and ourselves. This short video clip was continuously played on a screen. This way, people could have a visually appealing summary of the project.

For the demonstration, four working modules were available. As we did not have much time to make an innovative robot, we decided to build an old model we had called Trebuchet. We discovered that the Trebuchet could be quickly transformed into another type of robot by detaching a single module and attaching it to the other end. This would make a configuration similar to a sliding stick. We could therefore show the modularity of the whole system and how a robot could evolve to another morphology.

This event was also a good torture test for Bluemove and the Bluetooth protocol. While making the first tests at the place where the event was going on, we noticed that there was a large amount of Bluetooth cellphones in the area. This was a situation we had never faced as in the laboratory where we usually work, the number of non-YaMoR Bluetooth devices never exceed two or three units.

We feared this would make interferences with our robot but finally, this did not have any impact on the communication with the modules. We only had a few stalls which needed a reset of the Bluetooth board but this issue was already known prior to this event.

We could also test the autonomy of the robot. During the show, the robot was often moving around. We did not expect the modules to have

enough energy for more than 90 minutes. The batteries finally provided a sufficient amount of power for about 120 minutes. Most consumption was caused by the servos, the Bluetooth card and the FPGA only need a marginal part of the batteries energy.

On the other side, this event confirmed that the velcro connections were definitely too weak for other purposes than rapid testing and could not sustain the constraints which are present in such a modular robot. Attaching and detaching the modules to present new robots quickly worn out the velcro connections. At the very end of the show, the connections would barely stand the weight of a single module. A reconfigurable and robust solution is hence strongly recommended for future versions of YaMoR modules.

2.3 Conclusion

People have shown interest into this project. Many of them asked us about the applications of modular robotics which was a fresh concept for them. We tried to show that there was a strong link between nature and this type of robots, and how adaptation could occur in various environments thanks to the reconfiguration. Unfortunately, it was not possible to show an automatic reconfiguration as the YaMoR project is still in an early stage of development and modules do not have this feature yet.

We were glad to be able to show this project to attenders of this inauguration. While humanoid robots such as Asimo or mammal-like robots (Aibo) are often present in medias or movies thanks to the Japanese efforts in this field, modular robotics is less known and may positively surprise people who never heard about such researches.

Chapter 3

Eve, the YaMoR simulator

3.1 Brief overview

Professor Ijspeert suggested that it would be convenient to have a simulator where one could test the trajectories drawn in Bluemove. Previous works on YaMoR simulation were done by Yvan Bourquin and Daniel Marbach but they focused on evolution and exploration of new configurations. Interaction with the user was mainly restricted to setting initial parameters for the evolutionary algorithms. Our simulator aims at filling the gap between the user and the virtual robot *while* it is moving. The current version of our simulator works with the same configuration during runtime (ie. no morphological evolution) but the way the modules are connected can be quickly changed to conduct a new simulation.

A simulation offers many advantages :

- one can test an infinite set of configurations with a large number of modules
- a simulation is quickly launched compared to assembling and configuring real modules by hand
- a real modular robot autonomy is limited and will not last for more than two hours
- YaMoR connections are weak and real robots tend to lose some modules while moving, a simulation is not subject to these issues
- the same robot can be tested in various environments (flat floor, stairs, etc.) with different parameters such as gravity
- the user can visualize the effects of trajectories on the robot motion, he can then modify these trajectories
- the simulator can be extended to include new commands such as reconfiguration orders
- the simulator may help to test new hypothesis and features like sensors, neural networks, etc.
- a complex simulation could be performed and visualized on a computer while controlling is done on another computer, the low bandwidth traffic between the two machines would be restricted to simple commands such as position of the servos

Unfortunately, virtual robotics also brings its own list of problems which are mostly related to physics :

- numerical instabilities due to the physics engine, especially its integration scheme
- noise, imperfections, friction are difficult to reproduce and setting them aside would not be realistic
- collisions in the physics engine may lead to unexpected motions, this is caused by the simplified shape, weight balance in the meshes representing the modules as well as improper restitution after collisions
- applying the same trajectories on the real robot will produce a different motion, initial conditions are important
- a good and accurate simulation with many modules may need a powerful computer, though this is not a big deal with modern hardware

We will now discuss the pro and cons related to simulation in robotics. We will also

talk about the implementation of our simulator called Eve[1]

3.2 Simulation in robotics

Robotics is a field where simulation is very important. It allows the researchers to check whether their assumptions are correct and test new ideas without any specific hardware. Most of them make intensive use of physics libraries like ODE, Tokamak, Newton or Havok (commercial). Many projects including some at BIRG use the application *Webots* developed by Cyberbotics.

Physics on a computer is though prone to numerical errors and the mapping from the simulation to the real robot is a complex problem. Libraries like ODE are not designed for engineering problems but rather real-time applications like games where a large margin of error is acceptable. The physics is also simplified to speed up computation. It is difficult to properly simulate a complex system like a robot when some restrictive hypothesis are set on the software side.

3.2.1 The gap between simulation and reality

Natural imperfections are also extremely difficult to quantify. Noise is inherent to all real physical phenomena but these chaotic aspects are hard to insert into a simulation without bringing instabilities or unwanted side effects. Servos are not perfect and a small glitch in the gears may lead to an unexpected motion. We have noticed that the current prototype of YaMoR module is highly sensitive to parameters related to friction. This is due to the design of the case where pads, legs or hooks are missing. The modules are mainly sliding on their sharp edges. The surface in contact with the medium is hence drastically reduced compared to other kind of robots where the contact area is larger (robots with wheels or tracks such as on the Millibot modular robot).

Some YaMoR configurations take advantage of the sliding properties of the current prototype to move forward. Without sensors, the robot has no feedback about the effect of friction on its motion. Adding real-time monitoring together with an advanced processing to cope with unreliable measurements could help the robot to recover from errors and unwanted situations. For more information on the latest point, we invite the reader to explore [8] and its references.

In conclusion, tribology[2] is a vast field which is unfortunately too complex for real-time simulations. To some extent, it could be included in a non-real-time simulation but in that case, one loose the user interaction that we were looking for. In physical routines like those implemented in ODE, some efforts have been put into adding a basic support for things like friction with a simplified Coulomb model and parameterizable restitution.

But most problems with ODE and other physics engines are caused by numerical errors which may lead to unstable simulations where objects are exploding. This is particularly visible with joints moving away from their original axis and producing an oscillating motion. ODE provides some parameters to reduce these problems without completely fixing them. On the other side, this may look more realistic as YaMoR modules are connected using velcro. These connections proved to be quite weak in some situations where forces are acting on a direction parallel to the connector. Such forces on the velcro induce loose joints.

With some fine-tuning, one is able to produce satisfying results but tests and validation on the real hardware are undoubtedly needed. It would be interesting to minimize the errors between the virtual robot and its real counterpart by tweaking the physics and objects settings. However with the numerical problems in mind, it is probably impossible to find a fixed set of parameters for the physics engine that will nicely work for all possible configurations.

[1] named after the *Adam* framework by Daniel Marbach

[2] tribology is the science of friction, lubrication, and wear

3.2.2 Reviews of some simulated robots

It would be pretentious to give a complete list of all simulations done in robotics. Robotics is tightly related to numerical analysis and computations performed with the help of computers. But nonetheless, it is interesting to pick a few projects and check how their teams managed the simulations. Obviously, many of them have features that target a particular robot but some ideas can be expanded to other types of robots. We also examinate how some concepts could be mapped to YaMoR.

EyeSim

EyeSim is simulator which can handle more than one type of robot. It is comparable to Webots. It has been developed at the Mobile Robot Lab of the University of Western Australia at Perth [10]. The simulator is quite simple and uses OpenGL. The controlling is more interesting with a TCL/TK software that provides a visualization of the sensors values and allows tweaking the error models. As their vision-based robots makes an intensive use of sensors via distance detection and camera, they had to find ways to properly take noise and perturbations into account. A zero-mean Gaussian noise is simply added to the locomotion signals. The simulator has been used with several robots including a strange looking wheel-based vehicle, the Omni-2.

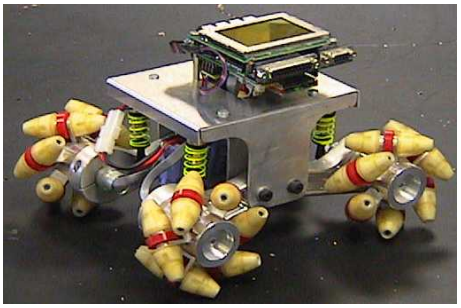


Figure 3.1: The Omni-2 robot

Another point that is worth mentioning is that they also created an error model for the wireless communication, they call it "channel noise" and their model reflects a binary symmetric channel. While this sounds appealing, it would be difficult to adapt such errors to a Bluetooth communication for the YaMoR simulator. The Bluetooth protocol is complex and provides error detection schemes with packets being sent again in case of corruption. This would add a large overhead to the simulator for a small gain. On the other side, one could use a simplified model based on distance between the transmitter and the modules. A YaMoR module could be "disabled" from time to time during a short period and hence reproduce problems with transmission.

In our simulator, transmission is supposed to be reliable. However, due to the underlying protocol, packets never arrive at the same time from Bluemove. These small delays bring a non-determinist component into the application, but synchronization is still enabled as Bluemove will wait an acknowledgment from the simulator before sending another position.

The team of EyeSim also included multi-threading and synchronization between robots. Interaction between YaMoR modules is something that will probably be addressed in the future. We did not add any support for synchronization between modules in our simulator as this feature is not available in our control software, Bluemove, as well as in the FPGAs mounted on the modules.

SubSim

SubSim is an autonomous underwater vehicles simulation which was developed by the same laboratory at Perth and which is freely distributed.

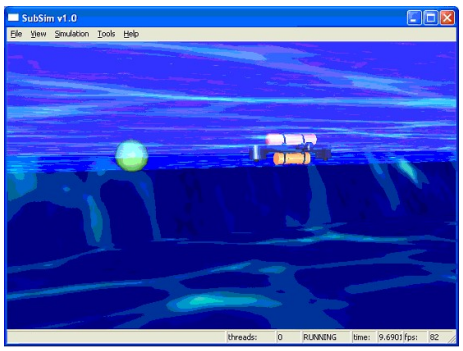


Figure 3.2: A screenshot of the SubSim simulator

One interesting feature is the use of XML files to store the data and settings about the objects.

In our simulator, we also used XML as it was a convenient choice compared to binary files. We will discuss in details about XML in the next section. SubSim comes with an API called EyeBot, it acts like an emulator for the embedded code present on the micro-controllers and a system of plugins allows the user to expand the behavior of a robot. We encourage the reader to browse the homepage of this laboratory [3] as many other simulations are available.

3.3 Introduction to Eve

Our simulator, Eve, is based on ODE (Open Dynamics Engine) and an open-source 3D engine called Irrlicht. It is written in C++. We wanted to build a realistic simulator that would allow the user to insert the robot inside a complete virtual word. Most simulators have a simple world consisting of a simple floor and at best a few static obstacles. As a modular robot could be deployed in various situations such as a rescue mission in a hostile environment, we decided to implement this feature using Quake III maps.

3.3.1 Physics engine

There are not many physics engines of quality that can be implemented in real-time applications such as simulations or games. Well-known libraries are ODE, Tokamak, Newton and Havok. Unfortunately all of them but ODE are not open source engines. Havok is an expensive physics engine while Tokamak and Newton have restrictions on their license and do not provide their source code.

ODE (Open Dynamics Engine) is an open-source physics engine. It is mainly dedicated to simulations and games that use rigid bodies connected with joints. Collisions are handled using hard contacts.

YaMoR servos are not perfect from the mechanical point of view and the lack of encoder do not allow the correction of errors. Updating the servo position in ODE is done by giving a speed which is computed using the difference between the actual and the desired position. As there is no generic method to simulate flaws in actuators, we first used an uniform noise between -1 and 1 degree. This is a smaller value than what is usually proposed in the literature for sensors (about 5%) as suggested in [14] and [15].

We then noticed that something comparable to noise was already brought by the delays between the packets sent to Eve from Bluemove. While the transmission is reliable, the time between two remote commands always fluctuate. This depends on several parameters such as processor speed, network load, etc. If the update arrives a few milliseconds before what is expected, the behavior will be different. Accordingly, we decided to remove the noise on the servos.

ODE dimensions and parameters have been converted to S.I units. The integration step must be set between 1/500th and 1/700th of seconds. Unfortunately, converting to S.I units also decreased the frame rate of the simulator. Before the conversion, a physics step of 1/700 was used but ODE was updated only every 1/300th of seconds. This way, we could keep a smooth frame rate at the cost of non S.I values. As S.I values are more convenient to deal with, the XML file contains values in S.I units.

To deal with the performance vs. accuracy concept, we introduced two parameters in the configuration of Eve (eve_config.xml). One tag is called "steps" and another is called "substeps". The first one is equal to 1.0 over the ODE integrator step while the latest is the rate of ODE update. For example, if "steps" is set to 1000 and "substeps" is set to 100, it will mean that an integration is performed every 10 milliseconds with a 1 millisecond integration delta. This can dramatically increase the frame rate of the simulator but will spoil the S.I units. Everything is consequently scaled. If the ratio between the "steps" and "substeps" is 10, the gravity must be multiplied by 10. To keep S.I units, these "steps" and "substeps" *must* be equal, above 500 but expect the rendering performance and response to drop.

If the computer can not achieve the physics rate, several integration steps will be computed before rendering the frame. The physics do not have its own thread and is inserted inside the main loop of the simulator together with the rendering calls. For example, if the last frame was rendered 7.5 milliseconds ago, we would call two times the integrator and then render the virtual world. The remaining 1.5 milliseconds

are reported to the next frame. Using what is called a *canonical loop* by game developers, we achieve a constant rate on any computer. The same system is used for the rendering. Constant steps are also recommended for physics as they usually produce less numerical problems, adaptive steps are less predictive in terms of stability[3].

3.4 Quake maps

Quake III is a game developed by ID Software where the player must shoot enemies moving inside rooms and corridors. There is a large community around this game on the Internet and many free maps are available. These worlds contain many interesting architectural constructions like stairs, pipes, holes or pits, bridges, etc. We decided to use the demonstration version of Quake III, all data (textures and four maps) are contained in a zip file that can be freely distributed.

Figure 3.3: Quake III

Irrlicht, the 3D engine we used for this simulation, can load Quake maps. The import code is not perfect, it misses some features present in Quake maps such as Bezier patches (curved surfaces) or sprites, but the remaining geometry is correctly loaded and is largely sufficient. Most of the job came down to linking the geometry of a Quake level with the collision system of ODE. It is basically a matter of extracting the triangles using Irrlicht and converting them to a Trimesh structure provided by ODE. We noticed that some triangles were defined in a clockwise way and others had an anti-clockwise order. This produced unexpected collisions where the robot would dive into the floor as the normal was inverted. It was corrected by setting all triangles to a consistent vertices order.

At first, we thought that limiting the number of triangles sent to ODE would improve the performances, a normal Quake map can contain up to 15000 triangles, this amount can be much larger and sometimes reaches 50000 triangles for complex levels. But we noticed that creating a new collision mesh for each frame using a bounding sphere around the robot would not improve the speed. We thus removed this code overhead and used a single Trimesh for the whole Quake level, ODE optimizes the collisions thanks to its hash space support.

3.5 Modeling

To add more realism to the simulation, we decided to build a fully textured mesh representing a YaMoR module using 3DSMax. This low-poly object is made of about 200 triangles and do not take much time to be rendered on modern graphic cards. The main mesh is only used for rendering, the collisions are performed on a simplified mesh consisting of a cube. Using a mesh of 200 triangles dramatically reduce the frame rate when collisions are happening. With a rough approximation of the shape, no slowdown is happening when many collisions are detected.

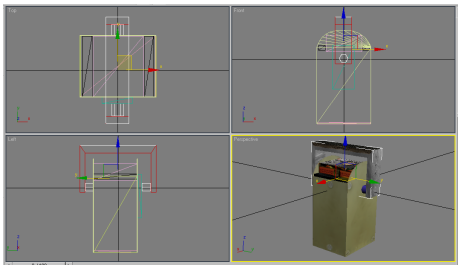


Figure 3.4: View of the YaMoR module mesh

The meshes (in 3DS format) are then imported into Irrlicht and can be easily added to the main scenegraph.

[3] dividing the integration step by two do not mean the accuracy will be doubled

3.6 Setting up YaMoR in Eve

The robot configuration is stored in a XML files. It is described as a tree. Each module has 6 faces: the 4 sides, the bottom face and the leveler. These connectors have been given names which are used in the XML file (see on figure 3.5).

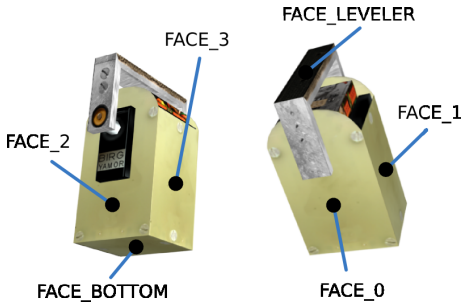


Figure 3.5: The six connectors on a module and their respective names in the XML file

In his evolutionary simulator *Adam*, Daniel Marbach used his own text format to describe the initial configuration of the robot. We decided to build another format based on XML instead of relying on a format that would need an additional parser. Our robot description is also much easier to use compared to Daniel's approach as the user do not have to give the numerical values of angles and positions. The user just needs to provide the names of two modules and their respective connectors, an angle around the pivot of the connection is also necessary. Eve will then automatically align and move the modules to match the connections settings. This way, a robot can be quickly set up and will not need much fine tuning before finding the correct configuration.

A possible configuration in XML may look like this :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<eve>
  <modules>
    <address value="424952470001" alias="head"/>
    <address value="424952470002" alias="body1"/>
    <address value="424952470004" alias="body2"/>
    <address value="424952470005" alias="tail"/>
  </modules>

  <connections>
    <connection moduleA="head" moduleB="body1"
                faceA="FACE_0" faceB="FACE_2"
                angle="0"/>
    <connection moduleA="body1" moduleB="body2"
                faceA="FACE_2" faceB="FACE_3"
                angle="0"/>
    <connection moduleA="body2" moduleB="taile"
                faceA="FACE_0" faceB="FACE_BOTTOM"
                angle="0"/>
  </connections>
</eve>
```

Each module has its own alias and this alias is then used to describe the connections. The user do not have to follow a special order to specify the connections. He must just avoid creating connections which produce cycles as we will see later.

Aligning connectors

Each face has a normal in the object space, an additional *up* vector is necessary to define a complete referential[4]. The center of each face is also known. The alignment between the face A and face B is based on quaternions. Quaternions are a convenient way to rotate objects around an axis. They are an extension of complex numbers with one real component and three imaginary values. The imaginary values can be compared to an axis in 3D and the real value is the angle of rotation around this axis. One advantage with quaternions is that they avoid the *gimbal-lock* problem caused by Euler angles[5]. A gimbal lock occurs when two main axis are aligned, this is caused by the successive evaluation of the x, y and z rotations with Euler angles. In some cases, one reference is canceled and the direct consequence is a wrong rotation.

For more information about quaternions, the reader is invited to read the pages available at this link [1].

The normals are $\vec{n}_A$ and $\vec{n}_B$. The *up* vectors are $\vec{up}_A$ and $\vec{up}_B$. Aligning two faces is performed via the following operations :

1. Normalize all normals and *up* vectors
2. Compute the crossproduct $\vec{n}_{AB}$ between the two normals $\vec{n}_A$ and $\vec{n}_B$
3. Compute the angle between the two normals : $\theta = acos(\vec{n}_A \cdot \vec{n}_B)$
4. Define a quaternion q_1 with angle θ and axis $\vec{n}_{AB}$

[4] the third *side* vector can be obtained via a cross product

[5] Euler angles define angles on the x, y and z axis

5. Transform the referential of both faces using the quaternion q_1. The faces are now aligned but the angle of rotation around the connector is still undefined.

6. Compute the angle between the two *up* vectors : $\rho = acos(\vec{up}_A \cdot \vec{up}_B)$

7. Define a quaternion q_2 with angle ρ and axis $\vec{up}_{AB}$

8. Transform the referential of both faces using the quaternion q_2. The faces are now completely aligned and the angle of rotation around the pivot axis is consistent.

9. Displace the second module to match the position of the first face.

Note that the description above lacks some tests in case of degenerated results (colinear normals or up vectors). These tests are actually performed in the code.

Restriction on cycles

The only restriction with our system is that cycles are not allowed. Robots based on cycles only represent a small part of all possible configurations. We also noticed that a configuration such as the "wheel" (Elmar Dittrich) heavily rely on the fragility of the velcro connections. With robust connections and stiff servos, getting the same effect would need an accurate synchronization between the modules. We consequently decided to discard this type of configurations.

3.7 Communication between Bluemove and Eve

Bluemove is completely programmed in Java (JDK 1.5) while Eve is in pure C++. Interfacing between two applications written in these languages is possible using different methods. We will now discuss about the various choices at disposal and what we decided to use to send data between Eve and Bluemove. Our main criteria were simplicity and ease of integration in both Java and C++ languages.

3.7.1 Java RMI

Java provides its own *remote method invocation* implementation : Java RMI. Without an additional Corba-related layer called IIOP (Internet InterORB Protocol), the original RMI is incompatible with applications which were not written in Java. We left this solution apart due to its complexity (see the next section about Corba). A server first needs to launch a naming service (comparable to a port listener) and then the server itself can be executed. The client will then have to connect to the reference provided by the naming service.

3.7.2 Corba

Corba (Common Object Request Broker Architecture) is a standard for interoperation between various languages and different operating systems. It allows remote invocation of methods and access to objects which are distributed between applications. Objects, methods and variables are defined using IDL (interface definition language). These standard specifications are then mapped to an implementation (Object Request Broker) which will interact with the native language (C, C++, Java, Python, etc.).

Corba is powerful but adds a lot of complexity to the whole system. As we had only a limited experience with Corba, we preferred to find a simpler solution.

3.7.3 Shared memory

This solution uses part of the memory to store variables which are shared between the two applications. Using JNI (Java Native Interface), it is possible to write some code in C that will manage the memory mapping. The main problem with this solution is inherent to JNI, some non-portable low level code is necessary to expand the restricted API of Java.

Another problem is that both applications must run on the same station, with the same operating system. But shared memory is very fast and if the real-time constraints were higher, it would have been a possible candidate.

3.7.4 Sockets

Sockets are basic communication primitives that open ports where one can write or read data using a TCP/IP protocol. For example, a server creates a socket on port 1234 and the client an access to the port 1234. It can also specify an IP address, it is hence quite easy to make an application that will be deployed in a network. Sockets are available in most programming languages and several free libraries provide the needed functions. The Java API has a namespace dedicated to communication and a complete library to handle sockets by the mean of data streams.

The main problem with sockets is that they are operating at a low level and a real protocol must be added. They were our main solution until we find the XML-RPC protocol.

3.7.5 XML-RPC

This is the solution we adopted for the interoperation between Eve and Bluemove.

XML-RPC is similar to Corba or Java RMI except that the protocol uses calls encoded in XML over HTTP. It is also much simpler than Corba[6]. XML-RPC was first created by Microsoft which then evolved it to a more advanced version (SOAP) [18]. XML-RPC is available in many popular languages, most implementations are free or open source libraries. For Bluemove, we use the *Apache XML-RPC Java* implementation, Bluemove operates in client mode. Eve is the C++ server, XML-RPC is provided by a library on Sourceforge called *XML-RPC for C and C++*.

XML-RPC basically allows one to perform remote method invocation with a set of parameters. The client communicates with the remote server by sending XML documents to an address such as : "http://128.178.45.154:9999/RPC2".

The methods are registered by the server, each function receives a name and a pointer to the function is kept. The XML-RPC server is then launched in its own thread. In Eve, we have for example a method called "eve.ping" which accepts two integers and returns the sum of these two values.

Several types have been defined, they are sufficient for most applications :

- array (composite)
- base64
- boolean
- date/time
- double
- integer
- string
- struct (composite)

The composite types can contain other types. In other words, it is possible to create an infinite set of parameters. For Eve, we only needed an array containing the information about each module (string for the address and double for the position).

Setting up the parameters and executing a call from Bluemove is quite simple, one just needs to create a vector with all parameters, provide the URL and the remote method name and execute the call. When the server receives the XML document, it dispatches the call to the proper function which will then "decompose" the parameters and store them in variables. The server and client must agree on the types of the variables otherwise cast errors happen.

On the whole, it would be straightforward to add new methods if new features would be added to the simulator. One drawback of XML-RPC is that it needs more bandwidth than conventional RMI due to the XML encoding. Our experience with the protocol shows it is not a concern with a good network. A more crucial point with the simulation in mind is the ping value between the client and the server. Delays have a direct impact on the response of the simulator. High and varying pings will produce jerky motions in the simulator. This is the case when using two computers over a wireless link.

[6]The specifications of XML-RPC lies on a few pages while Corba requires several books

Chapter 4

Bluemove, further improvements

4.1 Introduction

Bluemove is the application we developed during our first project on YaMoR. This software has been rather well tested during the project and the inauguration of the IC building at EPFL (see Chapter 2). Every time, Bluemove gave us entire satisfaction. A screenshot of this application is shown on Figure 4.1.

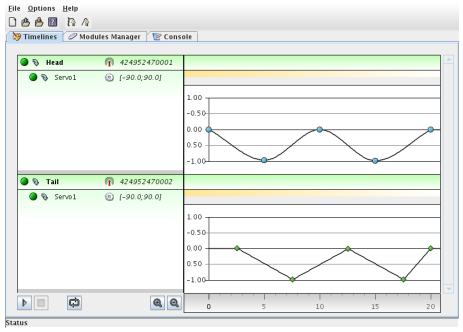


Figure 4.1: Bluemove, the YaMoR controller software

We already planned lots of features but had no time to implement them during the winter semester. We preferred to have a more solid framework rather than a weak structure with several half-finished features. One of the goal of the summer project was to add some of these features.

As of the end of the winter project, Bluemove allows the user to

- easily draw trajectories with the mouse
- inquiry for available YaMoR units
- create the description of the modular robot
- view the log messages printed in a nice console

These are the minimal features required in order to efficiently build modular robots. In the next sections, we will discuss in more details the new features available in Bluemove.

4.2 Possible new features

Right after the inauguration of the IC building (see Chapter 2), we did a meeting with Auke, Andres and Alessandro to talk about the directions to take during this semester. One of them was to implement new features into Bluemove. The following list summarizes the possible features that emerged from the meeting. We already had most of them in mind during our first project but they were not clearly defined.

- blocks in the timelines (loops, ping-pong, etc)
- real-time control of the units
- plugins
- data import/export
- integration with a simulator
- scripting
- FPGA programming with Bluetooth

We chose to implement real-time control, plugins, scripting and the integration with a simulator. FPGA programming is part of another semester project [12] but was also integrated into Bluemove. Data import/export did not seem very useful for the moment and would have served the same purpose as the

integration with a simulator.

The next section will cover the implementation of these features into Bluemove.

4.3 Implemented features

4.3.1 Blocks

Due to lack of time and redundancy with the real-time module, we decided to discard this feature and concentrate on more important features. It is still possible to produce loops or other timing blocks with dedicated plugins inserted in the control tree. For example, a plugin could act as a multiplexor that would switch between many inputs according to keys pressed by the user, by looking at a counter or at the current frame.

We also think that the purpose of the timing blocks was to allow using different gaits. Blocks in the timelines could not change during runtime while blocks in the real-time control are subject to actions performed by the user. The interaction with the user and the robot is coarser with fixed timing blocks in timelines, an issue that is solved with the real-time control.

4.3.2 Function generator

At the end of the winter semester, one feature was cruelly missing in Bluemove: a mathematical trajectory generator. We noticed that during our tests, we often had to draw sinusoidal signals with shifted phases/frequencies. This was a tedious task and each time we wanted to change the frequency of a module, we had to completely redefine its keys.

Using a free project called *Math Expression String Parser* (MESP) [16], we could add a simple but effective mathematical generator. MESP allows writing the expressions in a handy way compared to Java syntax. For example in Java, one could write `Math.sin(x*x*x+Math.PI)`. In MESP, the later is written using $sin(x^3 + PI)$ which is closer to what is usually seen in mathematical softwares such as Matlab, Octave or Mathematica. MESP allows mapping a numerical value to a string. It is then extremely easy to add constants and new variables which will then be mapped to their textual counterpart. Evaluating the formula is a matter of calling a single function and we really enjoyed the way MESP was designed. It helped us a lot during the inauguration (cf. Chapter 2). We could quickly set up new trajectories in a few seconds and demonstrate to the audience the results on the robot.

4.3.3 Plugins and real-time control

Bluemove allows to draw trajectories for each unit. The user can change the keys positions while playing allowing a kind of real-time control of the modular robot. We are able to reproduce basic gaits or speed control as demonstrated on the serpent video [7]. However, these manipulations are quite limited as it is only possible to move one trajectory key at a time.

The idea of plugins and real-time control is to have higher level functions which act on the whole curve or a set of curves. For example, the user could set an amplitude filter on a given trajectory. We can go a step further and change the amplitude parameter while playing the timelines. This parameter could be obtained from another curve output (coupled oscillators) or from an user input such as keyboard or mouse events.

With the combination of multiple filters, inputs, generators and outputs, it is now possible to control the movements of a modular robot in a very flexible way.

Core

This is the main part of the plugins system. Even without the graphical user interface (see below), it would be possible to setup a fully working plugins graph. As this part really needed to be well designed, we spent quite a lot of time in order to find the best classes organization. The final model is presented on Figure 4.2 in an UML like diagram.

There are mainly three singletons: `PluginsFactory`, `PluginsManager` and `GraphManager`.

The `PluginsFactory` is responsible for the creation of the plugins. It receives the

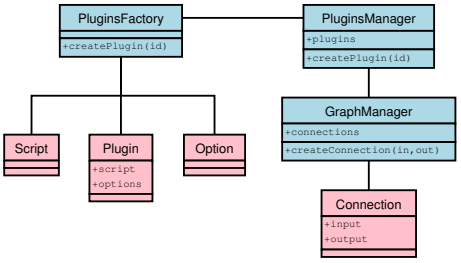


Figure 4.2: Plugins design

plugin identifier and generates it by taking the information from a XML file (see below). In order to create a plugin, the `PluginsFactory` generates the needed `Option`, `Script` and `Plugin` instances. It then merges the options and the script into the plugin. It is important to notice that each generated plugin is different from another plugin of the same type. In other words, two plugins of the same type do not share any data. Some objects should have been shared. For example, the script object is always the same for a given type of plugin. But for sake of simplicity, we have decided not to do it. `PluginsManager` also implements a kind of cache. For example, when trying to retrieve a script, a query is sent to the cache. In case of cache miss, the script is read from the XML file. Otherwise, the cache returns a new instance of the script object. The same mechanism happens for every components of a plugin such as the type, the inputs or the outputs.

The `PluginsManager` takes care of the created plugins via a linked list. It also redirects the plugin creation requests to the `PluginsFactory`. When the user edits a script, the `PluginsManager` is responsible of the propagation of the modifications. This is due to our choice of not sharing objects between plugins.

The `GraphManager` is charged of the creation and maintenance of the connections between the plugins. It is also responsible of the execution of the scripts in the right order and propagate the data between plugins. This is not a trivial task and requires good knowledges in graph theory. However, by adding some constraints, it is possible to implement simple algorithms which will explore the graph the right way. Moreover, we needed a quick algorithm in order to calculate the plugins transformations between two steps without significant slowdown.

We added the following constraints.

- Only one connection per option.
- No cycles allowed in the graph.

The first one is not a limitation but rather a simplification in the graph management. This is the case not only for the programmers but also for the user who does not have to think about how many connections he can set for a given option. It also avoids to deal with a crowded tree which would be difficult to visualize.

The second constraint is a limitation which allowed us to implement a quick and simple algorithm for graph evaluation. However, part of this limitation can be circumvent. In fact, a feedback loop can be useful so as to memorize a state by propagation to the next step. For example, state machines, which are often used in algorithms, need such a feature. It is implemented in BeanShell using static variables. The idea is to use persistent variables that keep their value between each call to the `eval()` method. We thus set the default value of each persistent variable before the first evaluation of the scripts. Afterwards, the last value of the variables are used in the next evaluation step of the plugin. For example, assume a variable named `foo` with a default value of `0`. A script performs a simple incrementation and prints the result, say `foo++; print(foo);`. On the first iteration, the printed value will be `1`. After the next step, the value of `foo` is still memorized and the script will print `2`.

The algorithm is described in Algorithm 1.

We first add the output plugins to the execution list and set their state to ready. These plugins are the main sources of data and do not depend on others plugins. Then, we iterate over the plugins until all of them are ready. A plugin is ready when all of its predecessors are already in the ready state. It means that all the required inputs are updated and that the plugin has all the data needed to compute its own outputs. An infinite loop in the *while* statement indicates the presence of one or more cycles in the graph. This is checked before the start of Algorithm 1.

Algorithm 1 Graph evaluation

```
clear the list execList
for all plugin in plugins of type OUTPUT do
    set plugin ready
    add plugin to execList
end for
while not all plugins are ready do
    for all plugin not in execList do
        if all plugins connected to the inputs of plugin are ready then
            set plugin ready
            add plugin to execList
        end if
    end for
end while
```

A simple optimization is to run the Algorithm 1 only once and store the plugin order into a list. Then, a simple iteration over this list will evaluate the plugins in the right order. We implemented this optimization into Bluemove.

We also added the possibility to run the script more than one time per step. This can be really useful when using the plugins system as a trajectories generator because you often need to integrate values. In order to avoid too high integration steps which make the integrator less stable, a possible solution is to run the script several time per step using a small delta.

Several solutions were discussed for the script implementation. We wanted an easy scripting language, a lightweight and well-documented interpreter offering good performance. Four solutions were considered.

- Jacl: Tcl Java implementation [4]
- Jython: Python Java implementation [9]
- Rhino: JavaScript Java implementation [2]
- BeanShell: Java source interpreter written in Java [13]

Jacl is a Tcl interpreter. As Tcl is not widely used and we lacked good knowledges in this language, we quickly skipped this solution. Moreover, Jacl performances seemed to be really poor.

Jython would have been a more reasonable solution. The Python language is really efficient for scripting. Despite all its advantages, Python is still marginal and the goal of Bluemove was to facilitate as much as possible its usage. On the other side, Jython looked like a convenient solution if speed was a concern.

Rhino is the JavaScript interpreter of the Mozilla project. Lots of people associate JavaScript with web page programming. But this scripting language can be used for application scripting with the help of Rhino. Rhino was a good candidate for our needs. JavaScript is widely used and quite similar to Java. Moreover, Rhino performances are really good but it is quite complex to integrate and use. We preferred to focus on a solution with a small learning curve.

We finally chose BeanShell. BeanShell is a scripting language for Java. It has been recently approved by the Java Community Process as the standard JSR-274. It will be thus integrated in the future J2SE. Its parsing and interpreting performances are not as high as the routines provided by Rhino but they are nonetheless reasonable. BeanShell is really easy to integrate into a Java application too. The script is a subset of the Java language except for some handy shortcuts which allow the user to quickly write scripts. For example, one do not need to declare variables. With BeanShell, a simple routine will be written this way:

```
i = 10;
while(i > 10) {
    print(i);
    i--;
}
```

In pure Java, the same example would have been written as follow:

```
int i = 10;
while(i > 10) {
    System.out.println(i);
    i--;
}
```

Of course, BeanShell perfectly interprets the pure Java form. As it optionally accepts typed variables, we did not set a specific type to the plugin option. We thought BeanShell would make the necessary casts for us. In practice,

it did not work as well as we expected. To handle these conversions, we introduced a type for each option. The user can choose between the following types:

- int
- float
- double
- char
- String
- Object

Additional types can easily be added but the ones we are providing should meet most user needs. It is always possible to use the `Object` type for any other type. The user must check that the corresponding cast is correctly done in the script. The input and output plugins for the modules always take a `double` value for their actuators.

Two variables are global and can be accessed in any script. The first one is the variable named `t`. It represents the current frame. This is useful when the user wants to generate trajectories with a plugin. A possible script example could be `out = 50 * Math.sin(t);`. Another use of this variable could be to generate an event at a given frame. A simple trajectory multiplexer could be `if (t < 100) out = in1; else out = in2;`. This can replace the blocks as explained in Section 4.3.1. The second one is an array of 256 boolean values which contains the state of the keys. This variable is called `keyTable`. Thus, a plugin can react to the user inputs. The following script shows a possible example.

```
if (keyTable[(int) ','']) {
    offset -= 1;
}
else if (keyTable[(int) '.']) {
    offset += 1;
}
out = 100 * Math.sin(t) + offset;
```

In order to catch the key events, the mouse must be on the plugins panel. A global key listener could have also done this task but we found that catching the key events only when the plugins panel has the focus was a cleaner solution. It also allows the user to edit preferences while playing without interfering with the real-time control system.

GUI

Bluemove was designed with simplicity in mind. So we had to find a user friendly interface for manipulating the plugins. Our first idea and probably one of the best was a graph representation as shown on the Figure 4.3.

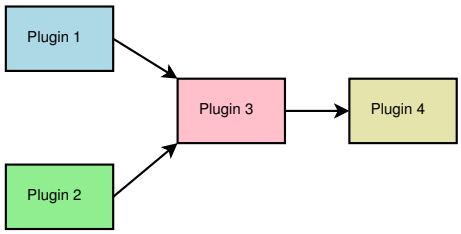


Figure 4.3: Graph representation

This kind of representation allows a good visualization of the data flow and an easy edition of the connections between plugins. However, it is not the easiest user interface to program. There are lots of graphical components to draw and several mouse events to take care of. Invalid user operation such as connecting an input with another input should also be detected during the graph construction. All of these operations are not trivial.

An alternative solution would have been a scripted language to describe the different plugins and their connections. This solution is a way easier to implement but is more difficult for the user to learn. This is definitely not the philosophy of Bluemove.

We looked at different softwares which use this kind of graph representation. Figure 4.4 shows NML [5], a texture generator developed by Kévin Drapel. NML takes advantage of graph representation to easily generate complex textures. The user can choose between several kind of plugins such as function generators, filters or blending boxes. The texture is then saved as a graph representation and a small decoder is available to regenerate the texture in any C/C++ software.

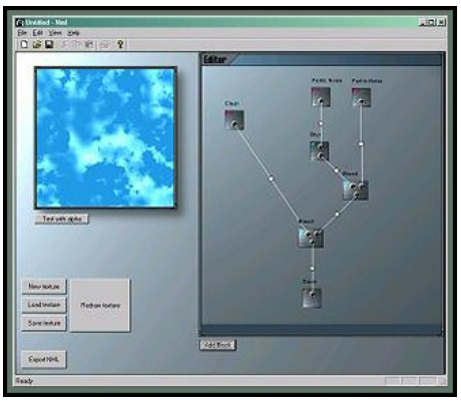


Figure 4.4: NML, a texture generator

Another software which uses a graph representation of plugins is Virtual Waves[17]. It is an audio generator. The idea of this software is quite the same as NML except that the user creates sounds. Figure 4.5 represents a screen shot of this application.

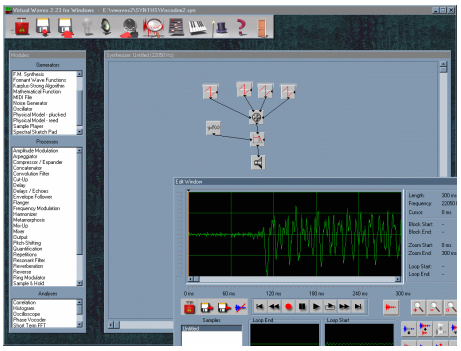


Figure 4.5: Virtual waves

After playing a bit with these two applications, we were convinced that such a interface would perfectly match our needs in terms of usage simplicity and representation power.

So, we tried to implemented the same graphical user interface into Bluemove. The panel is composed of two parts. The first one, on the left, shows the available plugins using a tree. The tree representation was chosen because it is already used in the modules manager (cf. [6]). Thus, this interface is familiar to the user. The second one, on the right, is the panel dedicated to the construction of the graph. You can see these two parts on Figure 4.6.

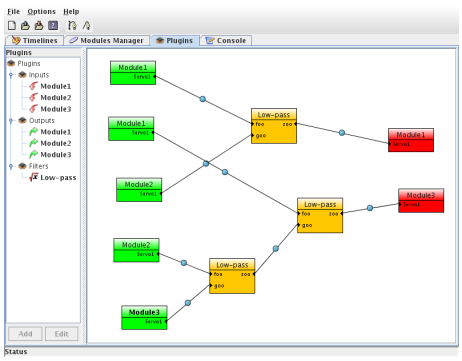


Figure 4.6: Plugins in Bluemove

This interface is a front end to the core of the plugins system. Please refer to the previous section 4.3.3 for more details. Thus, it is possible to easily implement a new graphical user interface. In Bluemove, we focused on a good separation between the core and the graphical interfaces.

Every time the user adds a plugin to the graph panel, a core and GUI plugin are created. The GUI plugin has a reference on the core plugin so as to propagate the changes made by the user. This also applies to the connections. Possible user interactions are listed below.

- create/delete/modify plugins
- add/remove plugins to/from the graph
- create/delete/modify connections between plugins
- edit plugins parameters and script

The script editor allows the user to create and modify plugins. You can see a screen shot on Figure 4.7.

It is possible to change the name of a plugin, to create or delete options, set their type and default value. In order to provide a more usable editor than a simple textarea component, we integrated an editor from the UJAC project [11]. This editor gives basic features such as undo, insert mode or highlight of the current line. A syntax highlight mode for BeanShell would be possible but we had not enough time to implement it.

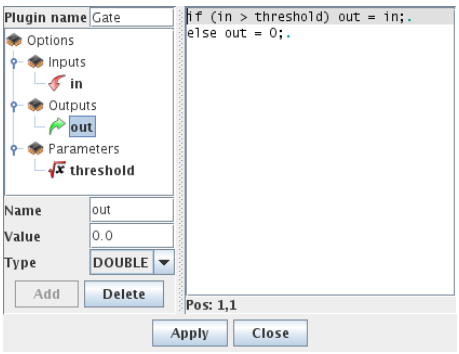


Figure 4.7: Script editor

Persistence

Every configuration file in Bluemove is a XML one. You are invited to refer to [6] for more information. Each plugin is saved with the project. This allows the user to design project specific filters without having to care about the availability of his plugins. The disadvantage is that the project files take more disk space. But this is insignificant as the project archives are packed.

There are three configuration files for the plugins system. The first one takes care of the persistence of the plugins themselves. Practically, it serializes the `PluginsFactory`. The second one saves the user graph with all its parameters. It serializes the `PluginsManager` and the `GraphManager`. Below is an example of the `PluginsFactory` file.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<bluemove>
  <plugins>
    <plugin id="bd493121-fa26-46a1"
      name="Adder" type="STANDARD">
      <input name="foo" value="0"/>
      <input name="goo" value="0"/>
      <output name="zoo" value="0"/>
      <param name="state" value="start"/>
      <script><![CDATA[zoo = foo + goo;]]>
      </script>
    </plugin>
  </plugins>
</bluemove>
```

The script depends on the plugin and not on its instance. For example, you can add two `Adder` plugin to the project. If you decide to change the script to `zoo = foo + goo + 10;` then the two instance of the plugin will add 10 to the output `zoo`. But the default values of the parameters of a plugin are instance related. For example, you can set the initial value of `foo` to 0 for the first instance and set it to 10 for the second instance. These data are stored in the second XML files as shown below. Notice that this file is not valid. It is just an illustration of the possibilities.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<bluemove>
  <plugins>
    <plugin id="d17e6537-4cd7-4987"
      type="bd493121-fa26-46a1">
      <option name="foo" value="-50.0"/>
      <option name="goo" value="123"/>
      <option name="zoo" value="73.0"/>
      <option name="state" value="start"/>
    </plugin>
    <plugin id="3b018a7d-7d1f-4df0"
      type="Module2-output">
      <option name="Servo1" value="-50.0"/>
    </plugin>
  </plugins>
</bluemove>
```

The `option` tag represents the initial value of the given option. For consistency's sake, the `GraphManager` is stored into another file as shown below.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<bluemove>
  <plugins>
    <plugin id="d17e6537-4cd7-4987">
      <pos x="204" y="275"/>
    </plugin>
    <plugin id="3b018a7d-7d1f-4df0">
      <pos x="351" y="158"/>
    </plugin>
  </plugins>
  <connections>
    <connection oin="Servo1" oout="foo"
      pin="3b018a7d-7d1f-4df0"
      pout="d17e6537-4cd7-4987"/>
  </connections>
  <graphmanager>
    <steps value="10"/>
  </graphmanager>
</bluemove>
```

The persistence of the plugins system take advantage of the XML API already available in Bluemove.

4.3.4 FPGA programming with Bluemove

Jérôme Maye did his semester project [12] about programming FPGA over a Bluetooth connection using the YaMoR platform. The main idea was to send the bitstream to the Bluetooth board and store it into the memory. Afterwards, the ARM processor programs the

FPGA using the JTAG protocol. All these operations are done with a modified version of the Zeevo Zerial software. Thus, we had nothing to change with the Bluetooth communication system in Bluemove. Moreover, this allows us to configure the FPGA directly from Bluemove. Figure 4.8 represents the packets sent to the Bluetooth board in order to program the FPGA.

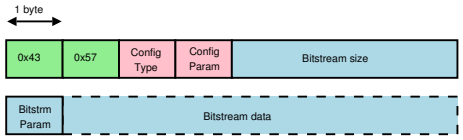


Figure 4.8: FPGA programming packets

Two bytes that contain an unique sequence are first sent. The modified Zerial detects them and switches to FPGA programming mode. Then, the configuration mode and its parameters are sent. The next four bytes contain the bitstream size. A last bitstream header is sent followed by the data. You can find more information about this protocol in Jérôme's report [12].

For each module in Bluemove, the user can load a specific bitstream. Afterwards, all bitstreams are sent with a single manipulation.

This wireless programming is a great improvement for the YaMoR project. An extra computer with the Xilinx software was previously needed to configure the FPGA. It is now possible to work with the YaMoR units with a single computer. One just has to store the `.xsvf` bitstream file somewhere on the hard disk of the computer.

Chapter 5

Hardware issues and improvements

5.1 FPGA problems

We only had one problem with the new FPGA board. Some FPGA designs seemed to work but most of them hung up the FPGA. After more investigations, we found out that a pin of the reset button was soldered on the new cards but not on the old ones. This caused the reset signal to be always asserted. Depending on the FPGA design, this produced a constant FPGA reset. You can see the pin which must not be soldered on Figure 5.1.

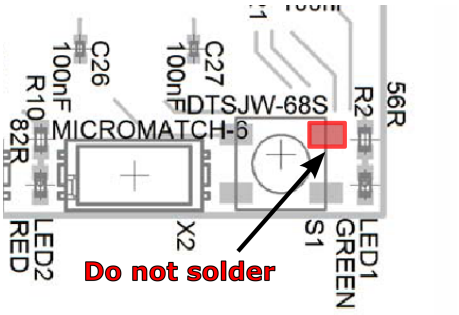


Figure 5.1: Not to solder pin

5.2 Bluetooth problems

At the end of the winter semester, we could not communicate with some Bluetooth cards that had been recently soldered. They were detected by a mobile phone but the dongles failed to scan them. The problem was caused by a frequency shift, the communication was slighty out of the spectrum window for a normal Bluetooth communication. The mobile phone was probably more tolerant than the dongles. The Zeevo chip contains a bank of capacitors that allow fine-tuning of the frequency, 16 possible values can be used but we quickly noticed that it was not sufficient. After reading the data sheet, we discovered that two capacitors around the crystal oscillator had wrong values. We increased them from 15 pF to 18 pF as shown on Figure 5.2. This fixed the communication problems.

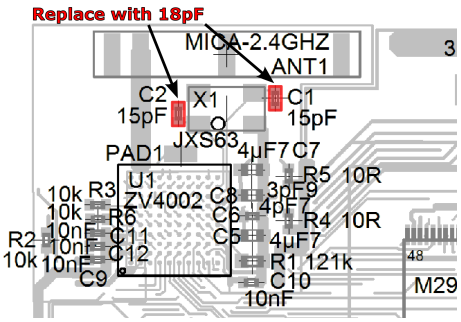


Figure 5.2: Capacitors to be replaced

5.3 Power supply problems

We also had problems with the newly soldered power boards. The `3.3V` and `1.2V` outputs did not work. We immediately thought that the `MAX1774` were defect. We spent a lot of time testing this component as well as the others. Finally, Fabien Vannel, once again, found the problem. A well hidden patch had been made under the `FDS8928A` on the previous board. Applying this patch on the new boards solved the problem. However, André Badertscher preferred to make a visible, clean patch instead of cutting the PCB track.

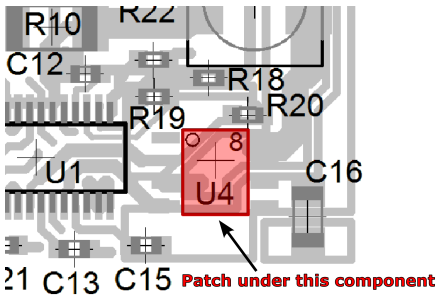


Figure 5.3: Patch under a component

5.4 Second generation

It is clear that the first prototype exhibits many problems. We have listed some of them in this report and our document of last semester [6]. After talking with the different people who were involved in the project, we have defined some ways that should be taken for the next version of YaMoR. The main trend is to reduce the size of the components as sensors will be probably added in the second generation.

Here is a non-exhaustive list summing up improvements that could be achieved on the hardware. We are just providing guidelines for people who would work on this project in the future.

5.4.1 Replace velcro connectors

As already discussed in the chapter about the inauguration of the IC building, the velcro connections are weak and do not allow reconfiguration. A stronger and better solution is highly recommended for the second prototype. Ideas with magnets or mechanical fixing were already discussed but need more investigations. Several solutions are possible and one of the most important question has still to be answered: do we need self or manual reconfiguration?

5.4.2 Change the way the batteries are charged

During fall 2005, a fire incident was caused by batteries which were charged in serial. A battery exploded inside a robot and burnt part of a room at ASL laboratory. Fortunately, people who were nearby could restrict the fire. These batteries were similar to the ones we are using in YaMoR.

After months of discharging and recharging cycles, a battery may shortcut and behave like a simple wire. As the second battery is connected in serial with this first battery, this single battery receives the voltage for two batteries. This extra charge may lead to explosion and fire. It is hence **strongly recommended to stop charging batteries which are in serial** unless there is a way to perform the same operation in a parallel way. Old batteries should be replaced after a few months to avoid these incidents.

For security reasons, the charging protocol is something that must be addressed as soon as possible. For the second generation, there should be a way to separate the batteries without opening the case and charge them in parallel.

5.4.3 Smaller batteries

Manufacturers now sell smaller batteries with similar electrical properties. As reducing the size of the components inside a module is necessary if we want to add sensors, a new battery model is something to consider for the second generation.

5.4.4 Small servos

Although it is true that the current servo used in YaMoR is inexpensive and provides enough torque for most configurations, one must admit that it is quickly damaged. A better, smaller but more expensive servo would be probably a wise choice as it would leave more room for other components and would need less replacements compared to the cheap servos. At the end, the advantages brought by another servo is probably worth the extra price.

5.4.5 Wires and soldering issues

Due to the restricted room available in the module, wires of the power board hardly fit in the small gap between the servo and the top of the module. We encountered many problems related to bad electronic contacts, wires with soldering that would break after a while or shortcuts at worst. A new solution should be con-

sidered for the second generation. The boards could be connected using flat parallel bus cables.

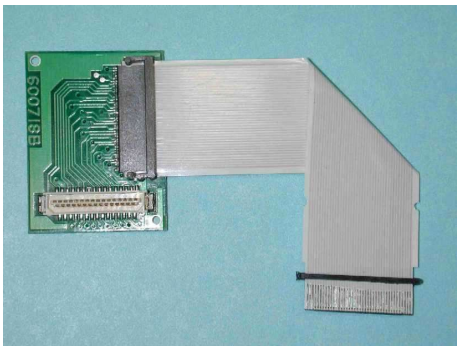


Figure 5.4: A (large) flat cable. Narrow cables could replace the wires in YaMoR.

Another solution consists in using *wires to board* connectors as shown on Figure 5.5. The boards could form a stack.

Figure 5.5: Sockets connectors

This approach allows unplugging a board from the rest and make a limited version of the module. Board to board connections are also more versatile as boards dedicated to specific jobs (DSP, other sensors, etc.) could be added to the rest of the components.

5.4.6 Multi-layers PCBs

André Badertscher stressed on the fact that it would be better to have multi-layers PCBs instead of two-sided PCBs. While they are more expensive, one board can contain more components and this would help to reduce the size of the module.

5.4.7 Sensors

During his semester project, a student in micro-engineering added sensors to our system. The idea was to design another board which contains the sensors and can be connected to our Bluetooth board. This sensor board was not designed to be used with the YaMoR units but rather for a distributed sensors network. However, the design can be adapted to meet the needs of the next generation of YaMoR units.

YaMoR units are built using PCBs and a promising way could be to directly integrate the sensors on the side of the module. Thus, a lot of place could be saved and the biological concept would be interesting.

Appendix A

Tutorial: Using Eve with Bluemove

This chapter quickly describes the user interface of the Eve application. It is divided into three parts: the first one explains how to install the application and its requirements, the second one gives an overview of the user interface and the XML data, the third part provides a step by step example which shows how to simulate a robot from Bluemove.

A.1 Compilation and installation

Compiling Eve requires a few libraries. They are provided with the sources in the /libs directory. Compiling under Linux is a matter of calling the following command in a terminal :

```
$ make distclean
$[ Cleaning ]
$[ Removing .depend ]
$[ HINT: Type 'make dep' to rebuild dep ]
$[ Removing covering informations ]
$[ Dist cleaning ]
$
$ make dep
```

The "distclean" command cleans all temporary and executable files (only those related to Bluemove, not the external libs). The "dep" command creates the dependency file which is useful for partial recompilation.

You can then build the executable using make:

```
$ make
```

The simulator can be launched with the following command:

```
$ make execute
```

This command is a shortcut for `"./bin/eve"`. Note that `"make execute"` will compile files if this has not been done before.

A.2 User interface

With the default settings, you should see a Quake map with the modular robot in the middle of the room. Moving the mouse while the left button is pressed will rotate the camera. The camera can be displaced by using the following keys :

q	move down
e	move up
a	strafe left
d	strafe right
w	move forward
s	move backward

On top of the screen, three buttons are used to control the simulator. The left button, "rotation", is used to change the orientation of the robot. The middle button, "translate" is used to change the position of the robot in space. The last button, "play", launches the simulation and waits for commands sent by Bluemove.

The "rotation" mode uses the following digits keys :

1	clockwise rotation on X axis
2	anticlockwise rotation on X axis
3	clockwise rotation on Y axis
4	anticlockwise rotation on Y axis
5	clockwise rotation on Z axis
6	anticlockwise rotation on Z axis
7	cancel all torques

The "translation" mode uses the following digits keys :

1	move forward on X axis
2	move backward on X axis
3	move forward on Y axis
4	move backward on Y axis
5	move forward on Z axis
6	move backward on Z axis
7	cancel all forces

The rotation and translation use torques and forces applied on the first module of the robot. The simulation is stopped when the rotation or translation modes are enabled. It is also recommended to stop sending new positions from Bluemove while the robot is moving or rotating.

Clicking on the "play" will activate gravity and the robot will fall on the floor with full collisions support. It is now ready to receive commands from Bluemove.

Pressing on "tab" key will slighty tilt the robot in case it gets stuck in a corner.

Pressing on "esc" key will close the simulator.

A.3 XML data

The configuration of Eve and the description of the robot are stored in XML files. They can be found in the /bin/data directory. The `"eve_config.xml"` file stores information about the 3D world, ODE parameters and XML-RPC server port. It is possible to change the resolution of the rendering window as well as the integration step of ODE.

The `"robot_desc.xml"` file contains the description of the robot topology with the various modules (Bluetooth address and alias). The connections are also described in a handy way. In the following XML, four modules have been defined with four distinct aliases. Three connections have been created. A connection must follow a strict format with the alias of the first module, the alias of the second module, the connector of face A, the connector of face B and the angle of rotation around the connector pivot.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<eve>
  <modules>
    <address value="424952470001" alias="head"/>
    <address value="424952470002" alias="body1"/>
    <address value="424952470004" alias="body2"/>
    <address value="424952470005" alias="tail"/>
  </modules>

  <connections>
    <connection moduleA="head" moduleB="body1"
                faceA="FACE_0" faceB="FACE_2"
                angle="0"/>
    <connection moduleA="body1" moduleB="body2"
                faceA="FACE_2" faceB="FACE_3"
                angle="0"/>
    <connection moduleA="body2" moduleB="taile"
                faceA="FACE_0" faceB="FACE_BOTTOM"
                angle="0"/>
  </connections>
</eve>
```

Each face of a YaMoR module has its own connector as described on A.1. You must use these names for the faceA and faceB values inside a connection tag.

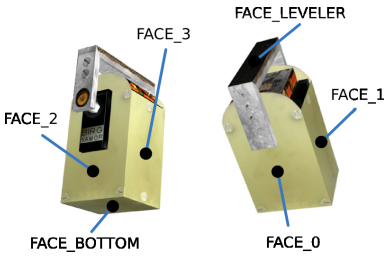


Figure A.1: The six connectors on a module and their respective names in the XML file

Please note that not all modules must be connected. A non-connected module will receive orders from Bluemove but will not interact with the robot.

All changes applied on the XML files require the simulator to be restarted. You can close it by pressing the "esc" key.

A.4 Step by step tutorial

In this section, you will get to know how to use the simulator and send commands from Bluemove. We will also modify the robot by adding a new module to the default configuration. We consider that you have already installed the application and that you are able to start it. We also assume that you have some experience with Bluemove.

A.4.1 Step one: setup the simulator and Bluemove

Start the simulator and introduces the address and port used by the simulator server in Bluemove. You will find the port of the server in the `"eve_config.xml"` file (default 9999). The address is either your IP or localhost if both the simulator and Bluemove run on the same station. You can use `"ipconfig"` on Linux to find your IP. The parameters window should be similar to what is seen on A.2. You must also activate the "simulator" device in Bluemove. You can activate another device while the simulator is running and test the result on both the real robot and the simulator.

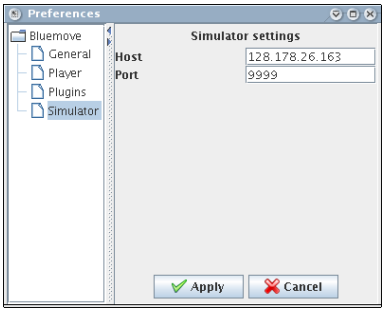


Figure A.2: IP and port of the server in Bluemove settings

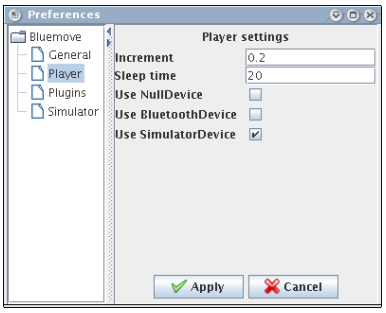


Figure A.3: Enable the "simulator" device

A.4.2 Step two: sending data from Bluemove

The simulator is ready to receive data from Bluemove. In Eve, press the "play" icon on the top of the screen, it should start to blink and the robot should fall on the floor.

Figure A.4: The robot before pressing the "play" icon

In Bluemove, by pressing the play button, the trajectories will be sent to the modules. You should now have a moving version of the robot. You can displace the robot while it is moving by clicking on the "rotation" or "translate" icons and then pressing the digits keys (see above).

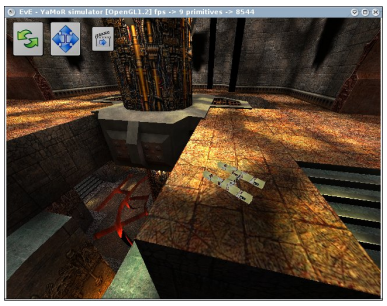


Figure A.5: The robot after displacement

A.4.3 Step three: editing the robot

The default robot is a spider-like creature. We will simply change the position of two modules on the rear legs by editing the file `"robot_desc.xml"`.

The modules will be connected to `"FACE_2"` instead of `"FACE_BOTTOM"`, this will produce two legs bent at 90°. Change the last four lines in the file such that the result looks like this :

```
<connection moduleA="tail" moduleB="legC1"
faceA="FACE_0" faceB="FACE_LEVELER" angle="90"/>
<connection moduleA="legC1" moduleB="legC2"
faceA="FACE_2" faceB="FACE_LEVELER" angle="0"/>
<connection moduleA="tail" moduleB="legD1"
faceA="FACE_2" faceB="FACE_LEVELER" angle="90"/>
<connection moduleA="legD1" moduleB="legD2"
faceA="FACE_2" faceB="FACE_LEVELER" angle="0"/>
```

Figure A.6: A new robot

Appendix B

Bluemove short manual 2

This chapter quickly describes the new features introduced in Bluemove v2.0. The first part gives an overview of the user interface and the second one provides a step by step example showing how to create a simple project with plugins. If you have not yet installed Bluemove, please refer to the *Bluemove short manual*. Moreover, you have to be familiar with Bluemove before starting this manual.

B.1 User interface

After starting Bluemove, you will see the main view of the application which contains four important parts:

- the menu bar
- the tool bar
- the tabbed pane
- the status bar

The tabbed pane contains several tabs which are respectively used to draw the module curves, to manage the modules, and to view the log messages. In the next section, we will take a closer look at a new tabbed pane added in Bluemove v2.0.

B.1.1 Plugins tab

The main new feature of Bluemove v2.0 is the plugins system. It allows the creation of plugins which can filter trajectories of the timelines tab or generate new ones from scripts. This system is really powerful and adds a new dimension in the use of Bluemove.

The interface is split in two parts. On the left, the plugins management which allows you to create new plugins, modify them and add them to the right part of the plugins system. This part, on the right, allows you to connect plugin outputs to other plugins easily. You can see an example on Figure B.1.

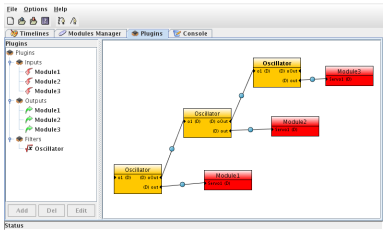


Figure B.1: Plugins tab

B.1.2 Script editor

Another important feature of the plugins system is the script editor. You can see it on Figure B.2. It allows the creation of the options of a plugin and the writing of the script. The script uses BeanShell and is hence quite similar to the Java syntax.

The interface is also split in two parts. On the left, the options management which allow you to add inputs and outputs as well as internal parameters to your plugins. On the right, the editor allows you to write the script that will achieve your needs.

B.2 Tutorial: a small example

In this section, you will get to know how to create a simple Bluemove project which take advantage of the plugins system. A simple oscillator will be used to create a basic locomotion

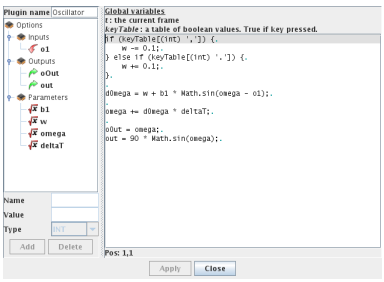


Figure B.2: Script editor

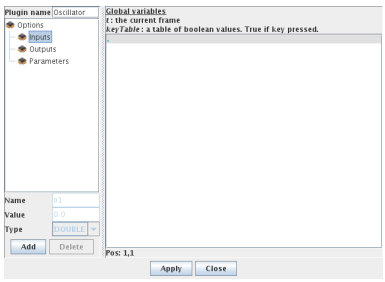


Figure B.4: Edit the plugin

movement. We consider that you already have installed the application and that you are familiar with it.

B.2.1 Step one: add a new plugin

First of all, start the Bluemove application. Add three modules in the Modules manager and switch to the "Plugins" tab. Your screen should be similar to Figure B.3. If you do not see the modules in the tree, expand the nodes by double-clicking on them.

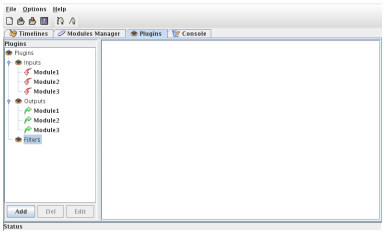


Figure B.3: Plugins tab with three modules

Now you are ready to create your first plugin. In order to do this, select "Filters" on the left and click on "Add". This action is shown on Figure B.3. A new plugin is inserted under the "Filters" node.

B.2.2 Step two: edit the plugin

Click on the newly created plugin and then click on "Edit". A dialog window will appear and you should get a screen similar to Figure B.4.

Change the name of the plugin and call it "Oscillator". Do not forget to click on "Apply" in order to save your changes.

B.2.3 Step three: add options to the plugin

In order to interact with other plugins, one needs to define input and output options. Create an input option by selecting the "Inputs" node and clicking the "Add" option. Then, select the newly created option and edit the parameters as describe on Figure B.5. Once you click "Apply", the screen should be similar to the Figure B.5.

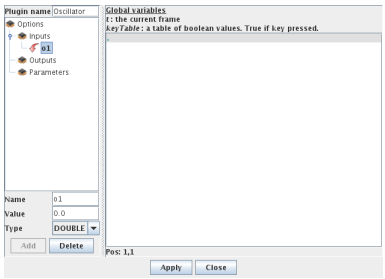


Figure B.5: Create an input option

Create an output option called "oOut" in the same manner. You should end up with a screen similar to the Figure B.6.

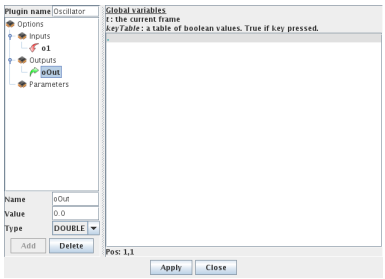


Figure B.6: Create an output option

Add another output option called "out" with the type "Double" and a value of "0". You can now create a parameter called "b1" and set it

with the value shown on Figure B.7.

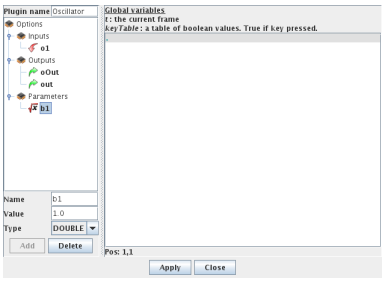


Figure B.7: Create a parameter

Finally, you can add all the parameters shown on Figure B.8. They are all of type "Double". You can set the value to "1.0" except for "deltaT" which needs a smaller value. Set it to "0.001".

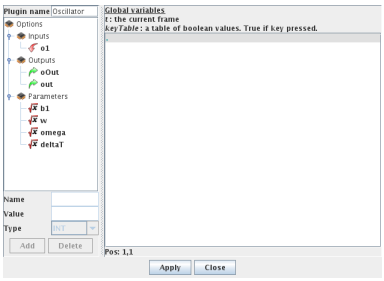


Figure B.8: Create parameters

B.2.4 Step four: write a script

You can now write the script of this plugin. Remember that we are trying to implement a coupled oscillator. Copy the script shown on Figure B.9. Note how the options are used. The third line is our simple integrator based on the explicit Euler integration.

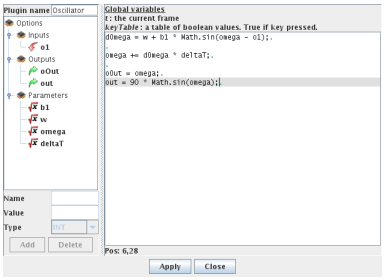


Figure B.9: Write a script

B.2.5 Step five: add real-time interaction

The script you wrote in the previous step is now fully functional. However, we want to show you another feature of Bluemove v2.0. Add the five first lines shown on Figure B.10 at the beginning of your script.

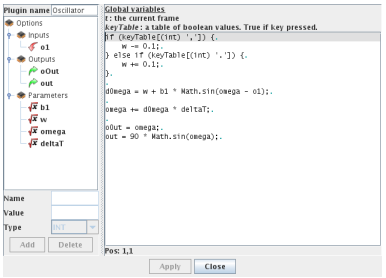


Figure B.10: Add real-time interaction

The variable `keyTable` is a special one. It is global to all plugins and contains the keyboard state. This allows you to create a real-time control of your plugin. In our example, pressing "," or "." while the system is playing will modify the variable `w`. You can therefore control the locomotion of your modular robot in real-time.

You can now click the "Apply" button and then "Close" button so as to return to the plugins panel.

B.2.6 Step six: add a plugin to the graph panel

The next step is to add our plugin to the graph panel, the right one. Click on the "Oscillator" node on the left and click on "Add". An orange plugin called "Oscillator" should appear at the top-left of the graph panel as shown on Figure B.11. You can move it by drag and drop.

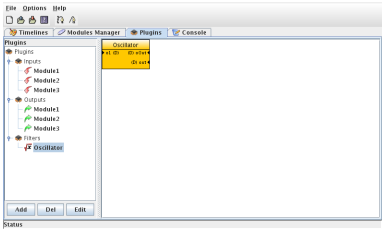


Figure B.11: Add a plugin to the graph panel

In order to complete our system, add three "Oscillator" plugins and three "Inputs" plugins for the modules 1, 2 and 3.

B.2.7 Step seven: connect the plugins

Once the plugins are placed on the panel, you can connect them. Click on the small triangle of a plugin option and drag the mouse to another plugin option. You should see a connection which follows your mouse while dragging. Remember that you can only connect options of the same type and only connect an input to an output. You should end up with a graph similar to the one shown on Figure B.1.

B.2.8 Step eight: delete plugin and edit default values

If you want to delete a plugin, you can simply right click on it. A popup menu will appear and you can click on "delete plugin" as shown on Figure B.12. The plugin and all of its connections will be removed.

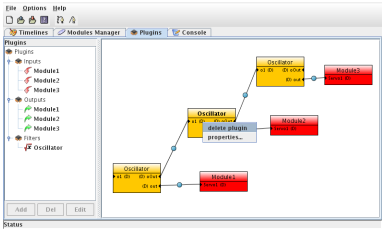


Figure B.12: Delete a plugin

You can notice that another menu entry called "properties..." is available. It allows you to set the default value of the plugin parameters. This dialog is represented on Figure B.13.

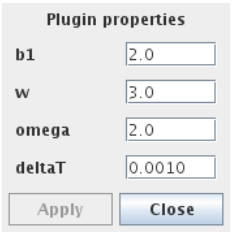


Figure B.13: Edit the default values

B.2.9 Step nine: delete connection

If you want to delete a connection, you can right click on the blue circle in the middle of the connection and click on "delete connection". You can also drag the connection by clicking on the corresponding option and drop the wire somewhere in the panel.

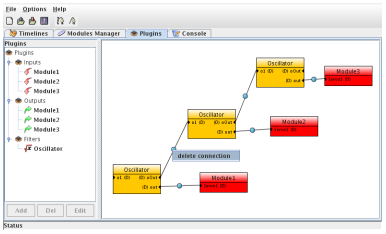


Figure B.14: Delete a connection

B.2.10 Step ten: evaluation steps

Our plugins system is now ready to be tested. We still have to set the evaluation steps. In order to integrate `omega`, we need to execute the script more than one time per frame. This parameter can be adjusted in the menu "Options", "Preferences..." and "Plugins". Set the "Evaluation steps" to "10". This will run the script ten times per frame.

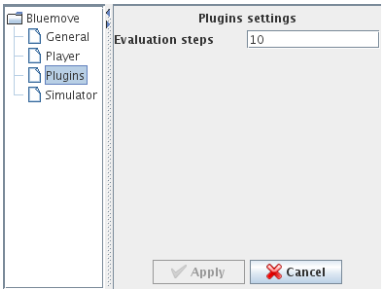


Figure B.15: Evaluation steps

B.2.11 Step eleven: set the FPGA bitstream

A new feature available in Bluemove v2.0 is the possibility to send the FPGA bitstream with Bluetooth. Switch to the "Modules Manager" tab and select "Module1". Click on the "..." button close to the "Bitstream" field. Select the "motor.xsvf" bitstream. Save your change with

"Apply". Your screen should be similar to Figure B.16.

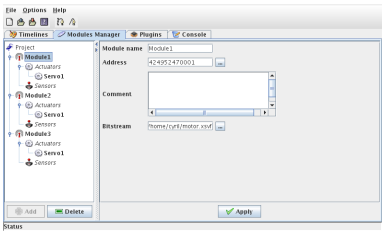


Figure B.16: FPGA bitstream

Repeat the same operation for every module. Click on the "Project" node in the left tree. You can now send the bitstreams by clicking on the "Send" button as shown on Figure B.17.

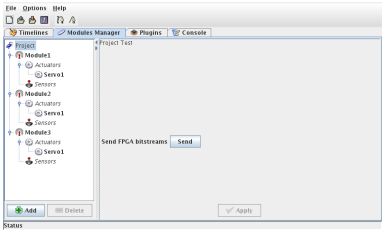


Figure B.17: Send bitstreams

B.2.12 Step twelve: play

You are now ready to click on the "Play" button available on the "Timelines" tab. Remember to switch the "Plugins" mode on by clicking on the fourth button near the "Repeat" one. Refer to the Figure B.18.

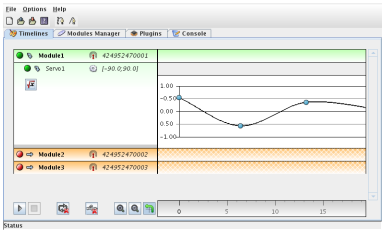


Figure B.18: Enable the plugins system

B.2.13 Optional step: function generator

The function generator is also a new feature of Bluemove v2.0. It allows you to draw trajectories in the "Timelines" tab with the help of mathematical functions. Click on the square root button in the "Module1" timeline. This button is visible on Figure B.18. A dialog will appear as shown on Figure B.19. You can enter a function and set the wanted parameters.

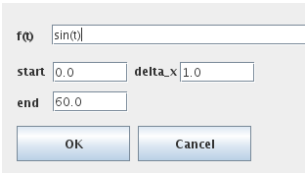


Figure B.19: Function generator

B.2.14 Conclusion

We briefly looked at the new operations of Bluemove. There are a lot of other features that were not described in this tutorial. However, you should be able to discover them on your own. Do not forget to periodically check the help website [6]. Lots of information on Bluemove are available there.

Bibliography

[1] Martin John Baker. Quaternions. `http://www.euclideanspace.com/maths/algebra/realNormedAlgebra/quaternions/index.htm`.

[2] Norris Boyd. Rhino: Javascript for java. `http://www.mozilla.org/rhino`.

[3] Prof Thomas Bräunl. Mobile robot lab. `http://robotics.ee.uwa.edu.au`.

[4] Mo DeJong. Jacl is an implementation of tcl in java. `http://tcljava.sourceforge.net`.

[5] Kévin Drapel. Nakedmonalisa. `http://icwww.epfl.ch/~drapel/softwares.html`.

[6] Kévin Drapel and Cyril Jaquier. Using bluetooth to control a yamor modular robot. `http://birg.epfl.ch/page56602.html`.

[7] Kévin Drapel and Cyril Jaquier. Video demonstrating trajectory changes with bluemove. `http://birg.epfl.ch/webdav/site/birg/shared/bluetooth/serpent-bluemove.avi`.

[8] Geir E. Hovland and Brenan J. McCarragher. A hidden markov approach to the monitoring of robotic assembly.

[9] Jim Hugunin. Jython is an implementation of python in java. `http://www.jython.org`.

[10] Andreas Koestler and Thomas Bräunl. Mobile robot simulation with realistic error models. *2nd International Conference on Autonomous Robots and Agents, New Zealand*, 2004.

[11] Christian Lauer. Useful java application components. `http://ujac.sourceforge.net`.

[12] Jérôme Maye. Fpga configuration with bluetooth. `http://birg.epfl.ch/`.

[13] Pat Niemeyer. Beanshell: Lightweight scripting for java. `http://www.beanshell.org`.

[14] Henrik Hautop Lund Orazio Miglino and Stefano Nolfi. Evolving mobile robots in simulated and real environments. *Artificial Life 2:4*, pages 417–434, 1995.

[15] Anil Seth. Noise and the pursuit of complexity : A study in evolutionary robotics. *Centre for Computational Neuroscience and Robotics, University of Sussex, Brighton*, 1998.

[16] Stormdollar. Mesp: Math expression string parser. `http://sourceforge.net/projects/expression-tree`.

[17] Synoptic. Virtual waves. `http://www.sonicspot.com/virtualwaves/virtualwaves.html`.

[18] Xmlrpc-c. Xml-rpc vs. soap. `http://xmlrpc-c.sourceforge.net/xmlrpc-howto/xmlrpc-howto-soap.html`.