Semester project report

Roombots Reconfiguration Inverse kinematics and Cooperative Movements Tasks

Manon PICARD

Supervisors :
Alexander Spröwitz
Stephane Bonardi

Abstract

Roombots are modular robots developed at BIRG, EPFL, to build adaptable furnitures

In this semester project, we improved the movements and trajectories of the Roombots meta-modules. We implemented a method for optimizing approaching method before locking to a external connector. We applied this method to handle passive plates.

Then we worked on collision cloud, to optimize their computation and decrease their size with the overall goal of avoiding obstacles during movement and reconfiguration. We developped and compared several methods.

Time constraints leaded us to abandon the cooperative movement parts to be able to go into the collision cloud task in more details.

Contents

1 Project's presentation

In this chapter, we offer a quick overview of the field of modular robotics (sect. 1.1). Then we present in more details the Roombots robot (sect. 1.2) and what has been done previously to control it (sect. 1.3). Then we present the project's objectives (sect. 1.4) and timeline (sect. 1.5).

1.1 Modular robotics

1.1.1 Definition and goal

A robot is usually defined as a machine that can achieve a goal (perform a task, clean a room, play soccer...). These robots are also called "monolithic robots". Its mechanical parts are designed so that it can achieve this goal as good as possible; it has sensors and actuators accordingly. But such a robot will be well-suited for only one task and cannot adapt to different situations.

Modular robots, on contrary, are designed to be very adaptive. Each robot is called a module. One module is a simple robot, with few sensors and actuators, pretty unable to achieve any task alone. But if many of this simple modules act together, they can reach complicated goals. Ideally, by adding the information of the sensors and the mobility of each robot, one can build a complex structure dealing with a large sensor network and able to execute complex tasks. One can compare modular robots with Lego games for example (Fig. 1). Each piece is simple but a lot of them allow to build astonishingly complicated structures. However, to be able to do that, a number of precondition must be completed to be able to build the structure and use it.

Figure 1: Lego bricks are simple but together, they can build complex structures [9].

Some of the main advantages of modular robots are ([5]):

- cost: All the modules are identical. They can be manufactured in big series and be therefore cheaper. However, as prototypes, as there are largely redundant, they can be pretty expensive.
- Resistance to failure: In traditional robotics, if one part of the robot fails, the entire robot fails. In modular robotics, if one module fails, the others

continue working. Moreover, the broken module can easily be replaced. This also reduces the maintenance costs.

Modular robots are a very promising branch of robotics if we want more adaptivity and robustness. However, no applications are currently available because of the difficulties of finding the right design and controlling the robot to achieve tasks, move and reconfigure in an efficient way. Moreover, for a given task, a monolithic robot is always better, so modular robots will be developed extensively when they will be able to perform efficiently tasks that can't be achieved by monolithic robots.

1.1.2 Chain-type and lattice-type modular robots

Modular robots can be divided into two major types, chain-type robots and lattice-type, according to their connectivity modes.

Chain-type robots have only few (most of the time two) connecting points and generally an elongated shape. They are assembled in chains to build branched structures. They are typically used to build locomotive robots (quadrupeds, snake-like...)

One example of chain like robots is the dof-box ([3], Fig. 2), developed at LASA, EPFL or its cousin, YaMoR ([13]), developed at BIRG. Each module has only one degree of freedom, but when many of them are combined, they can achieve locomotion in many different ways, quadruped walk, crawling, snake-like locomotion or even some kind of rolling.

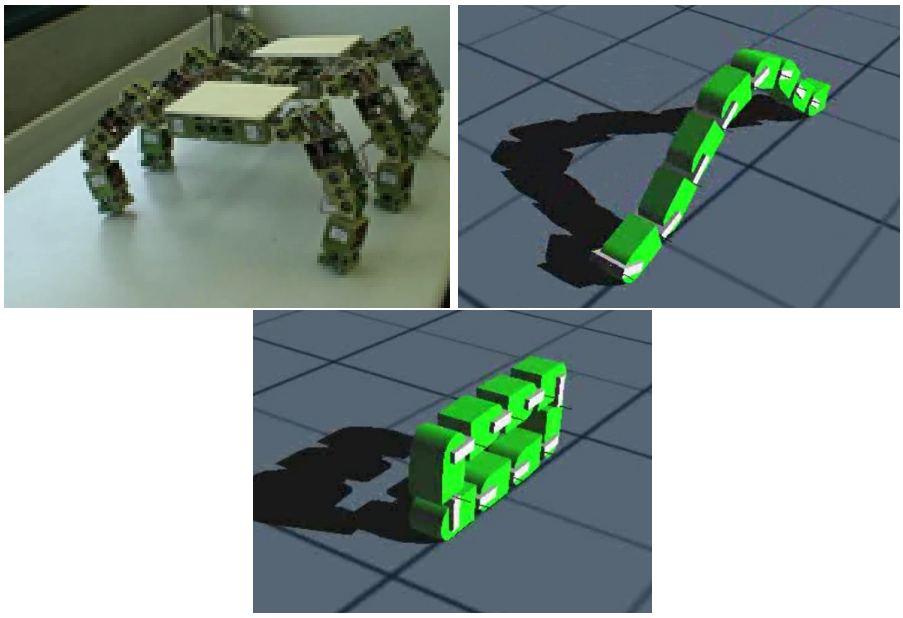

Figure 2: The dof-box module (left) and the YaMoR robot (right) ([1])

Other chain-type modular robots are for example Polypod ([19]) and CONRO ([15]).

Lattice-type robots have on the contrary many connectors and a cube or sphere-like shape. There is no clear separation between both classes of modular

robots however. Lattice-type robots can easily construct 2D or 3D lattices and dense structures. Their major advantage is the ability of self reconfiguration.

Indeed, if the robot can attach to and detach from the structure, we can go a step further in modularity: the robot can change its shape by itself. It can achieve locomotion by moving the modules in the back to the front of the structure (even if it is not very efficient in term of speed). It can adapt its shape to the task it has to accomplish or to the environment. Imagine for example a robot that has to do something in a collapsed or dangerous building. It takes a quadruped shape to go there, but has to go through a tiny space to enter. So it transforms into a snake, enters and transforms again to perform its task.

These modular robots offer a whole new range of possibilities, as a number of simple modules can achieve a lot of different tasks, depending on their shape and configure to adapt to their goals. Modular robots are not the best way of performing perfectly a unique task, but are well suited to achieve goals needing generality and flexibility in the shape ([24]).

Existing lattice-type modular robots are for example telecube [6], molecubes [2], and roombots, which we will present in more details in section 1.2.

1.1.3 Control

For reconfiguration, modular robots can be controlled in two ways: centralized or decentralized.

In the centralized way, an external controller sends orders to each module to tell them when and how to move. This remote centralized controller is usually not constrained in memory nor computational power. It knows the position of every module and can choose movements that avoid collisions. This is therefore the easiest way of controlling modular robots. However, this is not scalable (the more modules, the more time needed for computation) and we loose some autonomy of the robot as each module of the structure needs to be able to communicate with the controller. Either we use a wired connection and the structure must stay in the vicinity of the controller or bring it when it moves or we use a wireless communication, which has a more limited transmission rate and transmission range. Moreover, this approach is not very robust as, if the controller fails, the whole structure fails. A centralized control can be transported by the robot in a heterogeneous modular structure, but we loose the homogeneity, e.g. all modules are not the same any more and we still have robustness problems.

The decentralized way is when each module has an inside controller and act according to its rules and the information it gets from its sensors and the adjacent modules. This is totally scalable and more in the spirit of modular robots. However, it is very difficult to design the rules to achieve a global goal, because each module has only information from its sensors and by communicating with its neighbors. It has therefore only a local knowledge of the world. Moreover, all modules are interchangeable, and there is no master that decides which module goes where during reconfiguration. All processes must be done through self-organisation. It is really difficult to find the rules to give to each modules so that each of them acts locally to achieve one global goal. Moreover, these rules should be understandable for a human programmer because we want to be able to quickly design a new set of rules to use the robot in a different manner. So decentralized control of modular robots is still a broad research subject. Some

people try to use cellular automata, adapted to self organisation. See [18] for more details.

For locomotion, the control can be different and must adapt to the configuration of the robot. The problems of centralized or decentralized control remain.

1.2 Roombots presentation

Roombots ([4]) are designed in the BIOROB laboratory at EPFL since 2008. They are designed to build adaptive furniture that could achieve locomotion and transform to answer the needs of users (Fig. 4). A large variety of other usages could of course be found for this kind of modules, but their name comes from the fact that they are designed to be integrated in a domestic environment.

- Each module is composed of two parts, each part having the shape of the intersection between a cube of size 11 cm and a sphere of size 12.8 cm. The module is compact and solid. See Fig. 3
- There are ten faces and one connector on each face. A connector is composed of four fingers. They are retractable so that the connectors don't add sharp pieces and the roombots provides a smooth surface.
- Each degree of freedom is controlled in position for a good precision (control in force or speed are subject to drift). Each servomotor in a part (servomotor S) can do a 360 ° turn in 3 seconds; each central servomotor (servomotor M) can do a 360 ° turn in 2 seconds. The torque of the motors are enough to lift the module itself and a second module. Therefore, the servo are powerful enough to move a meta-module in all possible positions.

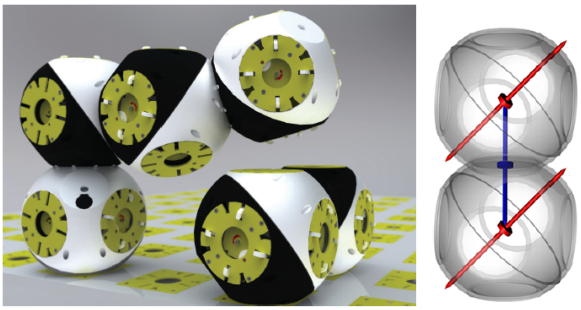

Figure 3: The roombots modules have soft angles, three degrees of freedom and ten connectors.

1.3 Previous work on Roombots control

This project has been done by a post-doc, Masoud Asadpour, two PhD students, Alexander Spröwitch and Rico Möckel. Many master students have also been involved. Lots of simulations have been done using Webots and the hardware system is about to be functional (Fig.5). We often work not with one module but with one meta-module, which consists in two Roombots modules connected

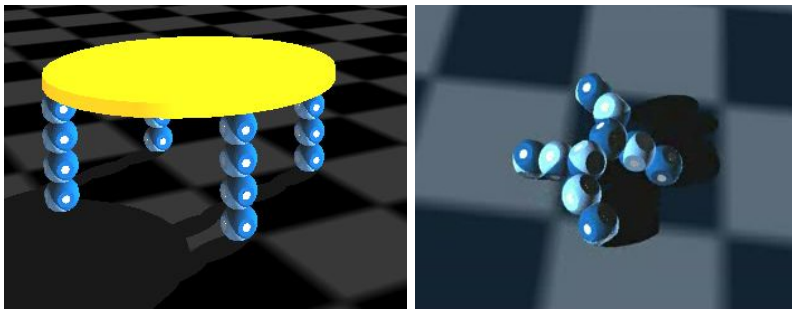

Figure 4: The roombots are designed to build adaptive furniture but can also build other structures, like walking quadrupeds ([22]).

in series. We can therefore achieve six degrees of freedom, which theoretically allows us to reach any position with any orientation within the reachable space of the meta- module.

Figure 5: The real hardware of roombots.

The first part of my work is mainly based on the achievements of Mikaël Mayer ([11]). He implemented a forward kinematics and inverse kinematics solver based on KDL. The Kinematics and Dynamics Library ([16]) develops an application independent framework for modelling and computation of kinematic chains, such as robots, biomechanical human models, etc. It provides class libraries for geometrical objects (point, frame, line,...), kinematic chains of various families (serial, humanoid, parallel, mobile,...), and their motion specification and interpolation.

Mikaël Mayer designed a few instructions allowing to command a Roombots meta module in the joint angle space or in the real space (see sect. 2.1).

The second part of my work is based on Philippe Laprade's collision cloud computation ([8]).

When a module wants to move from one position to another one, it has to make sure that there is no obstacle on its trajectory. If there is any, it is not allowed to move. It has a local knowledge of the environment, i.e., in the 3d grid of size 11cm, it knows which cells are free. Philippe Laprade computed a database of the cells that are collided by the Roombots for any shape to shape transitions (see sect. 3.2). This database is stored in an external supervisor that can communicate with each meta-module. With the collision cloud and the

local knowledge of the environment, the meta-module can know if it is allowed to move. This is a decentralized and therefore scalable method for collision free movements.

1.4 Project's objectives

The title of the project is: **Roombots Reconfiguration-Inverse Kinematics and Cooperative Movement Tasks.**. It was initially described by the four following tasks:

- **Minimization of the collision cloud, optimization of the collision space**: Currently we "sweep" from the initial meta-module shape into the final shape, and simply count and record the amount of cubes in space we collide with. More desired is a movement that minimizes the amount of collided cubes, i.e decreases the collision cloud.
- **Optimize approaching movements**: Another, closely-related point are the transfer movements of meta-modules, from initial to final shape. The transfer movement is usually done to connect to another meta-module, object or to the ground. At the end of the movement, the "head" segment of a meta-module should approach the desired connector in an almost parallel fashion less than in a rotatory movement. This should prohibit collisions. Hence some of the existing, recorded movements are more favourable than others. It is of interest to exactly formulate the requirements, and eventually alter the transfer-movement framework accordingly.
- **Handling of passive objects 1**: We want to extend the Roombots reconfiguration framework by passive mechanical elements, for the beginning with thin, light-weight plates that will act as a seating or a table area. First task would be the transport of such a plate via a number of meta-modules from one point to another. Meta-modules should take up a plate, rotate it towards the next meta-module, and reach it over.
- **Handling of passive objects 2**: In case larger plates need to be handled, it will be necessary to use two meta-modules to synchronously lift up the plate. This will create a closed-kinematic chain of 12dof, with the constrain for at least two connector segments given by the plate. Check for existing strategies to solve the inverse kinematics for this task, and implement the most feasible one. One can integrate passive or compliant joints in the kinematic loop to reduce some constrains.

We started by the second point, presented in details the chapter 2 and used it to solve a particular case of the third point: handling light passive plates. Then we worked on the first point, developed in chapters 3 and 4. After some results, a choice has been made to work on this point in more details and therefore to abandon the fourth point, because of time constraints.

1.5 Project's timeline

This semester project has to be realized in 14 weeks between the end of February and the beginning of June 2010. You can see in picture 6 the time line of the main sub tasks.

Figure 6: Gantt chart of the project. The red line stands for the intermediate presentation.

2 Optimized Approaching Movements

In this chapter, we want to improve the movements of a meta-module when it approaches of a connector on the floor or on another Roombots module. We want to avoid collisions with this goal connector and allow an easy connection. To do this, we first introduce the way the Roombots are controlled in the previous work section (sect. 2.1), then we define more precisely our objectives and constraints (sect. 2.2) and introduce the *virtual connector* (sect. 2.3). After this, we define the *docking angle* measure and discuss the results (sect. 2.4) and finally, we apply the virtual connector method to passive plate handling (sect. 2.5).

2.1 Previous work

2.1.1 Roombots control in Webots

The control of the Roombots in Webots simulations ([20]) was mostly implemented by Mikaël Mayer ([11]) and Philippe Laprade ([8]). It is shared between two entities (see Fig. 7). The first one is named *controller*, there is one for each module (all controllers are identical). The controller has access to the servos to make the module move and reads the position sensors of its robot. It communicates with dedicated messages (see [11]) with an external controller named *supervisor*. The supervisor reads a series of instructions (detailed in the next section) from a text file, interprets them and sends orders to the different *controllers*, containing angular positions to reach or requesting actual angular positions. There is one *supervisor* for each meta-module (all supervisors are identical but the instruction file can change). As the *controller* has only very local and limited knowledge of the world, the supervisor has a much more complete knowledge of the environment.

2.1.2 Shapes and configurations

The different instructions designed by Mikaël Mayer allow us to command the Roombots in 3D space or in joint angle space. To achieve this, he designed a whole range of referentials to describe exactly the position and orientation of every piece of the Roombots. He also numerated the connectors and described the different connection types. Indeed, there are four possible ways of connecting two modules to build a meta modules. As the Roombots architecture is not invariant for a 90 ° rotation, the different connection types lead to different reachable spaces. The connection type can be PAR, SRZ, SRS or PER, as described in Fig. 8.

Most of the time, we like the roombots to be in a grid. We call these special positions shapes. They are realized when all joint angles inside the sphere-like shapes (the s servos) are multiples of $\frac{2.\pi}{3}$ and the joint angles between two sphere-like shapes (the m servos) are multiples of $\frac{\pi}{2}$ (when all joint angles are 0, the meta-module is in I shape). There are 5 basic shapes, as presented in Fig. 9, which can be rotated. When a meta-module builds a shape, its s servos can take 3 different positions and its m servos 4 (see Fig.10). As there are 4 s servos and 2 m servos, there are $3^4.4^2 = 1296$ points in joint angles space that correspond to shape positions.

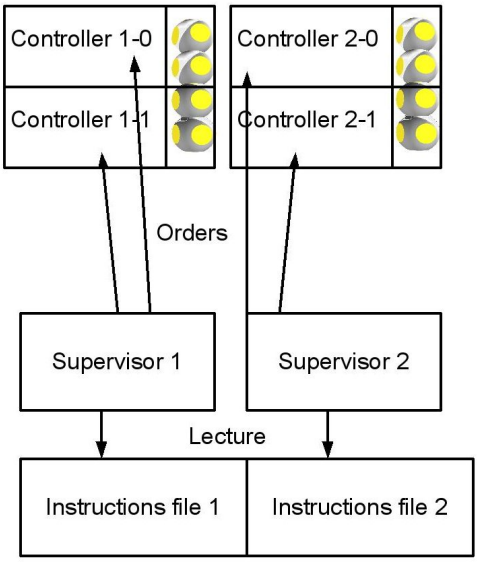

Figure 7: There is one supervisor for each meta-module. The supervisors are the same but read different instructions files.

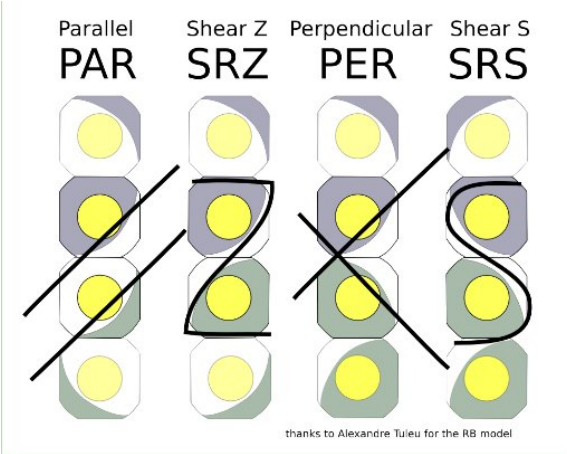

Figure 8: The different connection types of the roombots. If the two middle circles are parallel, we are in a PAR configuration, if they joint in a point, it is a PER configuration. Otherwise, depending on the direction, we have a SRS or SRZ configuration. (picture from [11])

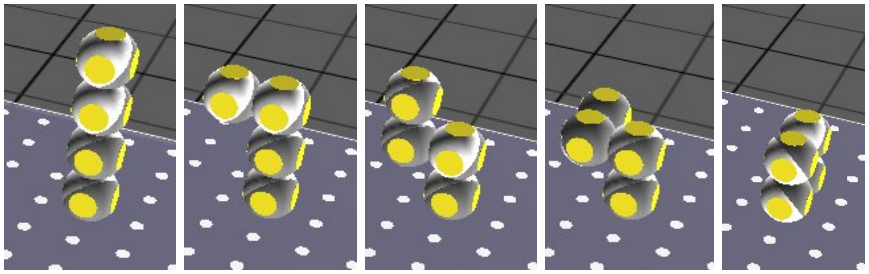

Figure 9: We mostly work with predefined shape positions (from left to right: I, L, S, 3dS, U).

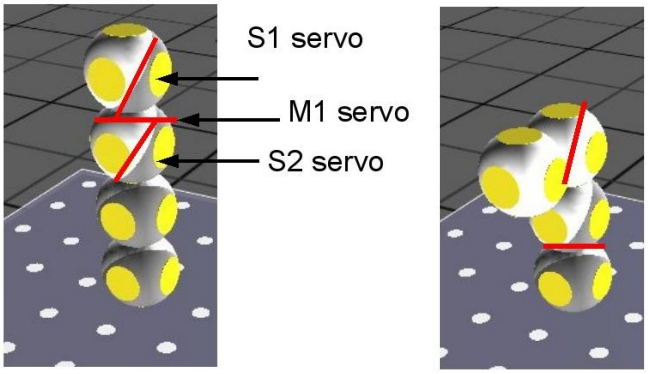

Figure 10: Right: the servomotors of a module are called S1, M1 and S2. Left: after the instruction **move 0 1 0 1 0 0**: the servo M1 of the first module moved of $\frac{\pi}{2}$rad and the servo S2 of the second modules moved of $\frac{2.\pi}{3}$rad.

2.1.3 Description of the instructions

The Roombots instructions are of four types:

- the instruction **step arg0** makes the robot stay still for a time $arg0 * basic\_time\_step$. The basic time step is currently 16 ms.
- the instruction **lock arg0 arg1** or **unlock arg0 arg1**activates the connector described by the argument arg1 of the module arg0. arg0 can be either 0 or 1 and indicate on which module of the meta-module we work. arg1 can take the values (C0X - C0Y - C0Z - C1Y - C1Z - C2Y - C2Z - C3X - C3Y - C3Z). These are the names of the different connectors: the number is the submodule and the last letter is the orientation when all servos angles are 0. We mainly use C0X of module 0 and C3X of module 1, which are the two end connectors.
- The instruction **move arg0 arg1 arg2 arg3 arg4 arg5** makes the robot move to reach the joint angles given by arg0 to arg5. This function is designed for shape to shape transitions so the arguments of the functions are not directly the angles. For the s servos (angles inside the spheres, arguments 0, 2, 3 and 5) the argument is the angle divided by $\frac{2.\pi}{3}$. The usual values are 0, 1 or -1 (or 2 which is equivalent to -1). For the m servos (angles between the spheres, arguments 1 and 4), the argument is the angle divided by $\frac{\pi}{2}$. Usual values are between - 2 and 3 (3 and -1 being equivalent, as 2 and -2). Non-integer values can of course be used to reach a non-shape position. See Fig.10
- The instruction **reach arg0 arg1 arg2 arg3 arg4 arg5 arg6** defines the position and orientation to reach for the end connector (see Fig. 11).

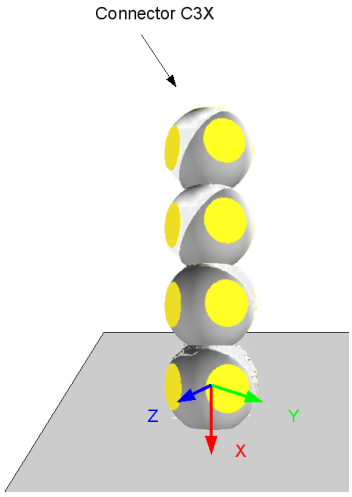

Figure 11: The M0 referential was designed with respect the the first submodule, so its orientation is not straightforward for our use. The C3X connector will be called end connector from now on.

The **reach** instruction creates a Frame object containing the position and orientation we want to reach. Then it computes the angles to reach using an inverse kinematics solver, based on the Newton-Raphson algorithm ([7]), which uses an incremental method. Sometimes, the solver fails, because the goal is not reachable or the algorithm is stuck in a local minimum.

- We now work in an external referential based on the first submodule of the first module. The environment is divided in a 3d grid of the size of the Roombots (cube size = 11cm). Each cube is designed by the coordinates of its center divided by the cube size: cube (0,0,0) is around the first submodule, cube (1, 0, 0) is its neighbor in the X direction. **arg0 to arg2** point out one of the cubes surrounding the first submodule, where we want the last submodule to arrive.
- **arg3 to 5** can take the values 0, 1 or -1. One of them and only one is non-zero. They represent one of the sides of the cube, the one we want to reach with the C3X connector (goal connector).
- **arg6** represents the rotation angle around the normal vector outside the side of the cube, divided by $\frac{\pi}{2}$ to have integers for shapes.

Non-integer values can be used for all arguments. They will lead to an end position not in the grid for arg0 to arg2, and a non conventional orientation for arg3 to arg 6.

The instructions **movebw** and **reachbw** are the same that **move** and **reach** and take the same arguments but suppose that the fixed connector is C3X instead of C0X. Therefore the arguments of **movebw** are inverted: **arg0** stands for the S1 servo of the second module and not of the first. The referential used in textbfreachbw is M3 (referential of the last submodule) instead of M0.

Indeed, when a meta module has to move on a long distance (more than its own size), the meta-module, locked with its C0X connector on the floor or on another Roombots, reaches with its end connector (C3X) the floor or another modules. Then it unlocks C0X and moves again, to reach another goal connector with its C0X connector. So when the C3X connector is locked, the instructions **movebw** and **reachbw** are easier to use than **move** and **reach**.

2.2 Objectives and constraints

When the *supervisor* reads a **move** or **reach** instruction, it computes the joint angles and sends them to the *controller*. But what if directly reaching the goal joint angles is not the best solution ? Indeed, even without considering collisions during the movement, when approaching the floor or another module to lock on it, one must be particularly cautious to avoid collisions and allow a good connection (see Fig. 12). Therefore, we would like the end connector (the one on the moving meta-module) to approach as much as possible with a parallel motion with respect to the normal axis of the goal connector (on the ground or on another module).

We will change the reach and move instructions to allow a better approach, but we want to preserve the final shapes of the meta-module and to induce only few changes in the trajectories. This will allow to simply add the amelioration

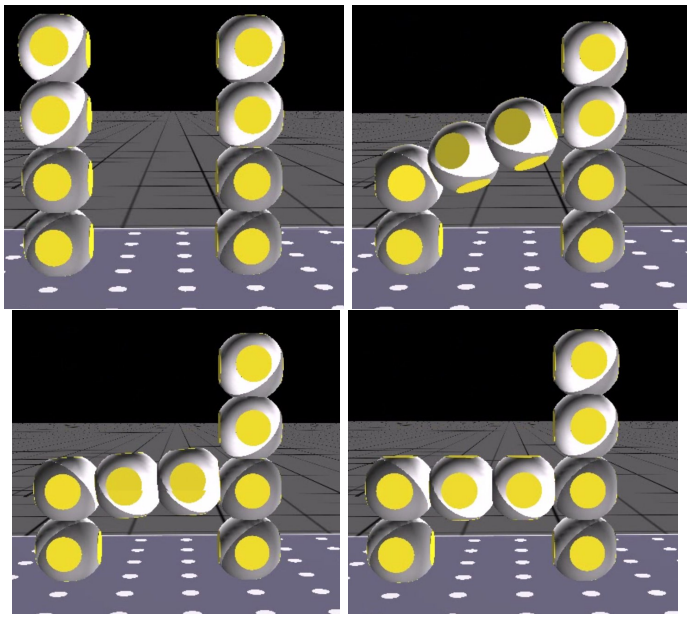

Figure 12: The modules reach the goal connector directly. The collision risk is high.

and use the sequences previously defined (as Mikaël Mayer's chair building [12]) without having to re-check every move to avoid collisions.

To achieve this goal, we defined and implemented a *virtual connector*. Then we quantify the amelioration it brings with the *docking angle* measure.

2.3 The virtual connector: definition and implementation.

2.3.1 Definition

An idea to achieve a good connector approaching movement is to force the connector to reach first a virtual connector (VC), which is a position at a certain distance from the goal connector, with the same orientation. The end connector of the meta module has therefore to reach the right orientation before the right position, but at a small distance of it. Then, we have to assume that, to move on the small distance between the virtual and the real connector, the angles move only little. Therefore, we made the assumption that the end connector also moves little and keeps an orientation close to the final one while it reaches the goal connector and lock on it safely.

This assumption of a small angle change for a small position change doesn't hold all the time, especially when we are close to the boundaries of the reachable space of the meta-module. But in these cases, the module has only few possibilities to approach the connector and we saw that it is less likely to arrive in a strange way.

To add this virtual connector, we had to modify the instructions. In fact, we added the instructions **moveVC**, **movebwVC**, **reachVC** and **reachbwVC** to the list of possible instructions. To calculate the angles needed for the virtual

connector, we have to use inverse kinematics (IK).

2.3.2 Implementation: first approach

For a **reach** instruction, we already have the Frame object describing the position and orientation of the end connector (using a rotation matrix, in the referential of the first submodule). We can simply add an offset of length VIRTUAL_CONNECTOR to the position to reach (after some trials, we decided to use VIRTUAL_CONNECTOR = 1cm), in the direction of the C3X connector (see Fig. 13).

Indeed, we assume, as Philippe Laprade and others did before us, that we will lock using the C3X connector, that we call the end connector. Reaching the right orientation for this connector needs only the two last servos (M1 and S1 of the second module). If we did not suppose this, we would have to add at least one argument to the instructions to specify which connector we will lock on.

We send orders to the concerned controllers and, once they answered to acknowledge that the position has been reached, we send another message to reach the goal connector, using the previous reach instruction.

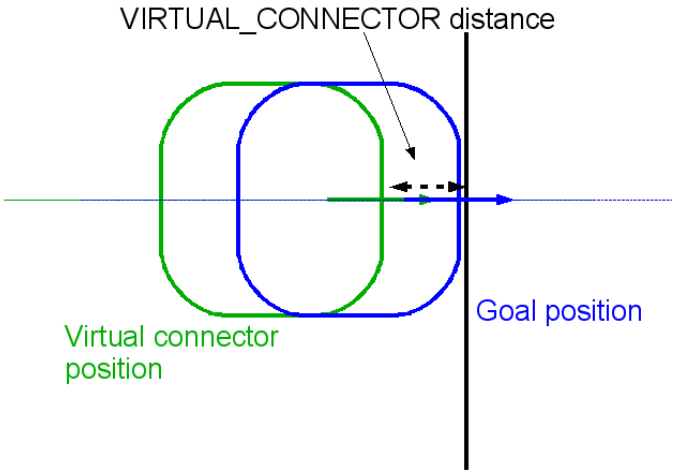

Figure 13: The virtual connector is a point in front of the real connector with an distance of length VIRTUAL_CONNECTOR.

For a **move** instruction, we first use the forward kinematics solver (function *Jnt2Cart* in the code) to get the position and orientation, i.e. the Frame of the final connector, which we modify in the same way as for the reach instruction. Then, again, we solve IK to get the joint angles corresponding to the virtual connector. At the end, we ask the modules to move to these joint angles and then to the real one, using the previous move instruction.

2.3.3 Implementation: 3-steps approach

However, the first approach has a drawback. We want to preserve the shape of the meta-module the same as if we didn't use the virtual connector. For the **reach** instruction, we observe that it is not always the case. There can be many ways to reach a given point and the IK solution to reach the virtual connector can be very different to the one found without VC, even if we can easily reach the final position from the VC one. For the **move** instruction, the final position is always the same as it is entirely defined by the joint angles, but we want to avoid joint angles at the virtual connector being very different from the final angles to avoid big movements when going from the VC to the final position (this would totally take away all improvement of the virtual connector) and to conserve a collided space close to the one without VC.

To avoid this, we add a first stage before reaching to virtual connector: going near the final point in the same way as if we didn't use the VC. We can do this because of the incremental way used to calculate the inverse kinematics: it updates the angles using a gradient descent method. If we start the process near to a minimum, we know it will find that minimum. If we start far away, we cannot know before computation in which minimum it will end. We therefore want to first reach a position with joint angles close to the desired ones. Then the IK algorithm will find a solution close (in the angular space) of this one for the virtual connector.

For a move instruction: we move to the joint angles defined by $initial\_angle + 0.9 * (final\_angle - initial\_angle)$ before starting the process to find the joint angles corresponding to the virtual connector.

For a reach instruction, we again use the way the IK is computed. The algorithm stops when it reaches a end position close enough to the goal one. The threshold defining *close enough* is fixed to 5mm by default. If we change this threshold to some centimeters for example, the algorithm will stop sooner, but having followed the same steps as when we calculate the angles to reach the goal connector. Therefore, the angles will be close to the ones without VC but the end connector won't directly reach the goal connector (see Fig. 14).

To summarize:

steps	**move**	**reach**
approach	$initial + 0.9 * (final - initial)$	IK imprecise
reach VC	FK → modify frame → IK	modify frame → IK
reach goal	final_angle	IK precise

2.4 Measure and results

To quantify the improvement added by the virtual connector, we need a measure of *how well we approach a goal connector*. We define the *docking angle*. This measure is based on the angle between the normal vector of the end connector and the normal vector of the goal connector, as a lateral displacement between two submodules with a right orientation is, maybe not optimal, but less harmful than a normal approach with a bad orientation.

The docking angle measure is defined as the maximal angle between the normal vector of the end connector and the normal angle of the goal connector when the end connector is close enough to his goal (see Fig. 15). After some

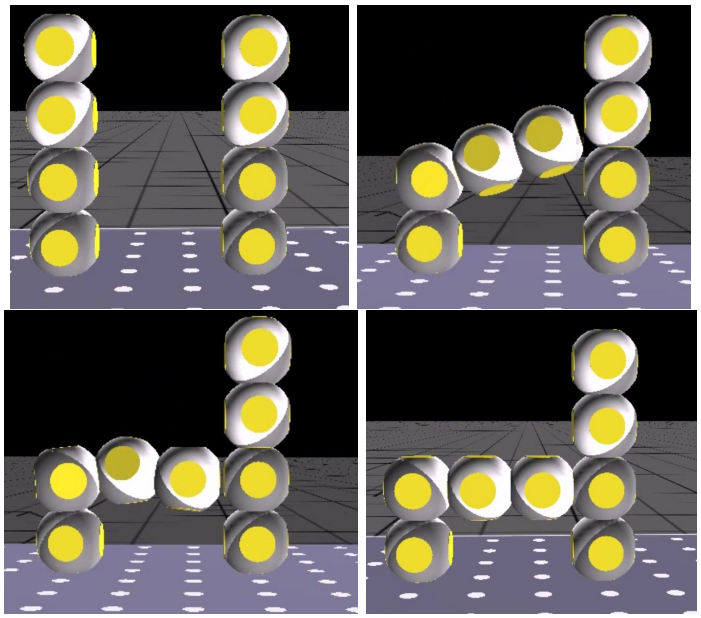

Figure 14: The modules reach the goal connector passing through a virtual connector: in the second image, we are in the *approach* phase, in the third we are in the virtual connector position.

trials, *Close enough* was set to 9mm. The virtual connector is therefore outside of the measured space and can have a visible effect on the measure.

The docking angle is calculated by the supervisor, because none of the controllers has enough global knowledge to know the shape of an entire meta-module. But the supervisor doesn't have access to the values of the joint angles at each time step, because it get this information only when it asks it to the module and getting it at each time step would slow down the all process. Therefore, instead of using the sensed angles, we simulate the whole moving process into the supervisor. We use the formula implemented in webots ([21]):

```
Vc = P * (goal_angle - current_angle);
if (abs(Vc) > MAX_VELOCITY)
  Vc = sign(Vc) * MAX_VELOCITY;
if (A != -1) {
  a = (Vc - MAX_VELOCITY) / DELTA_T;
  if (abs(a) > A)
    a = sign(a) * A;
  Vc = MAX_VELOCITY + a * DELTA_T;
}
```

This will have to be changed for the real robot, using a more realistic regulator, designed through tests and regression. To confirm our results, we used a controller to store the actual angles at each time step in a file and compared the simulated angles (in matlab) and the recorded angles. The difference was never greater than $3.10^{-3} rad$, i.e. $0.17°$.

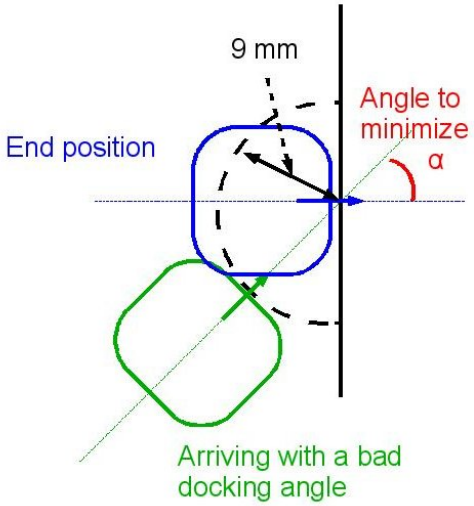

Figure 15: The docking angle measure how well we approach the connector.

	without VC	**with VC**	**with VC if** $DA \geq 2.9\,^\circ$
average docking angle and standard deviation	$4.0\,^\circ \pm 2.5\,^\circ$	$2.4\,^\circ \pm 4.1\,^\circ$	$1.3\,^\circ \pm 1.3\,^\circ$

Table 1: Docking angle results with and without virtual connector.

To test the efficiency of the virtual connector method, we chose 40 shape to shape transitions and computed their docking angles with and without the virtual connector. We can see in the table 2.4 that using the virtual connector decreases the average docking angle but increases the standard deviation.

Indeed, for certain positions, reaching the virtual connector may be quite difficult or impossible (for example a I shape with the end connector on the side: the I shape is the only position where the end submodule is at maximal height. It cannot reach a virtual connector on a side at the same height). In these cases, we could observe that the direct move was quite good and using the virtual connector was catastrophic. Therefore, we decided to first compute the docking angle without the virtual connector. If it was below a threshold (that we put at 0.05 rad, i.e. $2.9\,^\circ$ after looking at some simulation's results), we go directly, otherwise, we use the virtual connector. This gives us the last number of the table 2.4, where the average docking angle decreases even more and the standard deviation is more acceptable.

2.5 Application of the virtual connector to light-weight plate handling

To build a chair or a table, we won't use only roombots modules, but we want them to manipulate passive objects, in particular plates, to form the furniture.

The objects have connectors on them. Here we consider a plate, light enough to be manipulated by a single meta-module. The plate is 0.22 m long and large and 1.6 mm height. The corners have been rounded to avoid some collisions during the movement and sharp angles in the final structure.

This object is easy to manipulate using the virtual connector method: instead of reaching the virtual connector and then the goal one, the connector on the plate is at the position of the virtual connector. We can also quite easily put another virtual connector above the plate connector to assure a good approach. This is particularly important as the plate is not locked on the floor and can easily slip if the meta-module touches it in a wrong way. As we do not have any sensing of the external world in a roombots module, if the plate moves, even a bit, we have no way of knowing it and finding it back.

The implementation is as follow: we have an initial Frame object, given by the desired position in case of a **reach** instruction or by the FK solver for a **move** instruction. We create a Frame frameVC, with an offset of VIRTUAL_CONNECTOR + PLATE_HEIGHT and a Frame frameGoal with an offset PLATE_HEIGHT compared to the initial Frame. The three steps are the following:

- approach: depending on the instruction used
- reach frameVC
- reach frameGoal

The roombots can manipulate plates to build structures but also simply pass the object from one to another to carry the object on big distances. To avoid closing the kinematic loop, a module takes the plate at one place and put it down at another place near a meta-module that will execute exactly the same movement. The putting down of the plate must be done very carefully to ensure that the position of the plate is correct for the second module. Therefore, we also use a virtual connector to align the plate with the ground before putting it down. It also allows to avoid collisions, as the plate easily collides with the ground if we are not careful to have it parallel to the floor while approaching it.

Results can be seen in the Fig. 16 or the video can be found on the biorob website [14].

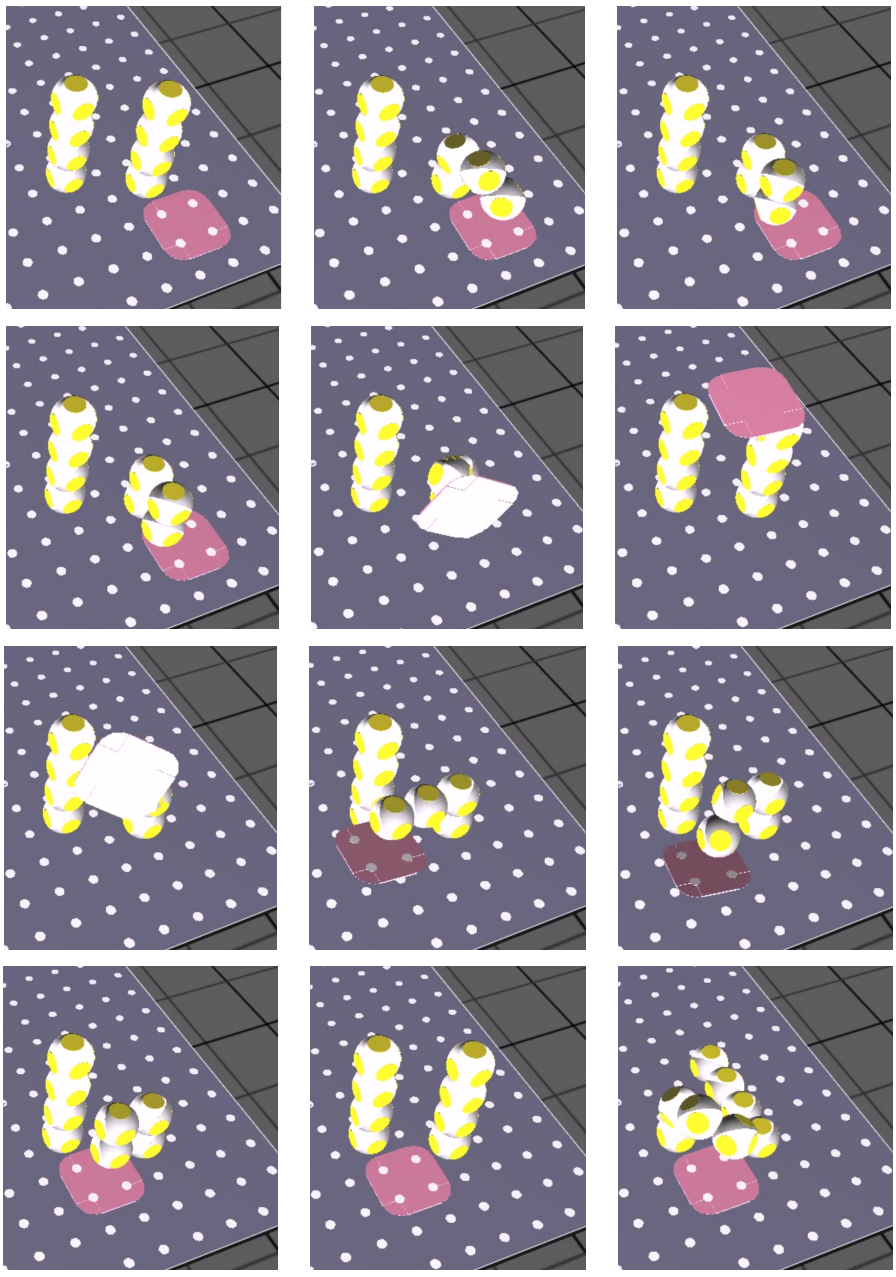

Figure 16: A meta-module lifts a light plate and puts it down in the position for the second meta-module to handle it.

3 Collision clouds: definition and computation

In this chapter, we introduce the collision clouds, used to avoid collisions with few online computation. After precising our objectives (sect. 3.1), we develop what has been done as previous work (sect. 3.2). As the methods we will use are computationally intensive (sect. 3.3), we first implement different strategies to reduce the computation time (sect. 3.4 and 3.5).

3.1 Objectives and constraints

The second part of my work is to optimize the collision cloud.

When a module wants to move, it has to be sure it won't collide with its environment. As they do not sense the environment, we need to make sure that the Roombots can avoid the obstacles on its way.

The first strategy implemented by Mikaël Mayer ([11]) was to recognize a collision when to module didn't move (he used the internal position sensors) and was not at its goal position. The module then came back to its original position and tried another movement.

But this is not a very efficient strategy, as we have to collide to detect an obstacle. We can do far better if the module has some information about its environment. A meta-module can't reach a cube that is more than 3 cubes away from its first submodule, which is the origin of the referential. Therefore the reachable space is included into a 7*7*7 grid. The meta-modules has local knowledge of its environment, i.e. it knows which of the cubes of the 7*7*7 grid are occupied.

But the modules are embedded devices so we have constraints in term of computational power and memory size. We use another possibility of the Roombots, the ability to communicate with an external supervisor. This supervisor is a computer, it has therefore a huge amount of available memory. It has also computational power but, as we want our system to be very scalable (adding modules does not modify the overall behaviour of the system), we would prefer to avoid computing too much things in the supervisor. Indeed, the less computation in the supervisor, the smaller the response time and the more module can be handled on the same computer without worrying about time-lag or task paralleling.

Philippe Laprade ([8]) used a database of collision clouds that allows meta-modules to detect collision with a minimum amount of computation.

3.2 Previous work

3.2.1 Definition of the collision cloud

We consider a 3D grid on the world, centered on the first submodule of the meta module. Each cube of the grid have a side of 0.11m, i.e. the meta-module in I shape (the position 0 for each servo) is exactly contained in 4 cubes. The collision cloud is a list of cubes that the meta-module collides when moving from one position to another one. The idea is to precompute the collisions clouds for a set of predefined movements (see Fig. 17).

These movements are the shape to shape transitions, which are the most used and allow us to have initial and final positions exactly on the grid. As we

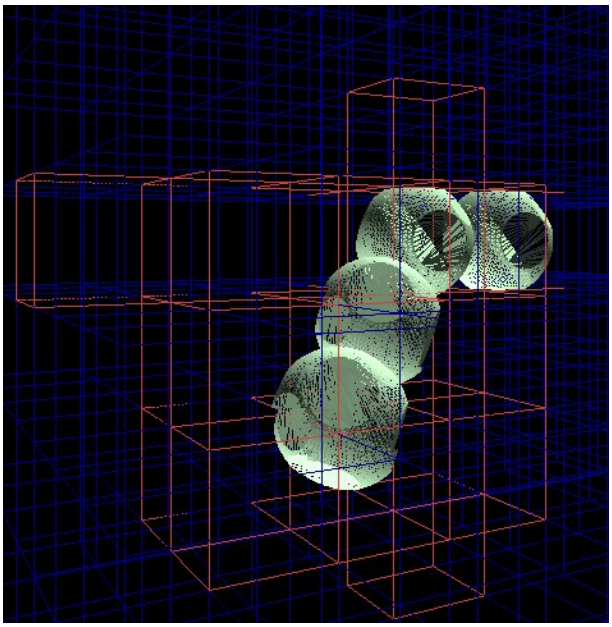

Figure 17: The collision cloud of a simple movement.

already showed in the section 2.1.2, the meta-module is in a shape positions if the servos 0, 2, 3 and 5 are in a position multiple of 120 ° ($\frac{2\pi}{3} rad$, 3 possible positions) and the servos 1 and 4 are in a position multiple of 90 ° ($\frac{\pi}{2} rad$, 4 possible positions). Those positions also allow to connect easily one module to another one using Mikaël Mayer's framework.

Therefore, before starting a move, the meta-module can ask an external supervisor for the collision cloud associated with the move and compare it with the grid of the environment. If the cloud doesn't contain any cube occupied by an external obstacle (Roombots, ground, passive object...), then the move is collision free and the meta-module does it. Otherwise, the move is not collision free and the module cancels its move or asks for another cloud.

With this method, once the database of collision clouds is done, we need only one communication and a little computation done by the module to determine if a move is collision free or not. The method is totally scalable as long as the supervisor is not saturated with communications.

3.2.2 Computation of a collision cloud

Philippe Laprade's implementation checks the position of each point of a mesh for each submodule and add the associated cube to the list of collided cubes. The mesh is composed of approximately 1000 points and there are 7 submodules to test (the first one, called the root submodule, is supposed to be static in the origin cube, so there is no need to compute it).

As we want the cloud not for a given position but for a movement between two positions,the algorithm simulates the movement by changing the angles of a certain amount of rotation and computes the collisions for each intermediate position.

The result is therefore a very precise collision cloud, in the sense that we have very few true negative, i.e. cubes that are actually collided but not added to the collision cloud and no false positive, i.e. cube that is recorded as collided but is actually not. However, a cube that is simply brushed against is added to the cloud as well as a cube that is crossed by the middle. Therefore, even if we are precise in term of collided cubes, we may be quite imprecise in term of collided space (see Fig.18). This can be improved by reducing the size of the grid cube. This can be done easily with all our algorithm, but the computational load and memory size must be studied before applying such changes.

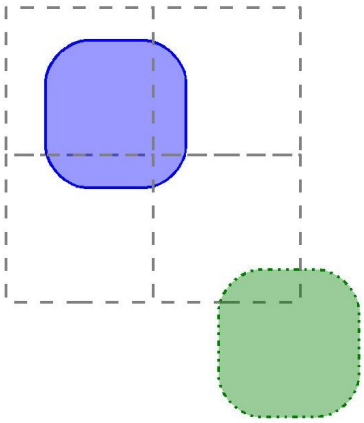

Figure 18: All four cubes are collided by the blue (upper left) Roombots element, therefore a collision is detected between the blue and the green (down right) Roombots elements.

Finally, this method is pretty computationally intensive.

3.2.3 Avoiding obstacles with an intermediate move

To improve his method in case of obstacles detected on the direct movement between an initial and a final configuration, Philippe Laprade implemented a level 2 (and even a level 3) search. The level 2 search consists in looking for a collision free movement to an intermediate position that gives a collision free movement to the final position. If one is found, the meta-module moves, connect in the intermediate position, disconnect from its previous position and moves to the final one. In the level 3 search, there are two intermediate positions, therefore many more move possibilities.

3.3 Summary of the different strategies.

Our goal is to reduce the size of the collision clouds of the database to help avoiding collisions. Indeed, if a cloud is small, the probability that it collides with an external element is reduced. We will also add some constraints to the clouds, as for example a good docking angle.

For the moment, we suppose that all servomotors start moving at the same time and move at the same speed, each of them stopping when the desired

joint angle is achieved. Therefore, we move directly from the initial to the final position. If we change the starting time or the speed of the servomotors, the meta-module doesn't go through the same positions, the collision cloud is therefore different. Thus, we want to optimize the servomotors' movements timing to decrease the collision cloud size.

The basic idea was to *add intermediate positions* to the movement when computing the cloud. This is different from what Philippe Laprade does to avoid obstacles: he looks online for positions where the meta-module can lock onto, as we look offline to a smaller cloud, conserving the initial and final positions. Philippe Laprade's method can still be used with our optimized clouds.

We used several approach to decrease the collisions cloud's sizes.

- The first approach was **brute force**: we computed the clouds with every possible intermediate positions and remembered the smallest one.
- Then we tried an **iterative approach**, starting with some intermediate position and modifying it gradually to decrease the cloud size.

These methods will be described in more details in the next chapter, as well as the **milestones method**, that allow us to find dynamically a set of intermediate positions to avoid an obstacle.

These method are quite computationally intensive, especially if we want a good sampling of the space (lots of possible intermediate positions) and several intermediate positions (for each intermediate position, we have six joint angles to optimize). Therefore we also tried to compute the collision cloud in a quicker way. We **modeled** each submodule as a half-sphere intersected by a half-cube and used geometric properties to check collisions instead of using a mesh (sect. 3.4). We also looked for other indicators to avoid the whole cloud computing and ended with the **length of the end effector trajectory** method (sect. 3.5).

3.4 The {sphere + cube} model.

3.4.1 Method presentation

A Roombots meta-module is composed of 8 submodules, each submodule's shape being the intersection between a half sphere and a half cube. The submodules go two by two. Two submodules form what we will call an element (see Appendix A). In the I shape when all joint angles are equal to zero, an element has exactly the shape of a sphere intersected with a cube. If the servomotor in the middle of the element (one of the 's' servos) moves, the shape of the element is a sphere intersected with two half cubes.

We decided to use this modeling to compute the collision cloud and take advantage of some geometric properties to avoid using the 1000 points mesh and reduce computation time. This is useful as all method we propose to optimize the collision clouds ask for a far greater amount of cloud computation that Philippe Laprade's method.

In the C++ code, this function is the *getCollisions* function, called by the *simulate* function. Philippe Laprade's function has been conserved and called *getCollisions_old*.

To avoid confusion between the cube defining an element and the cubes that compose the environment (and the collision cloud), we will call the latter "box"

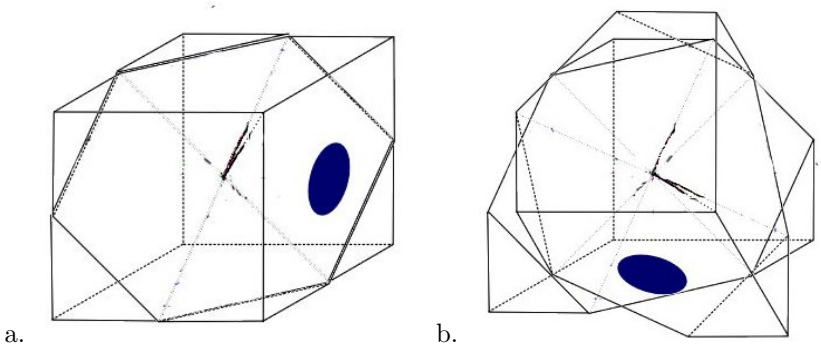

Figure 19: Two half cubes in the zero position (a) and rotated (b). Images from [11]

in the section (see Appendix B). We also call BOX_SIZE the length of the side of a box, which is currently 11 cm, but could be reduced for a better precision. We recall that SPHERE_SIZE, the diameter of the Roombots element, is 12.8 cm and CUBE_SIZE, the side of the cube defining an element, is 11cm. At each time step, we compute the forward kinematics to get the position of the centers of the elements and the orientations of the submodules.

Then we compute the following points :

- check if we are sure to collide (collision with the inscribed sphere)
- check if the box collides with the sphere.
- check if the box collides with the cube.

3.4.2 Check collision with the inscribed sphere

The movement is simulated and the position of each center is found by forward kinematics.

Then, for each of the four elements, we consider the boxes whose center are close enough to the center of the element. For each of the boxes, we first check if we are sure that there is a collision. Indeed, if the distance between the center of the box and the center of the element is less than $\frac{BOX\_SIZE}{2} + \frac{CUBE\_SIZE}{2}$ (**condition 1**), the box collides with the inscribed sphere of the element cube and therefore with the cube and the sphere, as illustrated in Fig.20.

If the condition 1 is not verified, we have to check separately if the box collides with the sphere and the cube of the element. We start with the sphere.

3.4.3 Box - sphere collision

Let call D the distance between the center of the box and the center of the sphere and:

$$Dx = |X_{center\_of\_sphere} - X_{center\_of\_box}| \tag{1}$$

$$Dy = |Y_{center\_of\_sphere} - Y_{center\_of\_box}| \tag{2}$$

$$Dz = |Z_{center\_of\_sphere} - Z_{center\_of\_box}| \tag{3}$$

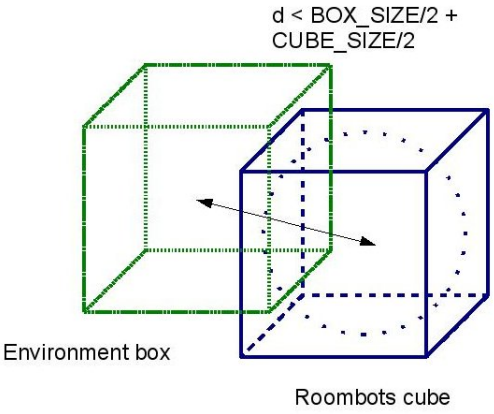

Figure 20: If the box collides with the inscribed sphere, it collides with the element.

There cannot be a collision if the center of the sphere is not in the cube of side $BOX\_SIZE + SPHERE\_SIZE$ centered on the center of the box, as the center of the sphere is at a distance greater than $\frac{SPHERE\_SIZE}{2}$ of any point of the cube. Therefore, if one of the values Dx, Dy, Dz is greater than $\frac{BOX\_SIZE}{2} + \frac{SPHERE\_SIZE}{2}$ (**condition 2**), there is no collision (see Fig.21).

If condition 2 is verified, we have to distinguish between four cases:

- **case1**: if all Dx, Dy and Dz are less than $\frac{BOX\_SIZE}{2}$, the center if the sphere is inside the cube. There is collision but this case should not happen if we don't verify condition 1.
- **case2**: if two of Dx, Dy and Dz are less than $\frac{BOX\_SIZE}{2}$, the sphere collides with a side of the cube.
- **case3**: if only one of Dx, Dy and Dz is less than $\frac{BOX\_SIZE}{2}$, the sphere may collide with an edge of the box. The center of the sphere must therefore be at a distance less than $\frac{SPHERE\_SIZE}{2}$ of the edge to collide with the box.
- **case4**: if none of Dx, Dy and Dz is less than $\frac{BOX\_SIZE}{2}$, the sphere may collide with a corner of the box. The center of the sphere must therefore be at a distance less than $\frac{SPHERE\_SIZE}{2}$ of the corner to collide with the box.

3.4.4 Box - cube collision

If the sphere of the element and the box collide, we have to check if the cube and the box also collide or, more precisely if at least one half cube of the element collides with the box. There is no exact method to check collision between two cubes if they are not axis aligned. But we will use an approximated method consisting in checking if at least one corner of each half cube is inside the box or at least one corner of the box is inside one half cube.

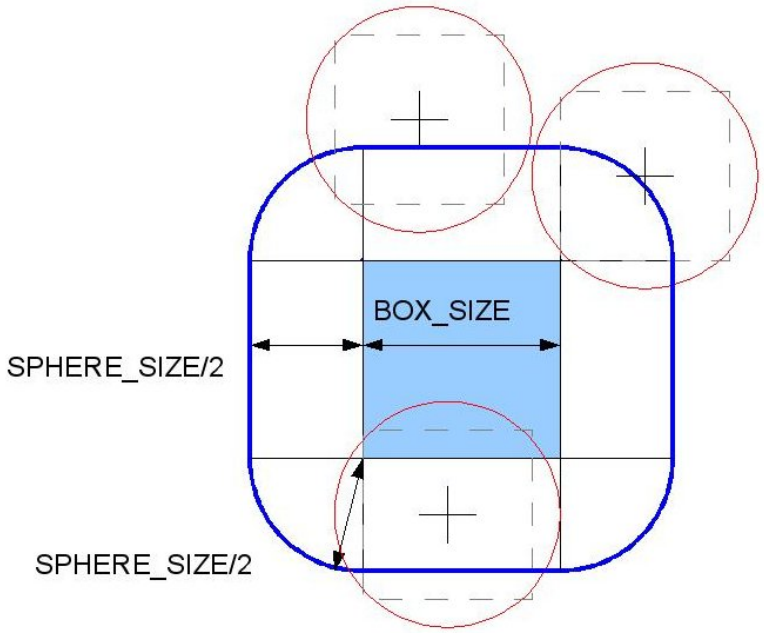

Figure 21: Different cases of sphere-box positions in 2D. If the center of the sphere is inside the blue perimeter, it collides with the box.

Checking if a point is in a cube or a polygon formed by two rotated half (see Fig.19) cubes is relatively easy: we have to check on which side of 6 plans (the sides of the cube) we are. This is done by 6 times a scalar product between the vector defining the point and the normal vector of the plane and a comparison.

This corner-checking method is not exact but proved very good results. We know that we are not colliding with the inscribed sphere so we don't bother of the case when the two cubes have the same center but are rotated one compared to the other. Moreover, if we don't detect a collision at one time step, we may detect it at the next one.

One particular case remains: if the only servomotor moving is a servomotor between two elements, the cube of the element rotates but it corner remains on the boundaries of boxes and the collision may not be detected. That's why we also test points in the middle of the concerned edges of the half cube, as shown in Fig.22.

Then, if the box collides with at least one of the half cubes, it collides with the Roombots and we add it to the collision cloud.

3.4.5 Results and variations

The { sphere + cube} method allows to **speed up the collision cloud computation of a factor almost 25**.

The test was done computing the best cloud using the brute force algorithm with one intermediate position and nAngle = 4 (see next chapter), therefore 512 clouds to compute. The method using the mesh answered in 247 seconds and

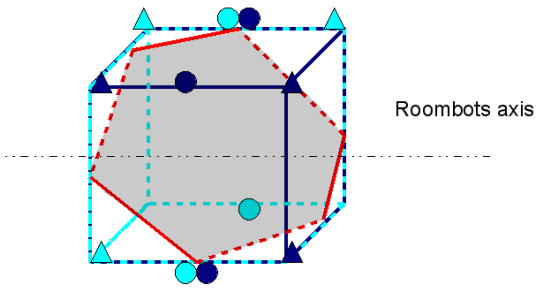

Figure 22: One submodule is dark and the other is light. The separating plan is grey with borders in red. Triangles show the edges of each submodule and circles the points we check in the middle of the edges.

the one with the { sphere + cube} method answered in 10.2 seconds on average, i.e. 0.020s per cloud.

When computing the cloud, we though of replacing the boxes by spheres (see Fig. 23), because the collision checking would be even easier and quicker. But as a meta-module in a shape position is well described by cubes and spheres would collide with each other, this idea was not developed.

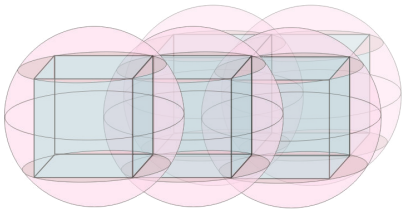

Figure 23: The environment can be filled by spheres or cubes.

3.5 The End Effector Trajectory Length method

3.5.1 Implementation

Checking for collision is the most costly part of the collision cloud computation. As most of the clouds we compute will never be used, we try to find other indicators of how good a trajectory is, without having to compute explicitly the whole cloud.

Our new strategy is now to look at the **end effector trajectory**. The end effector is in fact a point on the C3X connector (the upper connector when the module is in I shape). To record also movements of rotation around the axis of the connector, we chose a point on the side of the connector disk.

Indeed, if this point does a long trajectory, it is likely that the computed movement contains lots of parasite moves and therefore an increased collision cloud.

We tried different measures, as the length of the end effector trajectory or the swept space (sum of the distances between the center of the root element to the end effector for all time steps).

The first proposal showed the better results: on a sample of 100 computed clouds, the correlation between the end effector trajectory length and the cloud size was above 0.97. It means that, if a trajectory of the C3X connector is shorter during a move than during another one, the cloud associated with the first move has a high probability to be smaller that the one associated with the second.

This allows us to compute only the end effector trajectory length, i.e. to simulate the movement and add the traveled distance at each time step, without computing the whole cloud (see Fig. 24). The real best cloud can be computed afterward from the movement that leaded to the best trajectory. However, we can't be sure that this cloud will effectively be the best one, but it has a high probability of being at least one of the bests.

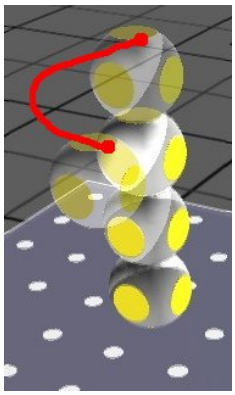

Figure 24: An exemple of end effector trajectory.

3.5.2 Results

The computation time of calculating all trajectories length and the collision cloud of the smallest trajectory is 0.72 seconds on average for the same example as before. This is 14 times quicker than when we use the {sphere + cube} method. If we remove the cloud computation time of 0.020s for the result of 0.72s, we find that this method needs only 0.00137s to compute a trajectory.

As a partial conclusion, we would prefer to use the end effector trajectory length if the results show that it effectively allows to discover the best cloud. However, the {sphere + cube} method will not be useless as we will have to compute the real clouds in any case.

We were wondering if these computation-light methods allow to compute a collision cloud online into a module or in the supervisor for movements that is not in the database. As the computation is still quite complicated, this remains to be proved.

4 Collision clouds: optimization and collision avoidance

In this chapter, we try different strategies to decrease the collision cloud's sizes. We first try a brute force method (sect. 4.1) and then a iterative one (sect. 4.2) that allows less computation and possible better results but can be stuck into local minima. We compare the results of the two methods in section 4.3. Then we explore the possibilities of alternative clouds (sect. 4.4) to avoid obstacles. The results of these three first parts leads us to reexamine our goals and develop a new strategy aiming directly at obstacles avoidance, the milestones strategy (sect. 4.5).

4.1 The brute force algorithm.

This method consists in :

- **Choosing a set of intermediate positions**: the more intermediate positions we have, the best result we can find. We divide each joint angle dimension of 2π rad into $nAngle$ angular steps. Therefore, intermediate positions are all positions whose joint angles are multiples of $\frac{2\pi}{nAngle}$rad (see Fig. 25).

  However, the number of intermediate positions $nPos$ is:

  $$nPos = nAngle^6 \tag{4}$$

  . This expression shows that $nPos$ increases very rapidly with $nAngle$: for $nAngle = 3$ (an angular step of 120 °), there are 729 intermediate positions, for $nAngle = 6$ (an angular step of 60 °), $nPos = 46656$. We can decrease a bit the number of intermediate positions by considering that the last servomotor has very few influence on the shape of the meta-module and therefore on the collision cloud. The number of intermediate positions is reduced to:

  $$nPos = nAngle^5 \tag{5}$$

  .

- **Choosing a number *nInter* of intermediate positions**: 1 or 2 are sufficient. Indeed, a basic strategy to decrease the cloud size is to first move into a very compact position, like a U-shape, move and then deploy again. To do this, we need only two intermediate positions. Increasing the number of intermediate positions increases a lot the computation time.

- **Computing the clouds**: For each set of $nInter$ intermediate positions, compute the collision cloud when the meta-module moves from initial to intermediate(s) to final position. There are $nPos^{nInter}$ different possibilities, each possibility consisting of $nInter + 1$ clouds to compute.

- **Choosing the smallest cloud**: the size of a cloud is the number of collided boxes of the 3d grid. We have to remember the sequence of intermediate positions associated with the cloud.

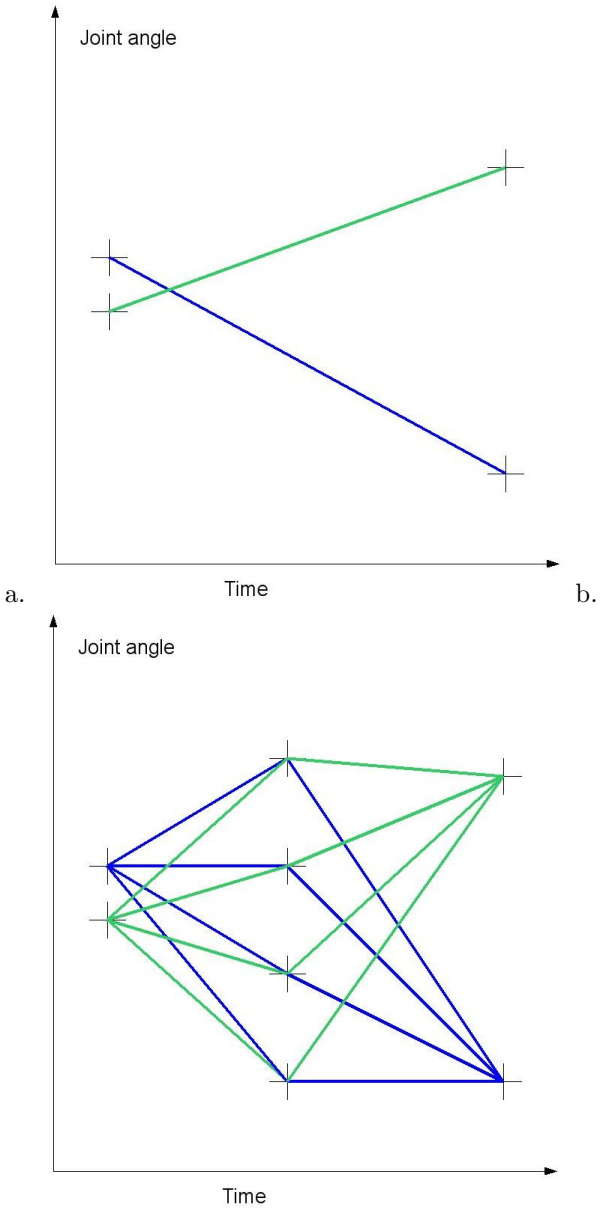

Figure 25: Illustration of the angular space with only two joint angles of the metamodule for simplification. a: the angles move directly from the initial to the final value. b: we test all the movements passing through the intermediate angles.

With this method, we compute all the clouds so we can be sure to find the best collision cloud using the chosen intermediate positions. But as $nAngle$ (the precision) and $nInter$ (the number of possible paths) increase, the number of collision clouds explodes. This method is computationally very intensive. Even if the database is created offline and the computation time is not a major drawback, this method is pretty inefficient, even for small precisions. Using the end effector trajectory length (see section 3.5) allows us to compute very quickly a lot of trajectories. Then we chose only the shortest one and compute the associated cloud. Then the brute force method becomes tractable for an increased precision.

We however look for a method less computationally intensive. The incremental method allows for a linear cost and an increased precision.

4.2 The incremental method

With this method, instead of optimizing with all joint angles at the same time, we want to find the smallest collision cloud incrementally by moving the joint angles of the intermediate positions one by one.

The method follows this procedure :

- **Choose a number *nInter* of intermediate positions**: 1 or 2 are enough.
- **Choose initial intermediate positions**: they can be chosen at random or on the way for the direct move ($\frac{final-initial}{2}$ for example). We also tried to put as initial condition the angles corresponding to a U shape. This shape is the most compact, we can therefore hope that this is close to a local minimum and improve the results (which empirically works in most of the cases, so this is the one we use from now on). This point is illustrated on Fig.26.
- **Choose an angular step *delta_angle***: we chose $\frac{\pi}{nAngle}$, with $nAngle$ between 2 and 8. This step is decreased during the computation (for example, in our implementation, it is divided by two when all angles have been tested once).
- **For all iterations until maxIteration**:
  - **Choose one joint angle of one intermediate position**. The angles can be chosen always in the same order (the solution we implemented) or at random.
  - **Compute the collision cloud for different values of this joint angle** while the other angles remain constant. The value of the joint angle is changed to $value + k.delta\_angle$ with k varying from $-nAngle$ to $+nAngle$. We remember the smallest cloud and put the joint angle to the corresponding value. See Fig.27 and 28.

This method allows us to have a better precision with less cloud computations, with the cost that we can't be sure to find a global optimum.

The number of computed clouds is:

$$nClouds = 2.nAngle.maxIteration \tag{6}$$

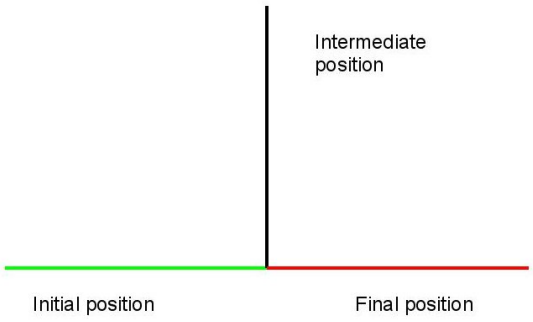

Figure 26: We want to find the intermediate position that produces the optimal cloud when moving from the initial to final position. **Step one**: initialize the intermediate position.

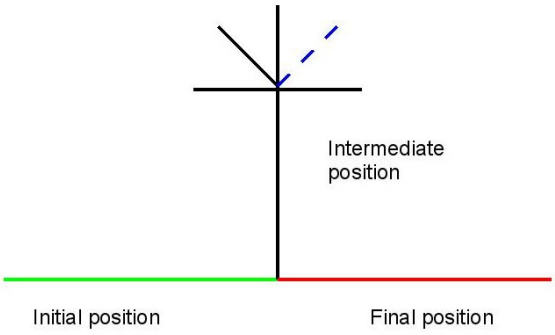

Figure 27: **Step two**: We choose a joint and compute the clouds for different values of its angle, the others remaining constant.

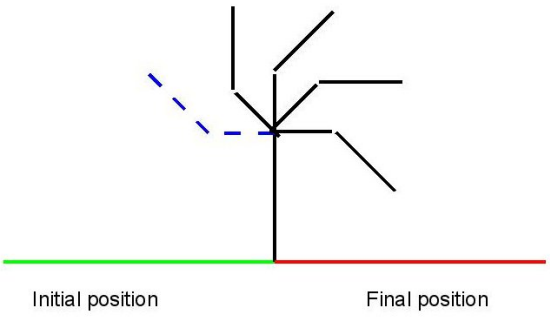

Figure 28: **Step two bis**: The previous joint angle is set the the best found value and we repeat the process with another angle.

To ensure that each joint angle is visited several time (for example 5), maxIteration should be more or equal to:

$$maxIteration \geq 5.nInter.nServo \tag{7}$$

, where $nServo$ is equal to 6 if we want to be very precise or 5 in most of the cases. The total number of computed cloud is linear with respect to each coefficient and not exponential any more. We can afford more intermediate positions and a higher precision, but there is no guarantee that we will find an optimal result.

The end effector trajectory length (EETL) method could be improved if we allow to move several angles at the same time, but the computational cost increases very rapidly. If we move all angle at the same time, we have the brute force approach.

4.3 Results

For time reasons, we consider only 7 shape positions out of 432 possibles for each shape to shape transition. For each pair (49 different pairs, all transitions representing 186 624 pairs) of these positions (one being the initial position and the other the final one), we compute the optimized cloud using different methods and different parameters.

Let start with the brute force algorithm (table 2). It should be noticed that the standard deviations are more representative of the spread of the clouds sizes for the different transitions than of the value of the algorithm:

	A	B	C	D
cloud size mean	54.1 ± 11.5	46.0 ± 17.2	42.9 ± 20.8	40.8 ± 20.0
computation time	516s	45 s	210s	1448s

Table 2: A: brute force algorithm computing all clouds with the {sphere+cube} model. $nInter = 1, nAngle = 4$. Others: using the EELT method. B: $nInter = 1, nAngle = 4$. C: $nInter = 1, nAngle = 6$. D: $nInter = 2, nAngle = 3$.

As expected, we can see that increasing the precision or the number of intermediate positions decreases the average size of the clouds. However, as the computation time increases very quickly, we cannot afford a high precision and the results are therefore not so good.

We also compare the results of the iterative algorithm for different parameters (table 3).:

	A	B	C
cloud size mean	39.3 ± 10.6	38.2 ± 10.9	32.5 ± 14.1
computation time	547s	915s	1653s

Table 3: The intermediate(s) position(s) are put to U-shape. A: $nInter = 2, nAngle = 3, maxIteration = 30$. B: $nInter = 2, nAngle = 3, maxIteration = 50$. C: $nInter = 1, nAngle = 8, maxIteration = 50$.

As expected, the precision increases with $maxIteration$ and with the number

of tested positions (a high $nAngle$ gives better results than a bigger $nInter$). The computation time increases also very rapidly.

Finally, let's compare the best results of each algorithm with the direct cloud computation (table 4):

	A	B	C
cloud size mean	38.4 ± 22.4	40.8 ± 20.0	32.5 ± 14.1
computation time	2s	1448s	1653s

Table 4: A: direct cloud. B: brute force with EETL, $nInter = 2, nAngle = 3$ C: iterative with U-shape, $nInter = 1, nAngle = 8, maxIteration = 50$.

We can see that the brute force algorithm does not find better results than the direct cloud. This can be due to two reasons: as the intermediate positions are not shapes, if the direct cloud is very close from optimality, the brute force algorithm can't mimic it. Or it can be that the end effector trajectory length is not a very good indicator of the cloud size when we look for the smallest one.

The iterative methods leads to a decrease of the cloud size, thanks to its adaptable step, but not very significant when we look at the standard deviation of the sizes of the direct clouds minus the sizes found by the best iterative algorithm: this standard deviation is 11.3. (it's 13.57 for the sizes of the direct clouds minus the sizes found by the best brute force algorithm). The computation time is also increased.

These quite poor results leads us to reconsider our goals. We have first tried to use what we have done to find alternative clouds (sect. 4.4) but we quickly change to the milestones strategy (sect. 4.5).

4.4 Alternative cloud computation

Up to now, we focused on finding the best collision cloud. But, if there are obstacles on this path, we wish to have in our database one or several alternative clouds. A good alternative cloud has the least number of cubes in common with the best cloud, so that it has a high probability of not colliding with the obstacle where the best cloud collides. The size of the alternative cloud is not a problem, but, as previously, a small cloud has less probability to collide.

4.4.1 The cloud distance

We define the **cloud distance** that measures how far away and how different two clouds are. To find the best alternative cloud, we will choose among all clouds (computed with the brute force method) the one that **maximise the distance** to the best cloud. The cloud distance is not a distance in the mathematical sens of the term.

The **cloud distance** of a cloud A compared to the cloud B is defined as follows: for each cube i of cloud A, we find the closest cube of B and call the distance between the cubes d(i). The distance d(A, B) is then defined as:

$$d(A, B) = \frac{1}{cloud\_size\_of\_A} \sum_{i=1}^{cloud\_size\_of\_A} d(i) \tag{8}$$

With this distance, cubes colliding with the best cloud add no term and therefore penalize the cloud when we maximize the distance. Therefore, this definition has a drawback: if a cloud is at a distance D of the best cloud, the same cloud with one more cube not colliding with the best cloud will have a greater distance.

But this distance is not the one we used the most because, if we use the end effector trajectory length, we don't compute all clouds but only the trajectories. We wish to define another distance using only the end effector trajectories.

4.4.2 The trajectories distance

To do this, instead of remembering only the length of the end connector path when we simulate the movement, we store some points representing the trajectory in a list called trajectoryList. During the movement, the speed varies and can be pretty small when approaching a intermediate position, so we don't store all the points. We add a point to the list if it is at a distance more that 1 mm of the previous stored point.

With this trajectoryList, we can define a **trajectories distance**. Two trajectoryLists don't contain the same number of points. We re-sample them to have 100 points in each trajectoryList and we add the distance between the corresponding points as illustrated in the Fig.29.

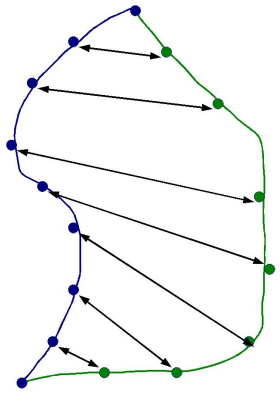

Figure 29: The distance between two trajectories.

However, we can't be sure that two trajectories for which this distance is big will have collision clouds with a high cloud distance as defined above or that their clouds will have few collisions. Indeed, when we tested the alternative clouds, we could observe that the alternative trajectory found wasn't respecting these criteria.

4.5 The milestones strategy for obstacle avoidance

The conclusion of the brute force and the incremental methods was that, often, the direct movement is close from optimality, particularly when the initial and

final positions are close. Indeed, in this case, intermediate positions only add parasite movement that increase the collision cloud size.

Moreover, if we adopt a larger point of view, minimizing the collision cloud or finding an optimal alternative cloud have as final objective to avoid obstacles. We decided to take some distance to collision cloud minimization and to find a way of directly avoiding obstacles.

4.5.1 Conclusion of the other methods and new strategy

The authors of [17] adopt an interesting approach. Their goal is to control one or several robot(s) in a collision free way in an environment with obstacles. To do this, they sample the space at random with *milestones*, which are collision-free positions. They adapt the sampling to the space to have a higher precision in narrow passages. Then they construct a *probabilistic road map* by connecting the milestones with simple collision-free paths, that they call local paths. Finally, using a two-rooted tree, they construct a path from the initial to the final configuration by concatenating local paths.

This method is based on the assumption that any short connection between two collision-free configurations has high prior probability of being collision free. The author also tries to reduce the number of collision checking, which are the most expensive parts of any planning algorithms.

4.5.2 The milestones strategy for Roombots

In this paper ([17]), the authors try to reduce the computational time by reducing the number of collision checking but don't try as we do to reduce the online computational load at most by using precomputation. Therefore, we won't implement directly this algorithm but take it as inspiration to design a *dynamic collision free path finder*, using an *intermediate position to intermediate position collision cloud database*.

Our algorithm is divided into two major phases, one off line, the database computation (see Fig.30 to 32) and one online, the collision free path finding (see Fig.33 to 36).

The offline phase consists in:

- sampling the space with intermediate positions
- defining a *neighborhood distance* between positions and a neighborhood relationship based on this distance; storing the list of neighbors for each position.
- computing and storing the collision cloud between each couple of neighbor positions

The online phase consists in a recursive algorithm that finds a collision free path between the initial and final position with a A* search (see [23]), based on a distance between the neighbors of the initial position and the final position.

It remains now to define more precisely how we sample the space and how the intermediate positions are defined, what distance we use to define the neighborhood and how we order the neighbors during the path finding.

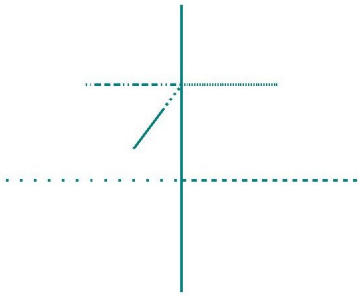

Figure 30: **First step (offline)**: define all possible intermediate positions.

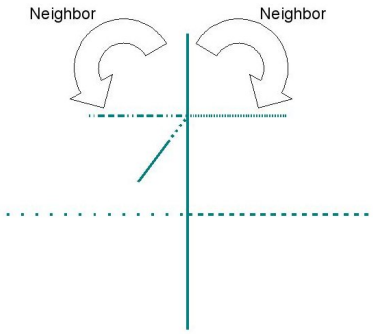

Figure 31: **Second step (offline)**: for each position, list all neighbors.

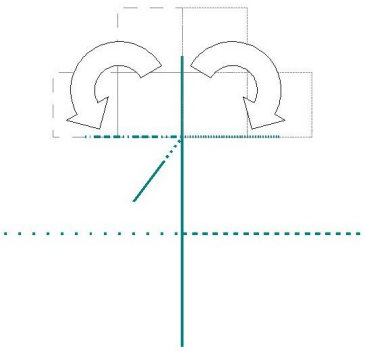

Figure 32: **Third step (offline)**: compute the clouds of neighbor positions.

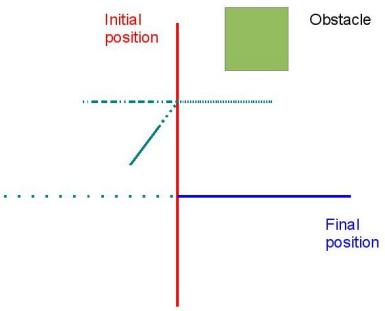

Figure 33: **First step (online)**: The module wants to go from initial to final positions. There are obstacles in the environment.

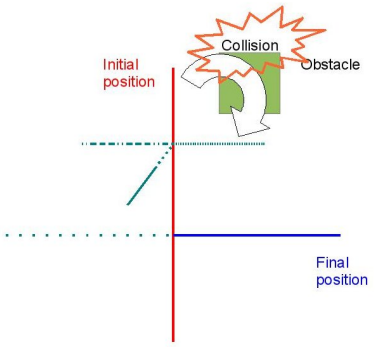

Figure 34: **Second step (online)**: Choose the neighbor closest to the final position. Unfortunately, there is an obstacle on the path.

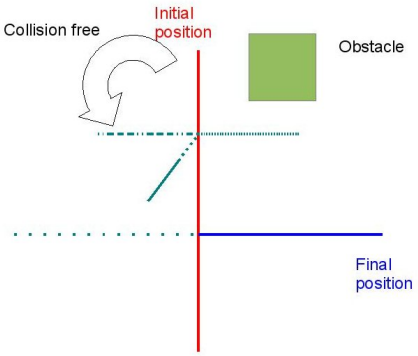

Figure 35: **Second step bis (online)**: Choose the next neighbor closest to the final position. The move is collision free.

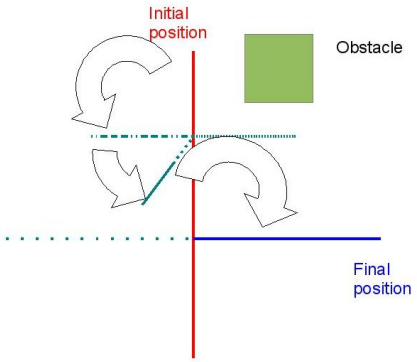

Figure 36: **Second step completed (online)**: A collision free path has been found.

4.5.3 Details on space sampling and different distances

The **space sampling** is done in joint angle space, as previously. Our goal is to have enough intermediate positions spread out in the whole reachable space to have a large choice of paths. We have therefore a good precision in the moves and the possibility to move into narrow spaces. However, more samples imply a larger database and more computation to find a collision free path.

It is important that the shape positions are intermediate positions, as initial and final positions will be shapes. For the moment, we use as intermediate positions all shapes positions, because our goal is more to prove the viability of the method than to actually implement it for the robot. But we will also try to quantify the amount of computation if we increase the precision.

We though of sampling the real space, i.e. instead of choosing the intermediate positions in the joint angle space, choosing some end effector positions (the end effector position being here the center of the last Roombots element). However, we would have to choose different orientations for each positions and then compute the inverse kinematics to find the joint angles associated with this positions. Sampling the joint angle space is easier and it is also easier to find the neighbors. Moreover, the shape positions already lead to different positions and orientations in the 3D grid. Therefore, we didn't try to sample the real space.

The **neighborhood distance** is computed from the joint angles as well as the end effector positions. Indeed, we want two intermediate positions to be neighbors if we can move from one to the other easily and quickly : the joint angles can all change between the initial and final positions, but the maximal angular difference is bounded.

However, depending on the position of the robot and the servos, moving slightly one joint angle can lead to a big displacement of the whole robot and particularly of the end effector, leading to a big collision cloud, which is costly to compute and has more probability to collide with the obstacles. To avoid this, we also impose that the distance between the two end effector positions is less than a threshold to consider two positions as neighbors.

The threshold for joint angles and end effector positions must be chosen such that each position has a number of neighbors big enough to have a high

probability of finding a collision free path, but small enough for the amount of computation to be reasonable. With a threshold of 120 ° for the angles and of 16cm for end connector distance with only shapes as intermediate positions, each position has between 4 and 25 neighbors.

When constructing a collision free path with the A* search algorithm, we test first the position that are the closest from the end position. The distance used here, called **path finding distance** is the distance between end effectors. We don't need to take into account the joint angles because we can only move from one position to its neighbors and, by definition, two neighbors have close joint angles.

But several intermediate position may have the same end effector position. This causes some problems when we arrive to the goal position as we also want to reach the right angles. Therefore we add in the path finding distance, with a very small coefficient, the sum of angular differences. If the end effector distance is non zero, this will have almost no influence, but if the end effector distance is null, the path finder will choose the position with all joint angles corresponding to the goal position.

4.5.4 Details on the computational load.

The last problem is about computational load. We now have much more online computation to find a collision free path than with the previous method, when the module only had to ask for one cloud. We have to decide who does the computation :

- **case 1**: Everything is done by the supervisor. The module sends one message with the initial and final positions and the obstacle list and the supervisor does the computation, the collision checking and returns the collision free path if there exists one. This has the advantage of asking for less communication but has the drawback that it is less scalable: the more computational load in the supervisor, the less module it can handle in a limited amount of time. Moreover, the more we depend on the supervisor, the less robust we are.

- **case 2**: The computation is shared but mostly done in the supervisor. The supervisor can send to the module every intermediate positions it considers and the module answers if the path is collision free or not. There is relatively few communication if the path is quickly found but a lot if several paths have to be tested.

- **case 3**: The computation is shared but mostly done in the module. The module does everything and only asks for the collision clouds of the different steps. There is as much communication as in the previous case. This supposes the list of neighbors for each position is stored in the memory of the module. If not, the module also has to ask for the list of neighbors or directly for the best neighbor (but if the supervisor has to find the best neighbor, we are very close to case 2)

We have not decided yet which solution is the best because we need to have a better idea of the computational loads implied.

4.5.5 Implementation and results

Because of time constraints of the project and the previously discussed problems of real time computational loads for the module and the supervisor, we did not implement the whole method usable by a real Roombots module, but we coded a *proof of concept* algorithm to prove the viability and performances of this method.

Our function **obstacleAvoidance** takes in arguments the initial and final positions (in joint angle space) and the list of obstacles. It returns the list of intermediate positions of a collision free path. It creates a table tested which contains the boolean value false for each intermediate position. This table will allow us to avoid testing twice the same position and creating loops in our path. The function obstacleAvoidance calls the function findPath.

The function **findPath** takes the same arguments and return a collision free path. It does the following:

- **If** the initial and final positions are the same, it returns true.
- **Else** it computes all reachable neighbors of the initial position and store them in a list neighborList. A position will be in the list if it satisfy the conditions of the **neighborhood distance**. and if the collision cloud between the initial position and the neighbor does not collide with the environment.
- The function chooses the reachable neighbors the closest to the end position, among all neighbors with value *false* in the tested table, using the **path finding distance**. A neighbor can't be chosen if it collides with the environment (we delay the collision computation as much as possible to do it the less often). It put the corresponding value in the table tested to true and calls the function **findPath** with this neighbor as initial position. If findPath returns true, we have found a collision free path.
- If **findPath** returns false, we try with the second closest neighbor. If all neighbors have been tested, the function returns false.

The difference with the real implementation is the neighbor calculation: we create the list, calculate the cloud and check the collision online in the same program. We record the number of cloud computation to have an idea of how much communication and database checking we will need, as well as the remaining computation time.

A simple movement in a free environment can ask for less that 20 cloud checking and the solution can be found in 0.3 seconds (see Fig. 37). In a more complicated case, when there are obstacles in the smallest path, the algorithm may compute up to 70 clouds and the path is found in 1.5 seconds (see Fig. 38). When we compare with the time needed to compute the clouds, we can see that the greatest proportion of the time is used for cloud computation and less than 0.1 seconds are needed to find the neighbors and the path. This seems promising for real applications if we don't have too much modules moving at the same time.

As a conclusion, this first results are encouraging but some problems remain: what is the optimal sampling of the space (more intermediate positions allow to

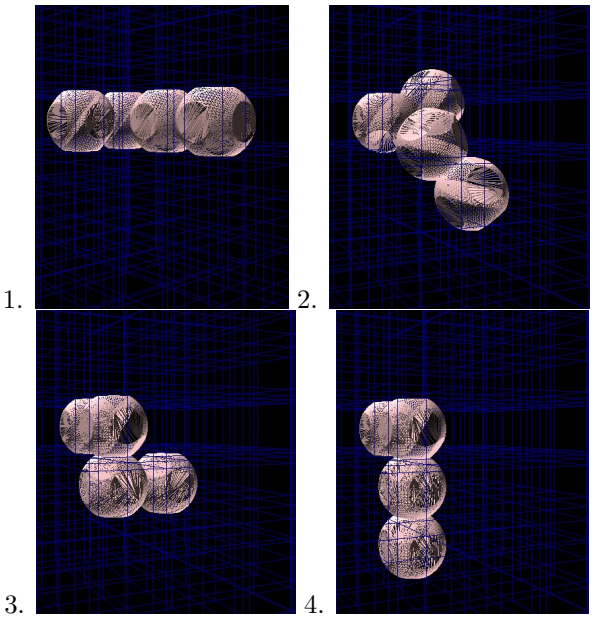

Figure 37: The meta-module goes from a position to one of its neighbors and not directly from the initial (image 1) to the final one (image 4).

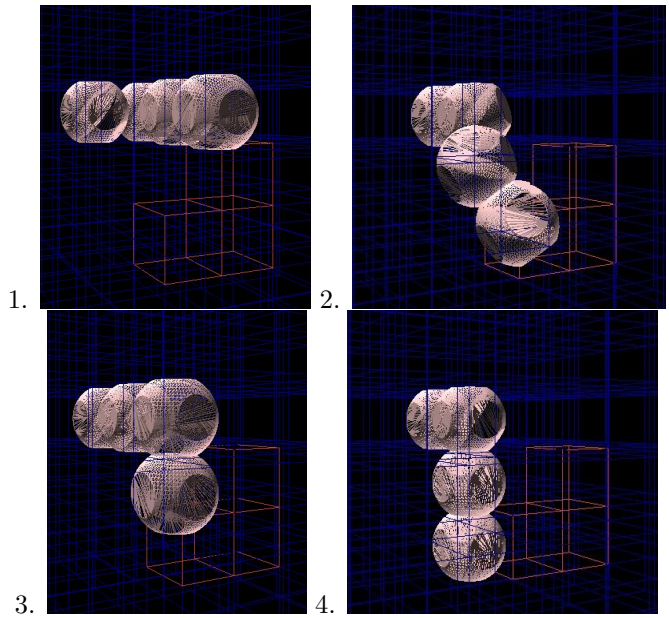

Figure 38: For the same movement in an environment with obstacles, the algorithm finds a collision free path.

find narrow passages but increase the computation time)? What is the corresponding optimal neighborhood distance ? Can the modules and the supervisor handle the increased computational load ?

5 Conclusion

5.1 Summary and results

In this project, we improved the way the Roombots move and reconfigure through locomotion.

5.1.1 Optimized approaching movements

In chapter 2, we improved the approaching movements of meta-modules to lock safely on a connector, using the virtual connector method. We defined the docking angle to measure the improvement and achieved a average docking angle of $1.3\,^\circ \pm 1.3\,^\circ$, to compare to the initial value of $4.0\,^\circ \pm 2.5\,^\circ$.

5.1.2 Collision clouds : computation

In chapter 3 and 4, we focused on the collision clouds. To avoid collisions, our initial approach was to decrease the size of the collision clouds. Therefore, we added intermediate positions during the cloud computation and optimized these positions to decrease the cloud size.

To do this, we first focused on the collision cloud computation (chapter 3). We used two method to decrease the computation time.

In the {sphere + cube} method, we model the Roombots and use geometric properties to compute the cloud. We decrease the computation time by a factor 25, the average time for computing a cloud passing from 0.48s to 0.020s.

Then we used the End Effector Trajectory Length method. With this method, we estimate the best intermediate positions by only computing the trajectory of one point on the end connector. Then we compute only the cloud associated with the shortest trajectory. This method allows to find an optimized cloud with $nAngle = 4$ in 0.72s, this is 14 times faster that with the {sphere + cube} method.

5.1.3 Collision clouds: optimization

With our fast cloud computation, we tested several methods to find the optimal intermediate positions.

The brute force method consists in testing all configurations of intermediate positions. It is very costly even for a low precision. The use of the end effector length trajectory method allows to increase the precision but the results don't show any improvement when we compare the cloud size to the one obtained with the direct cloud as computed previously.

The iterative method tries to find good intermediate positions by optimizing joint angle by joint angle, with a precision increasing with time. By doing this, we have a computation time linear with respect to each parameter and we can therefore reach a better precision for the same computation time. We can however be stuck in local minima. This method proves better results that the direct cloud computation (average size of 32.5 against 38.4), but not very significant regarding the standard deviation of the difference (11.3).

Therefore, it is better to use the incremental algorithm, even if the results are quite poor.

We tried to find interesting alternative clouds to face the case of an obstacle in the best cloud by defining between clouds and between tracjectories distances.

Then we adopted another strategy to directly find a collision free path in an environment with obstacles. This milestones strategy is inspired by [17]. The offline phase consists in defining intermediate positions and a neighborhood. Then we construct a database of collision clouds between neighbor positions. The online phase consists in finding a collision free path linking neighbor positions.

Lack of time for the project prevented us to implement the method for Roombots modules communicating with a supervisor, but we did a *proof of concept* algorithm that needed to compute between 20 and 70 clouds. We calculated that, removing the time needed for cloud calculation, this algorithm need approximately 0.1 second to find a collision-free path on a computer. This is the online part so we can hope that this is tractable for a Roombots module.

5.2 Future work

There are many ways of continuing my project, first by actually implementing the milestones strategy with different modules and a supervisor. Then the models we have should be adapted or replaced to match the real robot dynamics and controller. Then, of course, one can try to handle objects that are not plates or that are so heavy that a single module can't manage them alone, by implementing a cooperative method.

5.3 Acknowledgements

This project allowed me to discover the Webots software more extensively and to improve my C++ coding skills. I was happy to be part of a bigger project, the Roombots, that I could see evolve (the first hardware Roombots were made during the project), with many people working in different fields but on the same project.

I would like to thank:

- my supervisors, Alexander Spröwitz and Stephane Bonardi for their advices and the time they spend discussing the different possibilities and problems faced during this project,
- also Alessandro Crespi for his help and his patience,
- my fellow students for their support.

6 References

References

[1] BIOROB. http://biorob.epfl.ch/. Biorobotics Laboratory.

[2] Cornell University Computational Synthesis Lab. Molecubes for everyone ! http://www.molecubes.org/.

[3] D. Daidié, O. Barbey, A. Guignard, D. Roussy, F. Guenter, A. Ijspeert, and A. Billard. The dof-box project: An educational kit for configurable robots. http://lasa.epfl.ch/publications/uploadedFiles/papier_DOF_BOX_def.pdf.

[4] BIOROB EPFL. Roombots: Modular robotics for adaptive and self-organizing furniture. http://biorob.epfl.ch/page38279.html.

[5] Dario Floreano and Claudio Mattiussi. *Bio-inspired Artificial Intelligence*, chapter 7.4: Swarm Robotics. MITpress, 2008.

[6] Palo Alto Researche Center Incorporated. Parc modular robotics : telecube. http://www2.parc.com/spl/projects/modrobots/lattice/telecube/index.html.

[7] Jean-Michel JOLION. Newton-raphson algorithm (fr). http://rfv.insa-lyon.fr/%7Ejolion/ANUM/node17.html.

[8] Philippe Laprade. Distributed roombot locomotion and self-reconfiguration, 2009.

[9] Lego its the future ! http://jezzbean.wordpress.com/2010/01/01/lego-its-the-future/1197359210_lego-art-7/.

[10] Jocelyne Lotfi. Self-reconfiguration for adaptive furniture, 2009.

[11] Mickael Mayer. Roombot modules - kinematics considerations for moving optimizations, 2008.

[12] Mikaël Mayer. Movie: Roombot-back-chair-making on the biorob website. http://birg.epfl.ch/webdav/site/birg/users/183900/public/08roombot-back-chair-making.avi.

[13] R. Moeckel, C. Jaquier, K. Drapel, E. Dittrich, A. Upegui, and A. Ijspeert. Yamor and bluemove - an autonomous modular robot with bluetooth interface for exploring adaptive locomotion. http://lslwww.epfl.ch/~upegui/docs/CLAWAR05.pdf.

[14] Manon Picard. Movie: handling passive plates on the biorob website. http://biorob2.epfl.ch/utils/movieplayer.php?id=94.

[15] University of Southern California Polymorphic Robotics Laboratory. Conro: self-reconfigurable robots. http://www.isi.edu/robots/conro/.

[16] The Ocoros Project. Ocoros kinematics and dynamics. http://www.orocos.org/kdl.

[17] G. Sanchez and j.C. Latombe. On delaying collision checking in prm planning: Application to multi-robot coordination. *The International Journal of Robotics Research*, 2002.

[18] Kasper Stoy. *Emergent Control of Self-Reconfigurable Robots*. PhD thesis, University of Southern Denmark, 2004.

[19] Stanford University. Polypod. `http://ai.stanford.edu/users/mark/polypod.html`.

[20] Webots. http://www.cyberbotics.com. Commercial Mobile Robot Simulation Software.

[21] Webots. Webots reference manual. `http://www.cyberbotics.com/cdrom/common/doc/webots/reference/reference.html`.

[22] Sandra Wieser. Locomotion in modular robotics, roombot modules, 2008.

[23] Wikipedia. A* search algorithm. `http://en.wikipedia.org/wiki/A*_search_algorithm`.

[24] Wikipedia. Wikipedia self reconfiguring modular robots. `http://en.wikipedia.org/wiki/Self-reconfiguring_modular_robot`.

A Different parts of the Roombots

As we work on different scales on the Roombots and the names of all part are no always straightforward, this appendix should help the reader to follow our various discussions, by showing the difference between a submodule (Fig. 39), an element (Fig. 40), a module (Fig. 41) and a meta-module, which is the association of two modules. There are two meta-modules on each figure.

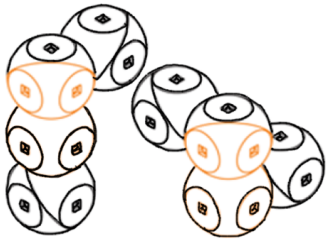

Figure 39: The colored part is a Roombots submodule (picture from [10]).

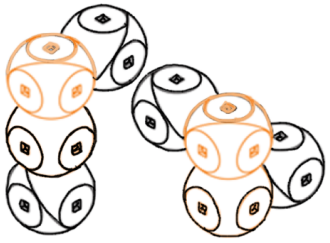

Figure 40: The colored part is a Roombots element (picture from [10]).

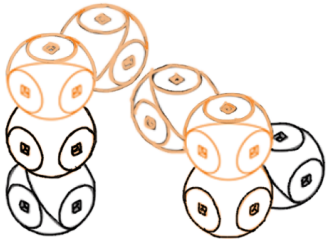

Figure 41: The colored part is a Roombots module (picture from [10]).

B Geometric elements

A Roombots submodule has the shape of a half-sphere intersected with a half-cube. We name **sphere** the sphere and **cube** the cube or the two half cube defining the shape of a Roombots element.

We name **box** a cube of the 3d grid of the environment.